31st Euromicro Conference on Real-Time Systems

ECRTS 2019, July 9–12, 2019, Stuttgart, Germany

Edited by

Sophie Quinton

Editor

Sophie Quinton
INRIA Grenoble Rhône-Alpes, France
sophie.quinton@inria.fr

ACM Classification 2012
Computer systems organization → Embedded and cyber-physical systems; Computer systems organization → Real-time systems; Software and its engineering → Real-time systems software; Software and its engineering → Real-time schedulability

ISBN 978-3-95977-110-8

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-110-8.

Publication date
July, 2019

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

Digital Object Identifier: 10.4230/LIPIcs.ECRTS.2019.0

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

Preface

Message from the Chairs

Welcome to the **31st Euromicro Conference on Real-Time Systems (ECRTS 2019)** in Stuttgart, Germany. ECRTS is the premier European venue for presenting research into the broad area of real-time systems. Along with RTSS and RTAS, ECRTS ranks as one of the top three international conferences on this topic. ECRTS has been at the forefront of recent innovations in the real-time community such as artifact evaluation and open access proceedings.

For ECRTS 2019, we received **80 submissions** from 19 countries, 7 of which are outside Europe and represent 37% of the submitted papers. Each submission was reviewed by at least three members of the program committee – all active researchers and experts in their field – with the help of 77 external reviewers. The submissions were evaluated according to their contribution, originality, technical correctness and writing quality. The program committee then selected, at a physical meeting of the program committee in Paris, 27 of these submissions for publication in the proceedings and presentation at the conference.

From the 27 accepted papers, three have been recognized as **outstanding papers** by the program committee and will be presented in the final session. One of these three papers will be selected as **best paper** by a dedicated committee, based on both the contribution of the paper and the presentation at the conference.

In 2016, ECRTS was the first conference on real-time systems to introduce an **artifact evaluation**, with the aim to promote reproducibility of research results. An artifact evaluation committee reviews the artifacts submitted by the authors of accepted papers who choose to do so. In 2019, eight papers (30% of the accepted papers) are marked in the proceedings with a seal indicating that their artifact has passed the repeatability test.

In 2017, ECRTS was the first conference on real-time systems to introduce an **open access publication model**, while retaining the existing quality-control measures. The open access model uses LIPIcs – Leibniz International Proceedings in Informatics, a series of high-quality conference proceedings established in cooperation with Schloss Dagstuhl, Leibniz Center for Informatics. The conference serves the research community and the public best when results are accessible to the largest audience, i.e., the research community and the public. This year again, the proceedings will be accessible free of charge for everyone.

The workshops of ECRTS, that take place the day before the main conference starts, are a key feature of the event. They are widely acknowledged to be lively and useful to the community. ECRTS 2019 will host the following workshops:

- **CERTS** – 4th International Workshop on
  Security and Dependability of Critical Embedded Real-Time Systems
- **OSPERT** – 15th International Workshop on
  Operating Systems Platforms for Embedded Real-Time Applications
- **RTN** – 17th International Workshop on Real-Time Networks
- **WATERS** – 10th International Workshop on
  Analysis Tools and Methodologies for Embedded and Real-time Systems
- **WCET** – 19th International Workshop on Worst-Case Execution Time Analysis

ECRTS 2019 will start with a **keynote** by **Thomas Kropf**, president of corporate research and advance engineering at Bosch. A second keynote will be given by **David Monniaux**, senior researcher at CNRS and head of the PACSS team (Proofs and Code analysis for Safety and Security) at Verimag.

The **interactive session** of ECRTS 2019 is one of the most attractive events at the conference. It will feature:

- a **work-in-progress** session where novel ideas are introduced to the ECRTS audience;
- **journal-to-conference** presentations where work so far only published in journals can be presented to the conference audience;
- an overview of the **industrial challenges** discussed at the WATERS workshop to foster collaboration between academia and industry;
- an **industry pitch session** with industry speakers pitching their current and future problems related to real-time issues.

The interactive session will be directly followed by a poster session and reception where all participants can exchange ideas in a relaxed and friendly atmosphere. Submissions to the work-in-progress, journal-to-conference and industry pitch sessions have been evaluated separately by dedicated committees, and are not part of these published proceedings.

Thanks to our location on the premises of Bosch in Stuttgart this year, this edition of ECRTS will represent an excellent opportunity for researchers and practitioners from academia and industry to meet, interact and initiate collaborations. In addition to the special industry focus of the first day, participants will have the chance to look behind the scenes of the Bosch research campus on the second day of the conference during a guided tour, to get insights into the different fields of research conducted at Bosch.

ECRTS 2019 is the result of the hard work of many people, whose names are listed in the following pages. We are especially grateful for the contributions of the **program committee** and the **external reviewers** for carefully reviewing the submitted papers and helping us build the high-quality program of ECRTS 2019; the **artifact evaluation chairs** and the **artifact evaluators** who help this conference pave the way for reproducible research; the **work-in-progress and journal-to-conference chair** and the **workshop chairs** for their hard work in organizing these key moments of the conference; and Dagstuhl Publishing for their support in publishing these proceedings. Many thanks to the **organization committee** and **Robert Bosch GmbH** for their help with the logistics. We also thank **Sebastian Altmeyer** for sharing his experience as the ECRTS 2018 program chair, and **Gerhard Fohler** for his steady guidance and support as the Euromicro Real-Time Technical Committee Chair.

Last, but not least, we thank all the authors who submitted their work to ECRTS 2019. This conference would not exist without them and we are proud of the high quality and scientific relevance of this year's program. Let us now enjoy ECRTS 2019!

Arne Hamann and Dirk Ziegenbein
General Chairs, ECRTS 2019

Sophie Quinton
Program Chair, ECRTS 2019

Committees

General Chairs

Arne Hamann and Dirk Ziegenbein, Robert Bosch GmbH, Germany

Program Chair

Sophie Quinton, INRIA Grenoble Rhône-Alpes, France

Real-Time Technical Committee Chair

Gerhard Fohler, TU Kaiserslautern, Germany

Organization Committee

Steve Goddard, University of Nebraska-Lincoln, USA
Martina Maggio, Lund University, Sweden

Artifact Evaluation Chairs

Sebastian Altmeyer, University of Amsterdam, The Netherlands
Alessandro Papadopoulos, Mälardalen University, Sweden

Work-in-Progress and Journal-to-Conference Chair

Andrea Bastoni, SYSGO AG, Germany

Workshop Chairs

CERTS – Security and Dependability of Critical Embedded Real-Time Systems
Mikael Asplund, Linköping University, Sweden
Michael Paulitsch, Intel, Germany

OSPERT – Operating Systems Platforms for Embedded Real-Time Applications
Adam Lackorzynski, TU Dresden / Kernkonzept, Germany
Daniel Lohmann, Leibniz Universität Hannover, Germany

RTN – Real-Time Networks
Guillermo Rodriguez-Navas, Nokia Bell-Labs, Israel
Ramon Serna Oliver, TTTech, Austria

WATERS – Analysis Tools and Methodologies for Embedded and Real-time Systems
Claire Pagetti, ONERA / IRIT-ENSEEIHT, France
Selma Saidi, TU Hamburg, Germany

WCET – Worst-Case Execution Time Analysis
Sebastian Altmeyer, University of Amsterdam, The Netherlands

Program Committee

Benny Akesson, ESI (TNO), The Netherlands
Sebastian Altmeyer, University of Amsterdam, The Netherlands
Jim Anderson, The University of North Carolina at Chapel Hill, USA
Andrea Bastoni, SYSGO AG, Germany

Artifact Evaluators

Additional Reviewers

Aakash Soni
Aaron Willcock
Abhishek Singh
Alexandre Venito
Ali Syed
Ankit Agrawal
Antonio Paolillo
Arpan Gujarati
Bo Peng
Braham Lotfi Mediouni
Catherine E. Nemitz
Cédric Ternon
Charlotte Seidner
Christoph Lambert
Clara Hobbs
Clément Ballabriga
Corey Tessler
Dakshina Dasari
Daniel Casini
Dirk Ziegenbein
Eberle Rambo
Enrico Mezzetti
Falk Wurst
Florian Heilmann
Gautam Gala
Gautham Nayak Seetanadi
Gowher Parry
Hamid Tabani
Ignaco Sañudo Olmedo
Jacques Combaz
James Robb
Jaume Abella
Jean-Luc Béchennec
Jean-Michel Dricot
Jérôme Ermont
Johannes Schlatow
Jordi Cardona
Jordy Ruiz
Jorge Martinez
José Carlos Palencia Gutiérrez
Juan M. Rivas
Junkil Park
Kevin Delmas
Konstantinos Bletsas
Kristin Krüger
Leonidas Kosmidis
Leonie Köhler
Manohar Vanga
Marc Boyer
Marco Solieri
Micaela Verucchi
Mikaël Briday
Muhammad Soliman
Nathan Otterness
Nathanaël Sensfelder
Nicola Capodieci
Oana Hotescu
Paolo Burgio
Pedro Benedicte
Radoslav Ivanov
Rany Kahil
Roberto Cavicchioli
Robin Hofmann
Rodrigo Coelho
Sebastian Tobuschat
Sergey Bozhko
Sergey Voronov
Stefanos Skalistis
Stephen Tang
Syed Aftab Rashid
Tanya Amert
Thomas Loquen
Tobias Blaß
Tomasz Kloda
Victor Milnert
Xavier Poczekajlo
Zahaf Houssam Eddine

DMAC: Deadline-Miss-Aware Control

Paolo Pazzaglia
Scuola Superiore Sant'Anna, Pisa, Italy
Department of Automatic Control, Lund University, Sweden
paolo.pazzaglia@sssup.it

Claudio Mandrioli
Department of Automatic Control, Lund University, Sweden
claudio.mandrioli@control.lth.se

Martina Maggio
Department of Automatic Control, Lund University, Sweden
martina.maggio@control.lth.se

Anton Cervin
Department of Automatic Control, Lund University, Sweden
anton.cervin@control.lth.se

Abstract

The real-time implementation of periodic controllers requires solving a co-design problem, in which the choice of the controller sampling period is a crucial element. Classic design techniques limit the period exploration to *safe* values, that guarantee the correct execution of the controller alongside the remaining real-time load, i.e., ensuring that the controller worst-case response time does not exceed its deadline. This paper presents DMAC: the first formally-grounded controller design strategy that explores shorter periods, thus explicitly taking into account the possibility of missing deadlines. The design leverages information about the probability that specific sub-sequences of deadline misses are experienced. The result is a *fixed* controller that on average works as the ideal clairvoyant time-varying controller that knows future deadline hits and misses. We obtain a safe estimate of the hit and miss events using the *scenario theory*, that allows us to provide probabilistic guarantees. The paper analyzes controllers implemented using the Logical Execution Time paradigm and three different strategies to handle deadline miss events: killing the job, letting the job continue but skipping the next activation, and letting the job continue using a limited queue of jobs. Experimental results show that our design proposal – i.e., exploring the space where deadlines can be missed and handled with different strategies – greatly outperforms classical control design techniques.

2012 ACM Subject Classification Computing methodologies → Computational control theory; Computer systems organization → Embedded software; Software and its engineering → Real-time systems software; Theory of computation → Stochastic control and optimization

Keywords and phrases Weakly-Hard Real-Time Systems, Deadline Miss Handling, Control Design

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.1

Supplement Material ECRTS 2019 Artifact Evaluation approved artifact available at
`https://dx.doi.org/10.4230/DARTS.5.1.3`

Acknowledgements All authors from Lund University are part of the ELLIIT Excellence Center. This work was partially supported by the Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation.

1 Introduction

Controllers are often executed alongside other tasks in a real-time platform, demanding that the scheduler ensures the timely execution of both the controller and the real-time workload that the platform should execute. Controllers can be designed taking into account resource limitations and scheduling constraints [16, 55, 54].

Studying the optimal design of a control task to be run alongside a given real-time workload can be considered an instance of the general problem of *composability*. Composability is the capability to integrate new functionalities into a preexisting system. This issue is particularly relevant in the automotive field, where the production of new vehicles requires a tight coupling of new software together with legacy code, with minimal adjustment of the original structure. In general, adding a new control task to a given taskset implies combining requirements that come from both control theory and real-time implementation. These requirements are different and often conflicting. As an example, selecting a high execution rate for the controller improves the control performance, but at the same time limits the guarantees on the timely completion of the control task code and forces the engineers to take into account overruns [13, 41]. Moreover, minimizing the monetary cost of the final system is an ever-present priority and over-provisioning resources is usually not a viable solution.

Timing constraints in real-time systems are modeled as *deadlines*, i.e., a threshold that the execution time of each task instance (*job*) should respect. We refer to a job that successfully completes its execution before the corresponding deadline as a *deadline hit* event. If the job could not terminate its execution before that deadline instant, we say that it *missed its deadline*. In *hard* real-time systems, missing a deadline has been always seen as a risk that must be avoided, with possibly catastrophic consequences. In reality, a limited number of deadline misses is an acceptable condition for many cyber-physical and control systems, since well-designed controllers often expose intrinsic *robustness* to timing non-idealities. Recently, researchers have then tried to formally relax deadline constraints, introducing the *weakly hard* real-time system paradigm [7] to describe the case where tasks are allowed to miss a limited number of deadlines. However, often control engineers lack information about the timing behavior of the control task and the taskset structure. Understanding how a control loop behaves under deadline misses may open the door to new and better control designs.

Inspired by this challenge, in this paper we tackle the problem of designing a controller for a generic physical plant, while exploring a range of periods which have historically been avoided for co-design: we are here interested in those period values that are *shorter* than the worst-case response time of the task, thus neglecting the common hypothesis of hard deadlines. Our design problem is to run the controller alongside a preexisting taskset. Tasks are described with probabilistic execution times, ranging from a best case to a (rare) worst case value. By leveraging the flexibility of robust control design techniques, we here propose a novel method for creating an optimal fixed controller, the *Deadline-Miss-Aware Control* (DMAC), which can be implemented in a real-time task that may miss some deadlines.

The DMAC design takes into account how the controlled system behaves when different patterns of hit and missed deadlines occur. For robustness, DMAC considers a safe (pessimistic) probability of deadline miss events. Lack of scalability impedes the computation of deadline miss probabilities analytically. However, bounds are extremely pessimistic and would not aid the control design method. To overcome this limitation, we obtain an estimate of deadline miss occurrence simulating the schedule execution, drawing execution times (for all the tasks) from the corresponding probability distributions. A robust control tool, the *scenario theory* [11], provides the means to select the worst-case sequence of misses and hits from the simulations. Leveraging the scenario theory, our approach allows us to provide probabilistic guarantees for worst-case conditions both in terms of the probability of not having taken into account conditions that will eventually manifest, and in terms of the design confidence. We obtain a controller which is optimal and robust to worst-case conditions.

The analysis presented in this paper considers three different strategies for handling deadline misses: kill the job that missed the deadline, let it continue and skip the next job, or let all jobs continue until completion, but limiting the ready queue to the most recently activated among the pending jobs (in addition to completing the one that missed the deadline). We implement the controller following the Logical Execution Time (LET) paradigm [28]. To the best of our knowledge, this is the first attempt to design an optimal controller for a real-time system that is aware of deadline misses and miss-handling strategies.

2 Methodology

The aim of DMAC is to provide the first control synthesis method that is *robust* both with respect to deadline misses and with respect to the strategy used to handle them. Our control design leverages knowledge of the probability that different sequences of deadline hits and misses may occur, and produces a fixed controller that is (on average) optimal with respect to a defined cost function. We obtain such knowledge by formulating a chance constrained optimization problem in a probabilistic framework, and obtaining guarantees on both the probability of neglecting important information and the confidence in the design.

2.1 Overview and Terminology

We present here an overview of the approach adopted for the control design and evaluation. For the application of the approach we rely on the following input data:

- ε: The user selects a defined value of ε, that represents a worst case bound on the probability of carrying out the control design operations neglecting important information. We rely on the generation of sequences of deadline hits and misses, from which we select the worst case and design our controller to be robust with respect to such worst case. While ε can be selected to be as small as possible, we still need to accept a certain small probability that the next generated sequence would be worse than the worst case generated up to the current one.
- $1-\beta$: The user selects a value for the confidence that one can have in the probabilistic guarantee that the approach is providing. The value of $1-\beta$ is determined together with the value of ε, to indicate that the approach is based on a confidence $1-\beta$ that the probability ε is the true probability of missing important information.
- Γ: The taskset that our design is targeting. We assume that additional (hard real-time) load is run alongside the controller task.
- n_{job}: Our control design is based on extracting timing behavior from simulations of a certain number of control jobs. The length of the sequences used for the controller design can be chosen depending on physical parameters, for example assuming that after a certain number of jobs the controller has settled. We recommend to select a value that contains at least a few hyperperiods, to capture chain effects if they happen.
- J_{seq}: The cost function that is used to evaluate the produced sequences to select the worst-case sequence for the controller design.
- ξ: The strategy used to handle a deadline miss. We consider three different strategies: killing the job that missed the deadline, letting it continue and skipping the next job, or letting it continue and enqueuing the next job (up to a maximum of one enqueued job at any point in time).
- J_{ctl}: The cost function that is used to evaluate the controller behavior and compare the different deadline miss handling strategies.

Figure 1 visually shows the different steps, inputs and outputs. As shown in the figure, our approach feeds the probability bounds (ε and $1-\beta$) to the “Scenario Theory” [11] block. The scenario theory is used in control for the design of robust controllers to handle

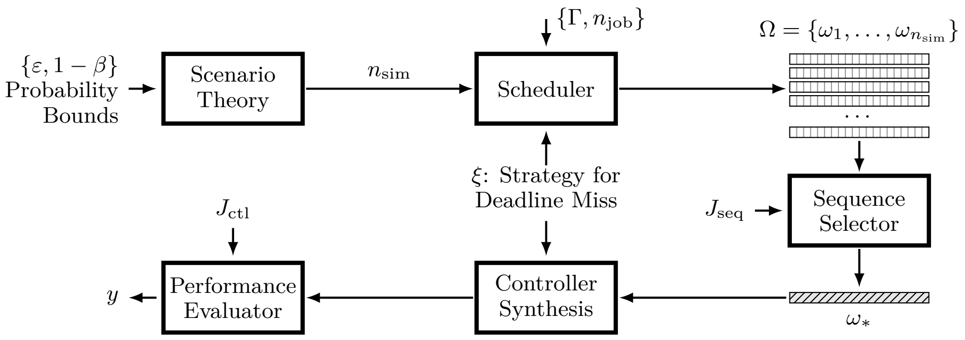

$\{\Gamma, n_{\text{job}}\}$
$\Omega = \{\omega_1, \ldots, \omega_{n_{\text{sim}}}\}$
$\{\varepsilon, 1-\beta\}$ Probability Bounds
Scenario Theory
n_{sim}
Scheduler
ξ: Strategy for Deadline Miss
J_{ctl}
J_{seq}
Sequence Selector
y
Performance Evaluator
Controller Synthesis
ω_*

Figure 1 Approach Overview.

uncertainty that is *a priori* unpredictable in the disturbance values and in the system model. In this paper we reinterpret the scenario results to enable a control synthesis strategy that uses deadline hits and misses information and provides (probabilistic) guarantees.

The scenario theory is a formal tool that determines how to analyze experimental data. In particular, we schedule our taskset extracting execution times from the corresponding probability distributions that are known in the Γ taskset. The theory provides us with information on how many experiments (scheduling simulations) we should execute in order for the probability that unforeseen circumstances are worse than the gathered data to be lower or equal to ε with confidence $1-\beta$. We denote the number produced by the scenario theory with n_{sim}. For each of the n_{sim} experiments, we randomly sample the probability distributions of the task execution times, to generate a set $\Omega = \{\omega_1, \ldots, \omega_{n_{\text{sim}}}\}$ scheduling sequences, in which the control task executes for n_{job} times, using strategy ξ to handle the deadline misses. Using our scheduler, we record sequences of deadline hits and misses.

We evaluate each of these sequences with a cost function J_{seq}, identifying the worst sequence ω_*, from the control perspective. From this sequence we extract the probability of deadline hits and misses for each of the n_{job} instances of the control task and the joint probability distribution for each sequence of hits and misses needed for the control design. The controller synthesis block uses the extracted information for the control strategy design. The generated controller is then evaluated when the taskset is executed and the controller is connected to the real plant, using a cost function J_{ctl}, which allows us to compare the performance of different deadline management strategies. We can then determine the best deadline management strategy and control period for the system under analysis.

As output of our approach we obtain y, the evaluation of each tested strategy ξ for the specific problem. As a by-product, we also obtain the set of sequences Ω. If we are not satisfied with our controller behavior, we can analyze the set of sequences to understand how to improve the control performance (i.e., for example optimize a certain task in the taskset).

Paper Organization

In the following, Section 3 discusses the model used for both the plant and the taskset, and Section 4 describes the behavior of the system using different deadline miss handling strategies. Section 5 presents the control design approach. In Section 6 we present the framework that we use to obtain probabilistic information about the scheduler behavior, and the scenario theory. In Section 7 we show our experimental setup and the evaluation criteria, and present our results. Section 8 discusses related work, and Section 9 concludes the paper.

3 System Model and Problem Definition

This section introduces the models used in the paper. Section 3.1 describes the model of the taskset executing on the hardware. Section 3.2 discusses the models of plant and control task. Finally, Section 3.3 introduces the three strategies used to handle deadline misses.

3.1 Taskset Model

In this paper, a real-time workload Γ is defined as the union of a (given) set of generic hard real-time periodic tasks, plus a real-time control task τ_d, which is the target of our design, i.e., $\Gamma = \Gamma' \bigcup \{\tau_d\}$. In this description, Γ' is a set of N_T periodic tasks, i.e., $\Gamma' = \{\tau_1, \tau_2 \ldots, \tau_{N_T}\}$ and τ_d is an additional periodic task that contains our controller operation. We assume that each task τ_i is independent from the others and released synchronously at a given starting instant. The tasks are scheduled using a fixed priority scheduling policy (e.g. Rate Monotonic) with preemption, and the indexing reflects their priority ordering, i.e. τ_i has higher priority than τ_j if $i < j$. In our design problem, τ_d is the task with the lowest priority.

Each task is characterized by a tuple of parameters, $\tau_i = (\mathcal{C}_i, f_i^{\mathcal{C}}, D_i, T_i)$. Here, $\mathcal{C}_i$ is a random variable that represents the task execution time, while $f_i^{\mathcal{C}}(c)$ is its probability density function, i.e. $\forall c \in \mathbb{N}$, $f_i^{\mathcal{C}}(c) = \mathbb{P}\{\mathcal{C}_i = c\}$; D_i and T_i are deterministic values, representing respectively the task deadline and period. In accordance with the literature on real-time applications for control systems, task periods are chosen among a limited set of possible values, typically related to physical requirements of the control task [32, 39].

For each task τ_i, we consider a discrete probability distribution $\mathcal{C}_i$ with N_i integer values, ranging between a Best Case Execution Time (BCET) $C_i^{\min}$ and a Worst Case Execution Time (WCET) $C_i^{\max}$. Furthermore, we consider tasks that behave well in most cases, i.e., tasks whose probability density functions are skewed towards lower values. In fact, while our approach can be applied to systems with generic probability density functions, we want to capture tasks which experience occasional faulty conditions. This choice is in agreement with most works that analyze execution time distributions for real-time tasks [53]. We will generally refer to the utilization of taskset Γ' as the *worst-case* utilization, i.e. $U_{\Gamma'} = \sum_{i=1}^{N_T}(C_i^{\max}/T_i)$.

We denote each periodic instance of $\tau_i \in \Gamma$ with the term job, and define it as $J_{i,k}$, with $k = 1, 2, \ldots$ representing the job index and i representing the task index. For every job $J_{i,k}$, $a_{i,k}$ denotes the activation instant, and $a_{i,k+1} - a_{i,k} = T_i$. Since we are considering synchronous release conditions, $\forall i$, $a_{i,0} = 0$ holds. In the following, $\mathcal{R}_{i,k}$ represents the random discrete variable that models the response time of $J_{i,k}$. The Worst Case Response Time (WCRT) of task τ_i is denoted as R_i^W and computed with standard techniques [33], by considering the condition where every task experiences its WCET. Similarly, the Best Case Response Time (BCRT) [43] is introduced as R_i^B and computed considering that every job executes with its BCET. Finally, in this work all tasks τ_i in Γ' are *schedulable*, i.e. $R_i^W \leq D_i$ for each τ_i. However, this hypothesis will not be required for τ_d. We will only assume that at least one job of τ_d respects its deadline, i.e. $R_d^B \leq D_d$.

3.2 Plant and Controller Model

The plant to be controlled by τ_d is described as a linear time invariant, multi-input multi-output system in continuous time. In line with standard assumptions, we assume the plant to be controllable and the state to be fully measurable. The plant dynamics is described as

$$\dot{\mathbf{x}}(t) = A_c\,\mathbf{x}(t) + B_c\,\mathbf{u_c}(t) + \mathbf{v_c}(t). \qquad (1)$$

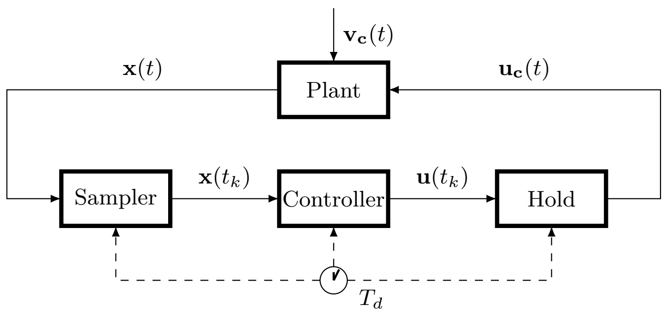


Figure 2 Plant and controller with time-triggered sampler and hold devices.

In Equation (1), every element in bold represents a vector, while A_c and B_c are the constant matrices that encode the dynamic evolution of the system. The term $\mathbf{x}(t)$ denotes the system state vector and $\dot{\mathbf{x}}(t)$ its time derivative. The term $\mathbf{u_c}(t)$ is the vector that contains the control signals. The vector $\mathbf{v_c}(t)$ represents the plant disturbance, modeled as white noise with known covariance matrix R_c. The goal of the control is to minimize a cost function, defined as the mean value of a quadratic function of the state vector and the control vector:

$$J_{\text{ctl}} = \mathbb{E}\left\{\int \mathbf{x}^{\mathrm{T}}(t)Q_{1c}\mathbf{x}(t) + \mathbf{u_c}^{\mathrm{T}}(t)Q_{2c}\mathbf{u_c}(t)\right\}. \tag{2}$$

Here, $\mathbb{E}$ indicates the expected value, while Q_{1c} and Q_{2c} are constant positive semidefinite matrices and design parameters of the controller. They represent the trade-off between regulating $\mathbf{x}(t)$ to zero and the cost of using the control signal $\mathbf{u_c}(t)$. This cost function is used both as a controller design objective and for performance evaluation of the control task.

The plant is connected to the controller via time-triggered sampler and hold devices as shown in Figure 2. The behavior of these devices can be modeled as a dedicated task that reads and writes data with zero execution time and highest priority. The plant state is sampled every T_d time units, implying $\mathbf{x}(t_k) = \mathbf{x}(kT_d)$. The control job $J_{d,k}$ is released at the same instant, i.e. $a_{d,k} = kT_d$, and the sensor data $\mathbf{x}(t_k)$ is immediately available to it. Based on the state measurement, the controller computes the feedback control action $\mathbf{u}(t_k)$.

As an hypothesis, our control task τ_d executes under the Logical Execution Time paradigm. Indeed, the job $J_{d,k}$ computes the control output using $\mathbf{x}(t_k)$ but makes it available to the actuator only at the first deadline instant after the termination of its execution. The control actuation is then held constant until the *next* update. This means that, if all jobs finish before their deadline, the following equation holds:

$$\mathbf{u_c}(t) = \mathbf{u}(t_k), \qquad a_{d,k} + D_d \leq t < a_{d,k+1} + D_d. \tag{3}$$

The execution time of the control task τ_d is given as a random variable with known probability density function, and is treated equivalently to any other task in Γ'. On the contrary, the deadline D_d and period T_d of the control task τ_d are part of the design. Being a LET task, we restrict our analysis to the implicit deadline case ($D_d = T_d$), although in principle the approach in the paper can be applied to other relative deadlines (and corresponding output times). We further assume that the execution time properties of the controller do not change with different periods and different controller parameters (since only the values of some parameter are modified but the operations done by the control task are the same).

In the paper, τ_d is not treated as a hard-deadline task. On the contrary, we actively look for those values of T_d such that the resulting task may miss some deadline with probability greater than zero, but still being able to guarantee a good control performance. This is to increase the system utilization, and consolidate workload on one single core. In Section 5, we present how to properly characterize the timing behavior of the controller and its synthesis.

▶ Remark 3.1. In this paper, we work under the assumption that τ_d is the task with the lowest priority. If other tasks with priority lower than τ_d do exist, the design proposed hereafter is still valid in principle, since those tasks cannot interfere with τ_d. However, if this is the case, the range of possible values of T_d should be tied with the schedulability guarantees for the lower priority tasks. We reserve to analyze this more general case as a future work.

3.3 Handling Deadline Misses

In classic control design, the control task behavior is assumed to be strictly periodic, or at least periodic with limited reponse time jitter [16]. To provide an implementation enforcing periodicity for a controller on a real-time platform, one usually selects a period for the control task that is greater or equal than the task's WCRT. This approach is safe but can be very pessimistic. In fact, WCRT conditions may be extremely rare. Selecting the period with the mentioned constraint could limit the achievable control performance, since longer sampling periods in general mean worse disturbance rejection and smaller stability margins [15].

The approach presented in this paper explores the possibility of designing a control task with periods smaller than its WCRT, i.e., $T_d < R_d^W$, thus greatly extending the design space. We remark that, with $T_d \geq R_d^W$ there are no deadline misses, and standard approaches for the control design can be used [29, 4, 55].

Choosing $T_d < R_d^W$ implies the risk that the control task will miss some deadlines. A deadline miss is a timing violation that can produce unbounded response times, due to self-pushing [47], and therefore it should be properly handled. In this work, we consider three different strategies to handle deadline misses, that have previously been explored in the control community [13]: **(i)** to *kill* the job that has not completed at the deadline, **(ii)** to let the current job continue but *skip the next* job(s), and **(iii)** to let the job continue, placing the next job(s) in a *queue of length one*. In more detail, these strategies behave as follows:

- Kill: A control job that is not able to terminate within its deadline is dropped at the deadline instant. When a job is killed, its (partial) computation is discarded and no output is produced. We assume that this dropping mechanism has negligible overhead and internal states of the controller are not altered by the partial computation.
- Skip-Next: A control job that is not able to terminate within its deadline is allowed to continue until completion. However, whenever the active job exceeds a deadline, the next instance of the control job is not activated (skipped). This is based on the idea that completing a job that has already started is preferred to starting a new one and incurring the risk that the computation runs longer than the deadline again.
- Queue(1): A control job that is not able to complete its execution within its deadline is allowed to continue its execution, while the following jobs are put in a queue that can contain a single element. Thus, at the activation of a new job, if there is already an active instance, the new job is enqueued, overwriting the currently existing job in the queue. Only the most recently arrived job is stored in the queue and is activated as soon as the current job completes.

An example of a schedule under the three strategies is presented in Figure 3, where the odd jobs are shown in dark gray, while the even jobs are shown in light gray. With the Kill strategy, the first job is killed at its deadline, before completing its execution. With the

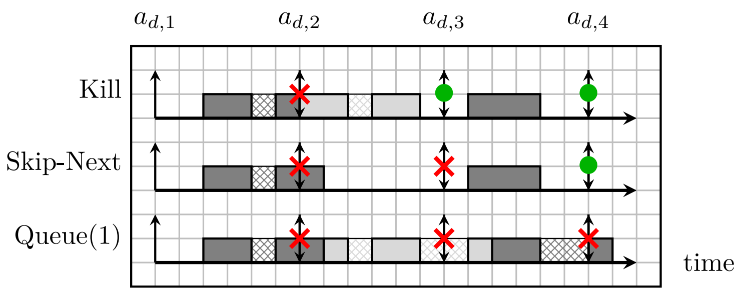


Figure 3 Schedule example using the three proposed strategies to handle deadline misses. Those jobs that missed a deadline (or are skipped) are marked with a red cross on their deadline, while a green dot identifies deadline hit events.

other two strategies, the job is given additional time in the second period, and is therefore able to complete, albeit running over time. With the Skip-Next strategy, the second job is not started, since there is an active control job that has not terminated its computation. With the Queue(1) strategy, the second job also runs over time, due to interference. In general, the Kill and Skip-Next strategies avoid self-interference conditions. This is not true for the Queue(1) strategy. However, Queue(1) may be seen as a particular case of finite buffer strategy [1], where the freshest job of the queue is always preferred, discarding the old one that has not yet started. This choice helps reducing the amount of self-interference and avoids unbounded response times. In practice, a job that is delayed more than one period by self-interference is skipped and the next one is put in the ready queue.

A sequence of consecutive control jobs may contain a certain number of jobs that are not actively contributing to the actuation $\mathbf{u}(t)$. This happens either with jobs that are terminated before completing their execution (killed) or with jobs not executed at all (skipped). For those jobs, a proper response time value may not be defined. Moreover, under Queue(1) strategy, it may happen that the output of a job which completes its execution after missing one or more deadlines is *overwritten* by the next job, if the latter completes before the same deadline. We therefore define the set of jobs that produce an output control that is actually provided to the physical plant, as the set of *valid* control jobs.

▶ **Definition 3.2** (Valid control job). *A* valid *control job ν is a job that successfully completes its execution and whose generated output is not overwritten before the next deadline instant.*

For each time interval $[0, t)$, we show that is possible to extract the ordered sequence of v valid jobs, defined as $S = \{\nu_1, \nu_2, ..., \nu_v\}$ (where the index does not count the passing of time) and the relation $v \leq \lceil t/T_d \rceil$ trivially holds. The sequence of valid jobs depends on the strategy used to handle deadline misses, and will be described in Section 4. Our control design should be robust not only to the possibility of missing deadlines, but also to the different pattern of delays that are produced depending on the strategy used to handle the miss event. In the following section, we discuss how this affects the control task behavior.

4 Controller Behavior with Deadline Misses

In theory, choosing a shorter period allows the discrete-time controller to achieve better control performance [5]. However, real-time constraints become harder to satisfy, due to the increased interference from higher priority tasks. Since we are targeting periods shorter than the WCRT, the probability of missing a deadline for the control task is greater than

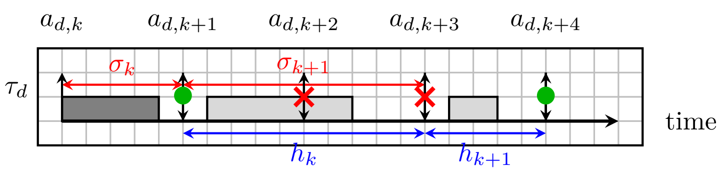


Figure 4 Example of delay and hold values for Skip-Next strategy.

zero. A controller designed with standard techniques and subject to overrun may still display some intrinsic robustness when experiencing occasional deadline misses [13, 41]. However, the controller performance depends strongly on the deadline miss handling strategy, and may become unacceptable when the probability of missing a deadline is too high. Here we analyze the timing properties of a control task subject to deadline misses. We show that only two parameters are needed to fully characterize the miss impact. We build the rules for computing them and extract the sequence of valid control jobs.

4.1 Defining Delay and Hold

Missed deadlines invalidate one of the basic hypothesis of control theory, which is the periodicity of the output pattern [41]. In this work, we exploit the knowledge of deadline misses directly in the control design step. For this purpose, we need to characterize how deadline misses affect the control performance. We fully describe the effect of deadline misses of LET-based controllers with two parameters, named respectively *delay* and *hold* interval.

▶ **Definition 4.1** (Delay σ_k). *The* delay σ_k *experienced by a control job* $J_{d,k}$ *is defined as the time interval between the activation instant of the job* $a_{d,k}$ *and the instant where its control output is made available to the actuator.*

In other words, the delay σ_k represents the time from sampling the plant state until updating the control signal. Using the above formulation, σ_k can only be properly defined for jobs that correctly complete their execution: if a job is killed or skipped, no delay information can be extracted, since its computation does not properly finish. We will refer to this condition as an *undefined* delay, represented with the symbol ∞. From a control perspective, the delay experienced by each (completed) job must be compensated accordingly by a predictor that computes the expected state at the output instant $(t_k + \sigma_k)$, using the knowledge of the current state $\mathbf{x}(t_k)$ and the control output(s) active in that time span.

▶ **Definition 4.2** (Hold interval h_k). *Given a control output computed by* $J_{d,k}$ *and available at the actuator for the first time at* $t_k + \sigma_k$*, the* hold interval h_k *is the time interval between* $t_k + \sigma_k$ *and the first instant where a new control output is made available.*

In other words, the hold interval h_k indicates the lifetime of the control signal computed by the k-th controller job and thus represents the time interval in which the computed control signal is held constant. Similarly to the delay, the definition of h_k is meaningful only for jobs that correctly complete their execution. If job $J_{d,k}$ is killed or skipped, the hold interval is *undefined* and will be represented with the symbol ∞. Moreover, if job $J_{d,k}$ correctly completes its execution, but its output is overwritten by the output of job $J_{d,k+1}$ before being used, we will assign a hold value $h_k = 0$. Figure 4 shows an example of delay and hold intervals for a sequence of four jobs using the Skip-Next strategy.

4.2 Computing Delay and Hold

Under ideal timing conditions – i.e., when all the deadlines of the control task are hit – the control signal produced in one period is always applied in the next period. The controller should thus compensate for a fixed delay $\sigma_k = T_d$. In this situation, the delay and hold have the same value and they are often not even defined as two different parameters. However, when considering deadline misses, σ_k and h_k may assume different values. Figure 4 shows one such example, where $\sigma_k = 1\,T_d$, $h_k = 2\,T_d$, $\sigma_{k+1} = 2\,T_d$, and $h_{k+1} = 1\,T_d$.

The potential differences between the delay and hold values have several consequences for the control design. First, a predictor designed for a one period delay may produce a value that is incorrect for longer delays. Second, a control signal calculated for a short hold interval could be too aggressive if applied in longer intervals. Lastly, the resulting delay–hold pattern may change across multiple control activations and depends heavily on the deadline miss handling strategy that has been chosen.

Computing the values of σ_k and h_k for each $J_{d,k}$ in a schedule is then crucial for the control design process. In fact, knowing in advance the values of delay and hold interval of each job, enables the design of an optimal *time-varying* controller, that for each control job selects how to compensate the particular combination of σ and h for the current and following jobs. This however would require a *clairvoyant* controller, that is not practically realizable. Here we extract the possible pairs $(\sigma_k,\, h_k)$ that may happen in a given scheduling sequence, and their associated probability, to design a fixed controller that behaves as close as possible to the ideal unrealizable one. We discuss the controller synthesis in Section 5.

Knowing σ_k and h_k for a given job $J_{d,k}$, it is also possible to determine whether the k-th control job is *valid*. This corollary follows from the definition of delay and hold interval:

▶ **Corollary 4.3.** *A job $J_{d,k}$ is* valid *if and only if it is possible to define both its delay σ_k and hold interval h_k (i.e., they are finite numbers) and if the hold interval is greater than zero.*

As a consequence, the pairs $(\sigma_k,\, h_k)$ can be leveraged to extract the set of ordered *valid* jobs, which are the ones effectively used for building the controller. We now discuss how to compute σ_k and h_k for each $J_{d,k}$ with the different miss-handling strategies. First of all, it is worth noting that the definition of σ_k is strictly related to the notion of response time of job $J_{d,k}$. In fact, the control output computed by $J_{d,k}$ is dispatched to the actuator at the first control activation (i.e. the closest incoming deadline) that follows the termination of $J_{d,k}$. The delay σ_k of $J_{d,k}$ can then be computed (for each strategy) as follows:

$$\sigma_k = \begin{cases} \lceil \mathcal{R}_{d,k}/T_d \rceil \, T_d & \text{if } J_{d,k} \text{ completes} \\ \infty & \text{otherwise.} \end{cases} \tag{4}$$

Trivially, the maximum value for σ_k is $\bar{\sigma} = \lceil \mathcal{R}_d^W / T_d \rceil T_d$. While extracting σ_k requires only the knowledge of $J_{d,k}$, in order to compute the value of the hold interval h_k it is necessary to know the behavior of the control jobs executing after $J_{d,k}$, until the release of a new control update. In practice, this means that only a finite number of sub-sequences needs to be checked for characterizing all possible combinations of $(\sigma_k,\, h_k)$. Below, the equations for computing h_k for each strategy are presented in detail.

4.2.1 Hold Interval with Kill Strategy

Using the *Kill* strategy, the control job either finishes within one period or it is killed at its deadline. An arbitrary sequence of deadline misses may happen between two jobs that complete successfully. Denoting with $\lambda_{k,\text{Kill}}$ the number of consecutive jobs that miss their deadline after $J_{d,k}$, the hold interval associated to $J_{d,k}$ is computed according to

$$h_k = \begin{cases} (\lambda_{k,\text{Kill}} + 1) \cdot T_d & \text{if } J_{d,k} \text{ completes} \\ \infty & \text{otherwise (if } J_{d,k} \text{ is killed).} \end{cases}$$

If $J_{d,k}$ has been killed, h_k is not defined. Note that if some weakly hard constraint is known for τ_d, the values of $\lambda_{k,\text{Kill}}$ to check may be upperbounded with the maximum possible number of consecutive deadline misses of that task.

4.2.2 Hold Interval with Skip-Next Strategy

Using the *Skip-Next* strategy, no new job may be activated while the active one is executing. Denoting with $\lambda_{k,\text{Skip-Next}}$ the number of skipped jobs that directly follows $J_{d,k}$, the hold interval of a job $J_{d,k}$ can then be computed as

$$h_k = \begin{cases} \sigma_{k+1+\lambda_{k,\text{Skip-Next}}} & \text{if } J_{d,k} \text{ completes} \\ \infty & \text{otherwise (} J_{d,k} \text{ is skipped).} \end{cases}$$

If $J_{d,k}$ is skipped, h_k is not defined. Intuitively, this means that for the Skip-Next strategy, the hold value of one completed job is equal to the delay of the subsequent job that completes (i.e., of the next valid job). In the example of Figure 4, job $J_{d,k}$ terminates correctly, while $J_{d,k+1}$ does not complete before its deadline. $J_{d,k+2}$ is then skipped. The hold value h_k is therefore equal to the delay σ_{k+1} which is $2T_d$ since the job has an overrun. The hold value h_{k+1} is equal to the delay of the next completed job $J_{d,k+3}$, i.e., $h_{k+1} = \sigma_{k+3} = T_d$. The values that $\lambda_{k,\text{Skip-Next}}$ may assume are upperbounded by $\lceil R_d^W / T_d \rceil - 1$.

4.2.3 Hold Interval with Queue(1) Strategy

Using the *Queue*(1) strategy, if a job misses its deadline, two scenarios may happen: it completes before the deadline of $J_{d,k+1}$, or it finishes later, then some of the subsequent jobs is skipped. In both those cases, however, the instant where the control output is published to the actuator falls exactly one period (T_d) after the activation of the next valid job. Denoting with $\lambda_{k,\text{Queue}(1)}$ the number of (eventually) skipped jobs that directly follows $J_{d,k}$, the hold value h_k for job $J_{d,k}$ is computed as follows:

$$h_k = \begin{cases} \sigma_{k+1} & \text{if } J_{d,k} \text{ hits its deadline} \\ \sigma_{k+1+\lambda_{k,\text{Queue}(1)}} - T_d & \text{if } J_{d,k} \text{ misses its deadline} \\ \infty & \text{otherwise (skipped).} \end{cases}$$

If $J_{d,k}$ is skipped because it is removed from the queue due to a subsequent activation, h_k is not defined. Note that if $J_{d,k}$ misses and $J_{d,k+1}$ hits its deadline – i.e., if both the k-th and the $k+1$-th control jobs complete before during the $k+1$-th period – then $\sigma_{k+1} - T_d = 0$, and the control signal produced by $J_{d,k}$ is never actuated. Finally, values of $\lambda_{k,\text{Queue}(1)}$ are upperbounded by $\lceil (R_d^W - T_d)/T_d \rceil - 1$.

5 Synthesis of Deadline-Miss-Aware Controllers

Standard digital control design assumes that samples are taken regularly and that there is a (most likely known and constant) delay from sampling to actuation [5]. When deadlines are missed, the actual hold and delay intervals will deviate from the assumed values, as explained in the previous section. This *control jitter* leads to degraded performance, and, in extreme

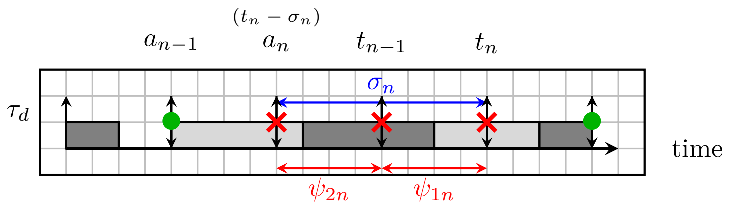


Figure 5 Example of ψ_{1n} and ψ_{2n}.

cases, even to instability of the control loop [16]. With some knowledge about the jitter, however, it is possible to synthesize a controller that partially compensates for the timing irregularities. We outline two variants of our Deadline-Miss-Aware Control designs below.

5.1 Clairvoyant Controller Synthesis

The controlled system evolution can be derived by sampling the plant only at the update instants of each valid job ν_n, i.e. at the time where the control output produced by ν_n is provided to the actuator. With a slight abuse of notation we will refer hereafter to the pair of delay and hold relative to ν_n as (σ_n, h_n), while its activation instant is a_n. The update instant of the control output produced by ν_n can then be defined as $t_n = a_n + \sigma_n$. Moreover, the relation $t_{n+1} = t_n + h_n$ trivially holds. For each valid control job ν_n in sequence $S = \{\nu_1, \nu_2, ..., \nu_v\}$, the state evolution can be calculated as

$$\mathbf{x}(t_{n+1}) = \mathbf{x}(t_n + h_n) = A(h_n)\mathbf{x}(t_n) + B(h_n)\mathbf{u}(t_n) + \mathbf{v}(t_n), \tag{5}$$

where $\mathbf{x}(t_n)$ is the state measurement sampled at time t_n, $\mathbf{u}(t_n)$ the control output released at time t_n, and $\mathbf{v}(t_n)$ a discrete-time model of the plant disturbance. The discrete matrices A and B are sampled from A_c and B_c of (1), respectively, with the step h_n. It is worth noting that different matrices $A(h_n)$ and $B(h_n)$ are created, depending on the possible values of h_n. In fact, a system described in this way behaves as a *switched-linear* system [48]. Computing the matrices can be done with standard procedures for sampled-data systems [5].

If the timing behavior of all jobs was completely known in advance, we would be able to design, by looking offline at the schedule, an optimal time-varying controller that minimizes the cost function (2). We call this a *clairvoyant* controller. The optimal control signal to be applied in the hold interval h_n is given by

$$\mathbf{u}(t_n) = -L_n\mathbf{x}(t_n), \tag{6}$$

where the sequence of feedback gain matrices $\{L_n\}$ are obtained as the solution to a time-varying Riccati equation involving the sequences $\{A(h_n)\}$, $\{B(h_n)\}$, and the sampled equivalents of the cost matrices Q_{1c} and Q_{2c}. The feedback matrices can be calculated off-line and stored in a table for on-line use.

The control law (6) cannot be implemented as it stands, though. The control action must be computed based on a state measurement that is σ_n time units old. Hence the controller must also predict the state from time $t_n - \sigma_n$ to t_n. Note however that in the time interval between $t_n - \sigma_n$ and t_n, the control actuation may not be constant, thus a slightly different modeling is needed. We will refer to the estimate of the state as $\hat{\mathbf{x}}$, which is computed as

$$\hat{\mathbf{x}}(t_n) = A(\sigma_n)\mathbf{x}(t_n - \sigma_n) + A(\psi_{1n})B(\psi_{2n})\mathbf{u}(t_{n-2}) + B(\psi_{1n})\mathbf{u}(t_{n-1}). \tag{7}$$

Here, ψ_{1n} represents the time interval in $[t_n - \sigma_n, t_n]$ when the control actuation of the previous valid job $\mathbf{u}(t_{n-1})$ is held constant, while ψ_{2n} is the (possible) interval where $\mathbf{u}(t_{n-2})$ is active. For the sake of clarity, an example is shown in Figure 5. An operative procedure for computing ψ_{1n} and ψ_{2n} is given as follows:

$$\psi_{1n} = a_n + \sigma_n - (a_{n-1} + \sigma_{n-1}), \qquad \psi_{2n} = a_{n-1} + \sigma_{n-1} - a_n. \tag{8}$$

5.2 Robust Controller Synthesis

The clairvoyant controller has two drawbacks. First of all, it relies on exact knowledge of the execution of the system, ahead of time. This is only possible in very special circumstances. The other drawback is that it is time varying, which is more complicated to implement and requires extra memory to store the time-varying feedback gain and prediction matrices. A more realistic approach is instead to design a fixed, *robust* controller, based on the statistical properties of the system.

Again starting from the sampled system description (5), we can instead solve a *stochastic* Riccati equation [38] based on the possible values of $A(h_n)$ and $B(h_n)$ and their relative frequency in the schedule during the execution of the system. The control law is then

$$\mathbf{u}(t_n) = -\bar{L}\,\mathbf{x}(t_n), \tag{9}$$

where $\bar{L}$ is a *fixed* gain matrix obtained from the solution to the stochastic Riccati equation

$$\begin{aligned}
\bar{X} &= \mathbb{E}\left\{ \begin{bmatrix} A(h_n)^{\mathrm{T}} \\ B(h_n)^{\mathrm{T}} \end{bmatrix}^{\mathrm{T}} \bar{S} \begin{bmatrix} A(h_n)^{\mathrm{T}} \\ B(h_n)^{\mathrm{T}} \end{bmatrix} + \begin{bmatrix} Q_1(h_n) & Q_{12}(h_n) \\ Q_{12}(h_n)^{\mathrm{T}} & Q_2(h_n) \end{bmatrix} \right\} \\
\bar{S} &= \bar{X}_{11} - \bar{L}^{\mathrm{T}} \bar{X}_{22} \bar{L} \\
\bar{L} &= \bar{X}_{22}^{-1} \bar{X}_{12}^{\mathrm{T}}.
\end{aligned}$$

This would be the optimal fixed-gain control law if the matrices $A(h_n)$ and $B(h_n)$ were random and independent from job to job. In reality, there is time dependence between the hold intervals due to the scheduling algorithm, and the control law is hence only sub-optimal.

The predictor (7) must also be modified to work with statistics rather than known-ahead values. The state can be predicted using expected value calculations as

$$\hat{\mathbf{x}}(t_n) = \mathbb{E}\left\{A(\sigma_n)\right\} \mathbf{x}(t_n - \sigma_n) + \mathbb{E}\left\{A(\psi_{1n})B(\psi_{2n})\right\} \mathbf{u}(t_{n-2}) + \mathbb{E}\left\{B(\psi_{1n})\right\} \mathbf{u}(t_{n-1}). \tag{10}$$

Again, the predictor will only be sub-optimal due to the time-dependence induced by the scheduling algorithm.

5.3 Controller Synthesis Example

The synthesis methods presented above are illustrated in a simple control example, which was used to evaluate the performance of a standard (non-deadline-miss-aware) controller under various overrun strategies in [13]. The plant to be controlled is an integrator process described by the parameters $A_c = 0$, $B_c = 1$, $Q_{1c} = 1$, $Q_{2c} = 0.1$ and $R_c = 1$. The plant is controlled by a control task with stochastic execution times, executing alone in a CPU. The execution time may assume value equal to 1 s with probability 0.8, or uniformly distributed in the interval $(1, 2]$ with combined probability 0.2. For periods ranging between 1 and 2, we compare the resulting performance under the Kill, Skip-Next, and Queue(1) strategies in Figure 6. Since J_{ctl} is defined as a cost, lower values in the graph mean better performance.

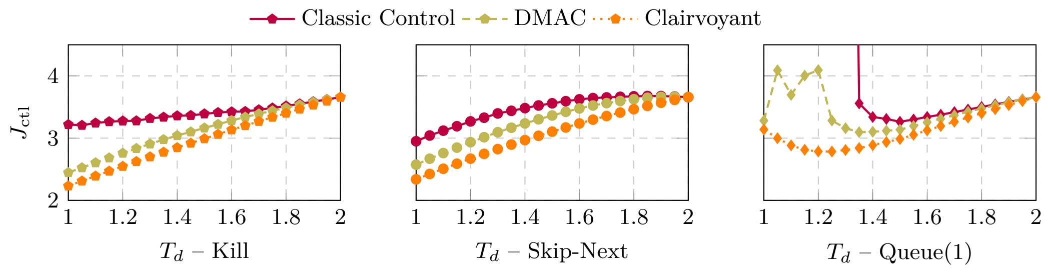


Figure 6 Control synthesis example: single task with deadline misses.

For each configuration, a standard controller (designed assuming no missed deadlines), a robust controller, and a clairvoyant controller are designed, and the performance of each controller, measured in terms of the cost function (2), is evaluated using JitterTime [14] in a simulation of 100,000 jobs. It can be noted that there is a strict ordering from the worst performance under standard control to the best performance under clairvoyant control, as expected. This means that designing control strategies that take into account deadline misses is beneficial in all cases. The DMAC design does not achieve the optimal cost that the clairvoyant design is able to achieve, but systematically beats classical control design due to its delay and hold compensation.

As the period is decreased from 2 to lower values, the Kill and Queue(1) strategies initially behave similarly, with decreasing cost. In fact, in the case of a miss followed by a deadline hit, the Kill and Queue(1) strategies have the same behavior (since the output of the late-completed job under Queue(1) is overwritten by the completion of the next one). Skip-Next initially has an increase in cost due to the waste of resources when a very small overrun leads to a whole period being skipped. For smaller task periods, Queue(1) suffers performance degradation and even instability ($J_{\mathrm{ctl}} \to \infty$) due to the lag introduced by the queuing. The Kill and Skip-Next strategies perform the best at $T_d = 1$, with very similar results for this example.

It should be noted that the results are problem dependent, and it is hard to judge whether Kill or Skip-Next works the best in general. In all examples, however, we have found that better performance can be achieved by shortening the period and allowing a few deadline misses. Some tests that include higher-priority tasks Γ' are presented later in Section 7.

6 Stochastic Analysis

Section 5 introduced a control design technique that exploits information about the probability of sequences of deadline hits and misses for the control job. Here, we provide a framework to robustly estimate these probabilities, and at the same time preserve a pessimistic bound that allows us to mitigate the effect of worst-case conditions. We formulate the estimation problem as a chance-constrained optimization problem [37], i.e., an optimization problem where we look for the probabilities of different sequences of hits and misses given the worst-case realization of the uncertainty inherently present in the taskset execution.

Analytical approaches extracting the probability of hits and misses for a schedule of jobs are either extremely pessimistic [17] or have a high computational complexity [51]. This limits the applicability of these techniques in non-trivial cases. Moreover, there are few works dealing with joint probabilities of consecutive jobs, like [49], but they still lack of scalability.

To handle the scalability issue, we adopt a simulation-based approach, backed up by the *scenario theory* [11], that *empirically* performs the uncertainty characterization, and provides *formal guarantees* on the robustness of the resulting estimation. The scenario theory allows us to exploit the fact that simulating the taskset execution (with statistical significance) is less computationally expensive than an analytical approach that incurs into the problem of combinatorial explosion of the different possible uncertainty realizations. In practice, this means that we: (i) sample the execution times from the probability distributions specified for each task, $f_i^{\mathcal{C}}(c)$, (ii) schedule the tasks, checking the resulting set of sequences Ω, and (iii) find the worst-case sequence ω_* based on the chosen cost function. The probabilities of sequences of hits and misses are then computed based on this sequence, and used in the design of the controller to be robust with respect to the sequence. We use the scenario theory to quantify, according to the number of extracted samples, the probability ε of not having extracted the *true* worst-case sequence and the confidence in the process $1 - \beta$. Scenario theory has for example found use in the management of energy storage[20].

6.1 Scenario Theory

The scenario theory has been developed in the field of robust control to provide robustness guarantees for convex optimization problems in presence of probabilistic uncertainty. In these problems, accounting for all the possible uncertainty realization might be achieved analytically, but is computationally too heavy or results in pessimistic bounds. The scenario theory proposes an empirical method in which samples are drawn from the possible realizations of uncertainty, finding a lower bound on the number of samples. It provides statistical guarantees on the value of the cost function with respect to the general case, provided that the sources of uncertainty are the same.

One of the advantages of this approach is that there is no need to enumerate the uncertainty sources, the only requirement being the possibility to draw representative samples. This eliminates the need to make assumptions on the correlation between the probability of deadline misses in subsequent jobs. If interference is happening between the jobs, this interference empirically appears when the system behavior is sampled. While there is no requirement on subsequent jobs interfering with one another, there is a requirement that different sequences are independent (i.e., each sequence represents an execution of the entire taskset of a given length, in the same or possibly different conditions). Taking the worst observed case in a set of experiments, the scenario theory allows us to estimate the probability that something worse than what is observed can happen during the execution of the system.

Specifically, for a sequence ω we define a cost function $J_{seq}(\omega)$, that determines when we consider a sequence worse than another (from the perspective of the controller execution). Denoting with $\mu_{\text{tot}}(\omega)$ the total number of job skips and deadline misses that the control task experienced in ω, and with $\mu_{\text{seq}}(\omega)$ the maximum number of consecutive deadline misses or skipped jobs in ω, we chose to use as a cost function the following expression:

$$J_{seq}(\omega) = \mu_{\text{tot}}(\omega)\,\mu_{\text{seq}}(\omega) \tag{11}$$

to determine the worst-case sequence of hits and misses. Given a set of sequences $\Omega = \{\omega_1, \dots \omega_{n_{\text{sim}}}\}$, we select $\omega_* = \arg\max_{\omega \in \Omega} J_{seq}(\omega)$. The choice of the cost function is anyhow not-univocal. For instance, other viable alternatives would be: (i) the number of subsequences of a given length with at least a given number of deadline misses, or (ii) the shortest subsequence with more than a given number of deadline misses.

6.2 Formal Guarantees

The scenario theory allows us to compute the number n_{sim} of simulations that we need to conduct to reach the required robustness ε and confidence $1-\beta$. The parameter ε is a bound on the probability of the obtained result being wrong, i.e., on the probability that another simulation would lead to a sequence with a higher cost function J_{seq} than ω_*. The parameter $1-\beta$ represents the confidence we have in this result, i.e., the probability of ε being an incorrect bound. It can also be interpreted as the probability that the drawn n_{seq} sequences are representative enough of the whole set of possible uncertainty realizations.

Equation (12) shows the relation between the number of experiments n_{sim}, ε and β [11]. Here, d is the number of optimization variables used for the selection. The cost function J_{seq} that we defined takes as argument only a sequence ω, hence $d = 1$.

$$\sum_{i=0}^{d-1} \binom{n_{\text{sim}}}{i} \varepsilon^i (1-\varepsilon)^{n_{\text{sim}}-i} \leq \beta. \tag{12}$$

Specifying β and ε univocally determines n_{sim}. If β and ε are sufficiently small, we can use the worst-case sequence for the design of the controller with high confidence.

6.3 Application and Threats to Validity

Similarly to any other empirical approach, the validity of the scenario theory depends on the representativeness of the sampling set. In our case, for example the validity of our results depends on the significance of the probabilistic execution time distributions for all the tasks.

Furthermore, the length of the simulations is a critical parameter. We simulate the system for a number n_{job} of executions of the control task. Clearly, we want to select n_{job} to cover an entire hyperperiod (to achieve complete analysis of the interferences between the tasks). In practice, we want to be able to detect cascaded effects that might happen due to the probabilistic nature of the execution times of the tasks. Some samplings could in fact make the utilization of instances of the taskset greater than one. For this reason simulations that include several hyperperiods should be performed. On top of that significancy with respect the controlled of the physical system is required (since the existence of the hyperperiod is not always guaranteed), hence the length of the simulated sequences should cover its dynamics.

7 Experimental Evaluation

This section presents and discusses the results obtained with our synthesis method. Experiments are obtained generating synthetic real-time workload. First, we generate the tasks Γ' and τ_d. We then use a simulator to draw execution time realizations from the determined probability distributions for the tasks and generate schedules and sequences of deadline hits and miss with different deadline-miss handling strategies, according to the scenario theory parameters. We select the worst-case sequence and use it for control design. Finally, we test the obtained controller on the physical plant, computing the control performance J_{ctl}. Section 7.1 describes our experimental setup, while Section 7.2 discusses our results.

7.1 Setup

To generate the taskset, and its execution time probability distributions, our experimental evaluation follows this procedure:

- Using the UUnifast algorithm [8], we generate an initial taskset Γ', composed of $N_T - 1$ tasks and having utilization $U_{\Gamma'}$. We order the tasks using Rate Monotonic priority. In the following, we show examples where $N_T \in \{5, 10, 20\}$, and $U_{\Gamma'} \in \{0.70, 0.80\}$. Tasks periods are chosen randomly from a bucket of values, ranging between 100 ms and 1000 ms, with steps of 10 ms. The execution times generated by the UUnifast algorithm are set as the WCETs of the tasks. All tasks in the generated task set must respect their (hard) deadlines using the WCET values.
- For each generated taskset, we build a control task τ_d. The control task is set to have the lowest priority in the set. We assume that the interval of interesting periods for the controller spans between 0.5 s and 2 s. These values are chosen according to the physical constraints that the plant imposes (e.g., the speed of the plant dynamics). We then select a random value for the WCET of τ_d, ensuring that the response time of its critical job (which corresponds to the WCRT of τ_d in case no deadline is missed) is between 2 s and 2.5 s. This choice guarantees that in the interval of interest, the control task has non-zero probability of missing at least one deadline.
- For each task in the taskset we randomly choose the BCET, such that the the BCRT of τ_d lies below the lower limit of 0.5 s of our interval of interest (coherently with our hypothesis that the controller period should be higher than its BCRT). Since the execution time probability of each task is skewed towards lower values, we expect that the probability distribution of response times will be skewed in the same direction too. We experimented with many values for the controller BCRT R_d^B. We found two representative intervals, that show different trends and behaviors and therefore selected for visualization the cases in which $R_d^B \in [0.15, 0.25]$ s and $R_d^B \in [0.4, 0.5]$ s.
- For each task, we choose N_e points uniformly spaced in the interval between the task BCET and its WCET as possible execution times. In our tests, we selected $N_e = 5$, but a higher number of points does not pose any scalability issue. The probabilities for each possible execution time have been assigned with the following heuristic: the lowest half (rounded up) of values have assigned a probability of that cumulatively sums up to 0.75, while the remaining sums up to $1 - 0.75 = 0.25$. This choice is based on the assumption that, realistically, the probabilities should be *skewed* towards the lower values.

For each period of interest T_d, we evaluate the control design as follows. We use a simulator, built in C++, to generate the sequences used for the scenario theory. We choose $\epsilon = 0.003$ and $\beta = 0.01$, obtaining a value for the number of simulations equal $n_{\text{sim}} = 1533$. A number $n_{\text{j}} = 500$ of control jobs has been chosen as representative temporal horizon for our system. After choosing a target period T_d for the controller in the interval $[0.5, 2]$ s, the simulator generates a vector of jobs activated in the time interval $n_{\text{job}}\, T_d$. Each job is characterized by its activation instant, priority, deadline, and an execution time (drawn from the probability distribution described above). The simulator computes the response time and the delay σ_k of each control job, by using the three deadline miss strategies – i.e., Kill, Skip-Next, Queue(1). For each sequence of control jobs the simulator computes a cost J_{seq}, weighting both the total number of deadline misses and the maximum number of consecutive ones as shown in Equation (11).

The worst-case sequence is the input of our control design script, expressed as a vector of delays. The Matlab control design script computes the hold interval using the rules presented in Section 4.1, and selects the set of valid control jobs from the sequence. The average probability of all combinations of delay and hold values are extracted from the sequence, and used to build the DMAC controller, as shown in Section 5.2.

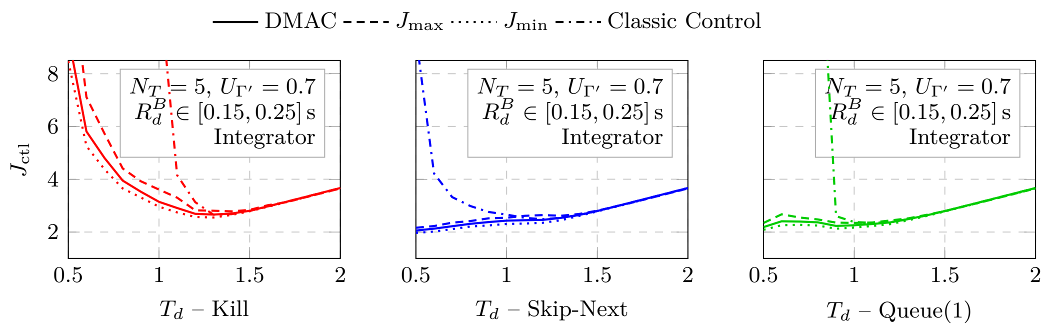


Figure 7 Comparison of DMAC with different deadline miss handling strategies (average, maximum, and minimum performance) with classical control.

Finally, the performance of the controlled system, where the control update is driven by the sequence of delays and holds from task schedule, is computed using *JitterTime* [14], a simulation-based tool for analysis of control systems performance inspired by *Jitterbug* [34] and *TrueTime* [15]. This new tool, built entirely in Matlab, is able to model transitions between different states with variable and conditional probabilities (overcoming some of the limitations of Jitterbug). JitterTime tests the various sequences of job schedules (with randomly generated execution times), producing the average and worst-case performance of the controller in terms of the cost function J_{ctl}.

7.2 Results

We conduct some experiments to study how the control performance, defined in Equation (2), changes when the control task period T_d assumes different values in the interval of interest. In the following, we show results obtained in different configurations. Specifically, for each deadline miss strategy we vary: (i) the number N_T of tasks in the taskset, (ii) the utilization of Γ', (iii) the control task best case response time R_d^B, and (iv) the dynamics of the physical plant to be controlled.

Figure 7 shows that DMAC design outperforms the classical control design (remember that J_{ctl} is defined as a cost, thus the lower, the better). We use the same plant as described in Section 5.3. From left to right, the figures show the cost function J_{ctl} obtained with the Kill, the Skip-Next, and the Queue(1) strategy when the period T_d varies. Solid lines represent the average performance of the DMAC controller (in the n_{sim} simulations). Dash dotted lines show the average performance of the classical control design method. When the period decreases, DMAC consistently and increasingly outperforms the classical design, obtaining a lower cost function. To ensure the robustness of the DMAC controller, we also plot the maximum and minimum cost obtained during the n_{sim} simulations (respectively using dashed and dotted lines). The area between the maximum and the minimum cost (which apparently includes the average value) is narrow, validating the robustness claim.

In Figure 8 we investigate the effect of varying the number of tasks N_T and the best case response time of the control task R_d^B. The number of tasks does not have a dramatic effect on any of the controllers, but the performance of a controller with the Kill strategy seem to benefit from an increase in the number of tasks. More generally, however, the Kill strategy is dominated by both the Skip-Next and the Queue(1) performance, that allow the design to reach shorter periods and to lower the cost function. The Kill strategy, that was achieving very good performance when tested with a single control task, does not handle additional load well. In fact, the failure of the Kill strategy is due to cascaded effects – killing subsequent

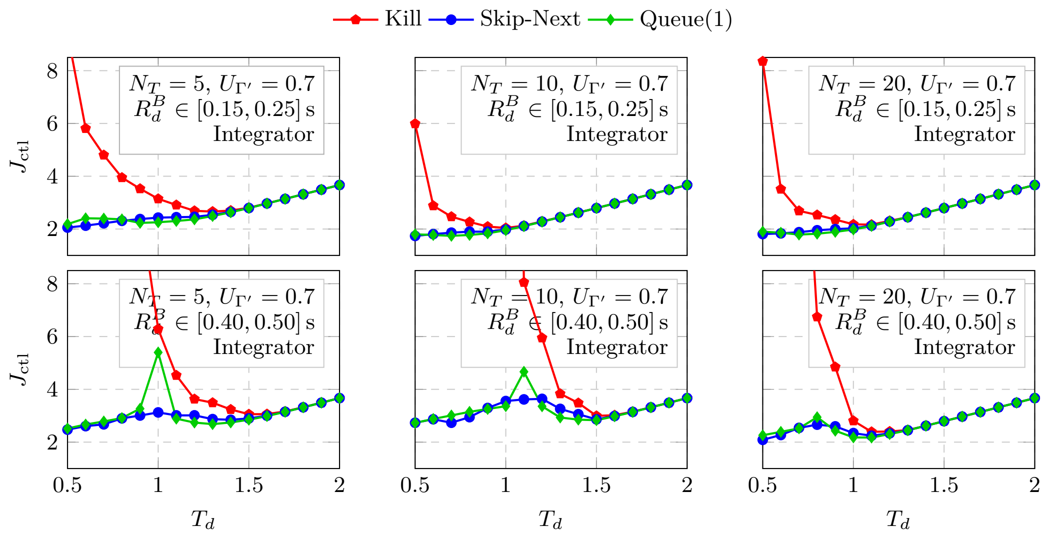


Figure 8 Control cost function for the Integrator plant with different miss-handling strategies, varying the number of tasks, and the best case response time.

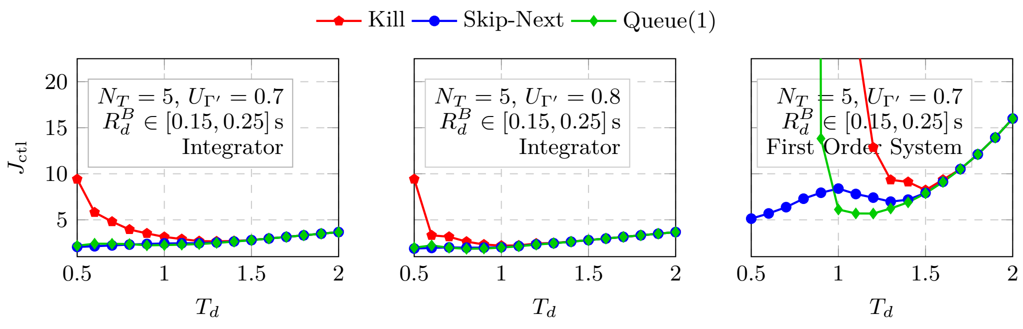


Figure 9 Control cost function varying utilization and plant dynamics.

jobs due to interference introduces long delays for the control signal, while allowing the job to terminate anyway (as the other two strategies do) leads to better performance. The Queue(1) and Skip-Next strategies behave similarly for both low values of R_d^B and high number of tasks. However, the Queue(1) shows some local performance drop in the cases of higher R_d^B. This happens when the average delay is just before the threshold of $2T_d$, i.e. when high delays but few skips occur. We then conclude that Skip-Next strategy is the most robust to both variations in the number of tasks N_T and in the best case response time R_d^B.

We conducted an extensive amount of tests, but due to space restrictions we can only show a limited number of additional results. Figure 9 shows the effect of increasing the utilization of Γ' and of changing the plant dynamics. An increase in utilization does not change the trends that can be observed (the Kill strategy being outperformed). However, changing the dynamics of the plant from a (marginally stable) Integrator to an (unstable) First-Order system has a dramatic effect on the cost function. The Queue(1) strategy behaves similarly (although better) with respect to the Kill strategy, not being able to handle short periods. The Skip-Next strategy, on the contrary, shows robust performance with respect to period shortening. The qualitative dependence on the plant dynamics will be explored further in future research.

8 Related Work

When designing a discrete time controller, it is fundamental to study if timing non-idealities may occur and how much they could harm the performance of the controlled system. The problem of analyzing the effects of late information on the system performance [31] has raised particular interest, especially in networked systems. In fact, transmission delays and packet drops may happen frequently when the transmission channels are heavily loaded or noisy. These timing effects are usually characterized as independent events with Gaussian distributions, or using worst case bounds [6]. By leveraging the knowledge of the timing non-idealities, many works proposed solutions for assuring the stability of the system [10, 35] and improving the control performance [44]. Sinopoli et al. [45] proposed an optimal control design for networked system leveraging the probability of packet losses. Similarly, the problem of designing an optimal control considering packet drops from the sensor is faced in [26] and [50]. In [46], the authors design an adaptive control that switches between normal, abort and skip mode depending on the delay (but which is always lower than than the period).

When dealing with controllers implemented in real-time systems, however, a different and more complex analysis is needed. Here, the input-output delay experienced by the control flow comes from the interference of higher priority tasks due to limited computational resources, that may even cause some job to miss their deadlines. Unforeseen delays may be caused, for example, by overload activations [27, 54], cache misses [22, 3] or complex interactions between scheduling and system state [9]. In recent works, systems that experience deadline misses are described using the so called weakly-hard model [7]. In this model, the possibility of missing a deadline is upper-bounded by a constraint (m, K), which gives the maximum number of deadlines m that may happen every K activation of a task. This model has proved being suitable for studying the effects of missed deadlines on the performance of control tasks and scheduling [42, 24]. A detailed modeling of the control performance considering different deadline miss handling strategies is presented in [41]. The effects of missed deadlines on system performance have been studied also using co-simulation [40]. Other works faced the co-design problem in overloaded systems by using complex mechanisms that take into account system stability and processor load [25, 56, 19].

In this paper, we study the effects of missed deadlines on the control performance by describing miss and hit events in a probabilistic fashion. The urge to bound WCET estimation and ensure timing correctness of systems led to the development of many probabilistic modeling techniques with remarkable success when applied to real systems [12, 52, 21, 30].

In our paper we assume that the execution times of each job are assumed as independent [36, 2], but we overcome the limitation of classical approach, not requiring that response times are modeled as independent variables. Exact methods for computing probabilistic response times of jobs exist [23], but their major downside is that they do not scale well and are applicable only to limited task sets and short hyperperiods. Other approaches face the problem by extracting probabilistic bounds with various approximation techniques [17, 51, 18]. The particular case of extracting joint probabilities of successive jobs is however less studied, and has been found e.g. in [49]. Again the work in [17] develops a bound for l-consecutive deadline misses, but it is still insufficient for our purposes. The path chosen for our work leverages the *scenario theory* approach [11] for performing a robust estimate of the hit and miss probabilities, by simulating multiple possible schedules.

9 Conclusion

This paper presented DMAC, a novel control design technique for building a fixed *Deadline-Miss-Aware Controller*. It also contributes with a methodology to evaluate control design strategies in the presence of deadline misses and possible overruns. Our controller leverages the probability of possible sequences of missed deadlines to compensate for the introduced delays. Our experimental results show that our design obtains better performance, while (safely) exploring period ranges that are usually avoided in state-of-the-art approaches. Moreover, we discussed how the control performance changes with respect to different deadline miss strategies and different taskset parameters. This paper highlights how, choosing the deadline miss handling strategy is one of the most critical parameters in the control design.

References

1. Leonie Ahrendts, Sophie Quinton, and Rolf Ernst. Finite ready queues as a mean for overload reduction in weakly-hard real-time systems. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 88–97. ACM, 2017.
2. Sebastian Altmeyer, Liliana Cucu-Grosjean, and Robert I Davis. Static probabilistic timing analysis for real-time systems using random replacement caches. *Real-Time Systems*, 51(1):77–123, 2015.
3. Sebastian Altmeyer and Robert I Davis. On the correctness, optimality and precision of static probabilistic timing analysis. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 26. European Design and Automation Association, 2014.
4. Amir Aminifar, Soheil Samii, Petru Eles, Zebo Peng, and Anton Cervin. Designing high-quality embedded control systems with guaranteed stability. In *2012 IEEE 33rd Real-Time Systems Symposium*, pages 283–292. IEEE, 2012.
5. Karl J Åström and Björn Wittenmark. *Computer-controlled systems: theory and design*. Courier Corporation, 2013.
6. Philip Axer, Maurice Sebastian, and Rolf Ernst. Probabilistic response time bound for CAN messages with arbitrary deadlines. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pages 1114–1117. IEEE, 2012.
7. Guillem Bernat, Alan Burns, and Albert Liamosi. Weakly hard real-time systems. *IEEE transactions on Computers*, 50(4):308–321, 2001.
8. Enrico Bini and Giorgio C Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.
9. Alessandro Biondi, Marco Di Natale, Giorgio C Buttazzo, and Paolo Pazzaglia. Selecting the transition speeds of engine control tasks to optimize the performance. *ACM Transactions on Cyber-Physical Systems*, 2(1):1, 2018.
10. Rainer Blind and Frank Allgöwer. Towards networked control systems with guaranteed stability: Using weakly hard real-time constraints to model the loss process. In *Decision and Control (CDC), 2015 IEEE 54th Annual Conference on*, pages 7510–7515. IEEE, 2015.
11. Giuseppe C Calafiore and Marco C Campi. The scenario approach to robust control design. *IEEE Transactions on Automatic Control*, 51(5):742–753, 2006.
12. Francisco J Cazorla, Eduardo Quiñones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, Emery Berger, Jaume Abella, Franck Wartel, Michael Houston, et al. Proartis: Probabilistically analyzable real-time systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s):94, 2013.
13. Anton Cervin. Analysis of overrun strategies in periodic control tasks. In *Proc. 16th IFAC World Congress, Prague, Czech Republic*, page 137. Citeseer, 2005.
14. Anton Cervin. JitterTime 1.0 reference manual. Technical report, Department of Automatic Control, Lund University, 2019. Technical Report TFRT-7658.

15 Anton Cervin, Dan Henriksson, Bo Lincoln, Johan Eker, and K-E Arzen. How does control timing affect performance? Analysis and simulation of timing using Jitterbug and TrueTime. *IEEE control systems*, 23(3):16–30, 2003.

16 Anton Cervin, Bo Lincoln, Johan Eker, Karl-Erik Arzén, and Giorgio Buttazzo. The jitter margin and its application in the design of real-time control systems. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications*, pages 1–9. Gothenburg, Sweden, 2004.

17 Kuan-Hsun Chen and Jian-Jia Chen. Probabilistic schedulability tests for uniprocessor fixed-priority scheduling under soft errors. In *Industrial Embedded Systems (SIES), 2017 12th IEEE International Symposium on*, pages 1–8. IEEE, 2017.

18 Kuan-Hsun Chen, Georg Von Der Brüggen, and Jian-Jia Chen. Analysis of deadline miss rates for uniprocessor fixed-priority scheduling. In *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 168–178. IEEE, 2018.

19 Hoon Sung Chwa, Kang G Shin, and Jinkyu Lee. Closing the gap between stability and schedulability: a new task model for Cyber-Physical Systems. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 327–337. IEEE, 2018.

20 G. Darivianakis, A. Eichler, R. S. Smith, and J. Lygeros. A Data-Driven Stochastic Optimization Approach to the Seasonal Storage Energy Management. *IEEE Control Systems Letters*, 1(2):394–399, 2017.

21 Robert Davis, Tullio Vardanega, Franck Wartel, Liliana Cucu-Grosjean, et al. PROXIMA: a probabilistic approach to the timing behaviour of mixed-criticality systems. *Ada User Journal*, 2:118–122, 2014.

22 Robert I Davis, Luca Santinelli, Sebastian Altmeyer, Claire Maiza, and Liliana Cucu-Grosjean. Analysis of probabilistic cache related pre-emption delays. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 168–179. IEEE, 2013.

23 José Luis Díaz, Daniel F García, Kanghee Kim, Chang-Gun Lee, L Lo Bello, José María López, Sang Lyul Min, and Orazio Mirabella. Stochastic analysis of periodic real-time systems. In *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, pages 289–300. IEEE, 2002.

24 Goran Frehse, Arne Hamann, Sophie Quinton, and Matthias Woehrle. Formal Analysis of Timing Effects on Closed-Loop Properties of Control Software. In *RTSS*, pages 53–62, 2014.

25 Mongi Ben Gaid, Daniel Simon, and Olivier Sename. A design methodology for weakly-hard real-time control. *IFAC Proceedings Volumes*, 41(2):10258–10264, 2008.

26 Vijay Gupta, Babak Hassibi, and Richard M Murray. Optimal LQG control across packet-dropping links. *Systems & Control Letters*, 56(6):439–446, 2007.

27 Zain AH Hammadeh, Sophie Quinton, and Rolf Ernst. Extending typical worst-case analysis using response-time dependencies to bound deadline misses. In *Embedded Software (EMSOFT), 2014 International Conference on*, pages 1–10. IEEE, 2014.

28 Thomas A Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Giotto: A time-triggered language for embedded programming. In *International Workshop on Embedded Software*, pages 166–184. Springer, 2001.

29 Byung Kook Kim. Task scheduling with feedback latency for real-time control systems. In *Real-Time Computing Systems and Applications, 1998. Proceedings. Fifth International Conference on*, pages 37–41. IEEE, 1998.

30 Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J Cazorla. A cache design for probabilistically analysable real-time systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 513–518. EDA Consortium, 2013.

31 Antzela Kosta, Nikolaos Pappas, Anthony Ephremides, and Vangelis Angelakis. Age and value of information: Non-linear age case. In *Information Theory (ISIT), 2017 IEEE International Symposium on*, pages 326–330. IEEE, 2017.

32 Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

33 John P Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 201–209. IEEE, 1990.

34 Bo Lincoln and Anton Cervin. Jitterbug: A tool for analysis of real-time control performance. In *Proceedings of the 41st IEEE Conference on Decision and Control, 2002.*, volume 2, pages 1319–1324. IEEE, 2002.

35 Steffen Linsenmayer and Frank Allgower. Stabilization of networked control systems with weakly hard real-time dropout description. In *Decision and Control (CDC), 2017 IEEE 56th Annual Conference on*, pages 4765–4770. IEEE, 2017.

36 Rui Liu, Alex F Mills, and James H Anderson. Independence thresholds: Balancing tractability and practicality in soft real-time stochastic analysis. In *Real-Time Systems Symposium (RTSS), 2014 IEEE*, pages 314–323. IEEE, 2014.

37 Bruce L. Miller and Harvey M. Wagner. Chance Constrained Programming with Joint Constraints. *Oper. Res.*, 13(6), 1965.

38 Johan Nilsson, Bo Bernhardsson, and Bjorn Wittenmark. Stochastic analysis and control of real-time systems with random time delays. *Automatica*, 34(1), 1998.

39 Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard, and Pierre Siron. The ROSACE case study: From Simulink specification to multi/many-core execution. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 309–318, 2014.

40 Paolo Pazzaglia, Marco Di Natale, Giorgio Buttazzo, and Matteo Secchiari. A framework for the co-simulation of engine controls and task scheduling. In *International Conference on Software Engineering and Formal Methods*, pages 438–452. Springer, 2017.

41 Paolo Pazzaglia, Luigi Pannocchi, Alessandro Biondi, and Marco Di Natale. Beyond the Weakly Hard Model: Measuring the Performance Cost of Deadline Misses. In *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

42 Parameswaran Ramanathan. Overload management in real-time control applications using (m, k)-firm guarantee. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):549–559, 1999.

43 Ola Redell and Martin Sanfridson. Exact best-case response time analysis of fixed priority scheduled tasks. In *Real-Time Systems, 2002. Proceedings. 14th Euromicro Conference on*, pages 165–172. IEEE, 2002.

44 Luca Schenato, Bruno Sinopoli, Massimo Franceschetti, Kameshwar Poolla, and S Shankar Sastry. Foundations of control and estimation over lossy networks. *Proceedings of the IEEE*, 95(1):163–187, 2007.

45 Bruno Sinopoli, Luca Schenato, Massimo Franceschetti, Kameshwar Poolla, and Shankar Sastry. An LQG optimal linear controller for control systems with packet losses. In *Decision and Control, 2005 and 2005 European Control Conference. CDC-ECC'05. 44th IEEE Conference on*, pages 458–463. IEEE, 2005.

46 Damoon Soudbakhsh, Linh Thi Xuan Phan, Anuradha M Annaswamy, and Oleg Sokolsky. Co-design of arbitrated network control systems with overrun strategies. *IEEE Transactions on Control of Network Systems*, 5(1):128–141, 2018.

47 Youcheng Sun and Marco Di Natale. Weakly Hard Schedulability Analysis for Fixed Priority Scheduling of Periodic Real-Time Tasks. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):171, 2017.

48 Zhendong Sun. *Switched linear systems: control and design.* Springer Science & Business Media, 2006.

49 Bogdan Tanasa, Unmesh D Bordoloi, Petru Eles, and Zebo Peng. Probabilistic response time and joint analysis of periodic tasks. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 235–246. IEEE, 2015.

50 Eelco P van Horssen, AR Baghban Behrouzian, Dip Goswami, Duarte Antunes, Twan Basten, and WPMH Heemels. Performance analysis and controller improvement for linear systems with (m, k)-firm data losses. In *2016 European Control Conference (ECC)*, pages 2571–2577. IEEE, 2016.

51 Georg von der Brüggen, Nico Piatkowski, Kuan-Hsun Chen, Jian-Jia Chen, and Katharina Morik. Efficiently Approximating the Probability of Deadline Misses in Real-Time Systems. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 106 of *ECRTS*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

52 Franck Wartel, Leonidas Kosmidis, Code Lo, Benoit Triquet, Eduardo Quinones, Jaume Abella, Adriana Gogonel, Andrea Baldovin, Enrico Mezzetti, Liliana Cucu, et al. Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study. In *Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on*, pages 241–248. IEEE, 2013.

53 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

54 Wenbo Xu, Zain AH Hammadeh, Alexander Kroller, Rolf Ernst, and Sophie Quinton. Improved deadline miss models for real-time systems using typical worst-case analysis. In *2015 27th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 247–256. IEEE, 2015.

55 Yang Xu, Karl-Erik Årzén, Anton Cervin, Enrico Bini, and Bogdan Tanasa. Exploiting job response-time information in the co-design of real-time control systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2015 IEEE 21st International Conference on*, pages 247–256. IEEE, 2015.

56 Tatsuya Yoshimoto and Toshimitsu Ushio. Optimal arbitration of control tasks by job skipping in cyber-physical systems. In *Proceedings of the 2011 IEEE/ACM Second International Conference on Cyber-Physical Systems*, pages 55–64. IEEE Computer Society, 2011.

Control-Flow Integrity for Real-Time Embedded Systems

Robert J. Walls
Worcester Polytechnic Institute, Worcester, Massachusetts, USA
rjwalls@wpi.edu

Nicholas F. Brown
Worcester Polytechnic Institute, Worcester, Massachusetts, USA
nfbrown@wpi.edu

Thomas Le Baron
Worcester Polytechnic Institute, Worcester, Massachusetts, USA
tlebaron@wpi.edu

Craig A. Shue
Worcester Polytechnic Institute, Worcester, Massachusetts, USA
cshue@cs.wpi.edu

Hamed Okhravi
MIT Lincoln Laboratory, Lexington, Massachusetts, USA
hamed.okhravi@ll.mit.edu

Bryan C. Ward
MIT Lincoln Laboratory, Lexington, Massachusetts, USA
bryan.ward@ll.mit.edu

Abstract

Attacks on real-time embedded systems can endanger lives and critical infrastructure. Despite this, techniques for securing embedded systems software have not been widely studied. Many existing security techniques for general-purpose computers rely on assumptions that do not hold in the embedded case. This paper focuses on one such technique, control-flow integrity (CFI), that has been vetted as an effective countermeasure against control-flow hijacking attacks on general-purpose computing systems. Without the process isolation and fine-grained memory protections provided by a general-purpose computer with a rich operating system, CFI cannot provide any security guarantees. This work proposes RECFISH, a system for providing CFI guarantees on ARM Cortex-R devices running minimal real-time operating systems. We provide techniques for protecting runtime structures, isolating processes, and instrumenting compiled ARM binaries with CFI protection. We empirically evaluate RECFISH and its performance implications for real-time systems. Our results suggest RECFISH can be directly applied to binaries without compromising real-time performance; in a test of over six million realistic task systems running FreeRTOS, 85% were still schedulable after adding RECFISH.

2012 ACM Subject Classification Security and privacy → Embedded systems security

Keywords and phrases Control-flow integrity

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.2

1 Introduction

Real-time and embedded systems (RTES) are predominantly developed in C because it offers high performance, low-level hardware control, and is often the only language supported by the manufacturer-provided toolchain for the target device. However, C also brings a host of potential memory errors, or vulnerabilities, that are both easy for developers to make, and easy for attackers to exploit. For example, memory-corruption vulnerabilities (e.g., buffer overflows) allow an attacker to overwrite portions of memory with attacker-provided values. Such vulnerabilities can be leveraged to hijack the control flow of a program by overwriting code pointers (e.g., function pointers or return addresses). Such attacks, commonly called *control-flow hijacking*, manipulate the execution of a program by redirecting control-flow transfers to either attacker-supplied code [33] or useful code sequences already in the program (e.g., return-oriented programming [ROP] [37]).

Several classes of defenses have been proposed for general-purpose systems to address control-flow hijacking. These include control-flow integrity (CFI) [5], which prevents such attacks by enforcing a precomputed control-flow graph (CFG) to runtime (indirect) control transfers in an application. Various other randomization-based [23, 26, 7, 24] and enforcement-based defenses [30, 31, 25, 32] have also been proposed in the literature.

However, there are a number of unique challenges that make existing implementations of these defenses ill-suited to RTES. First, embedded hardware is less capable and often lacks important hardware features that existing software defenses leverage. For example, the ARM Cortex-R architecture that we target in this work does not have a memory management unit (MMU) and, consequently, it does not support the abstraction of virtual memory nor does it provide isolation between kernel and application code. Second, in order to ensure the temporal correctness of the system, overheads associated with security defenses must be analyzed and factored into schedulability analyses. Third, embedded systems rely on toolchains tailored to each board and architecture, including custom versions of compilers (e.g., GCC) and proprietary IDEs (e.g., CodeComposerStudio). It is time-consuming (or impossible) to modify each of these toolchains to support new defenses.

Given these challenges, the security posture of many RTES lags behind that of general-purpose systems, despite being deployed in safety- or mission-critical applications. Given the proliferation of cyber-physical systems and Internet-of-things (IoT) devices, such systems are becoming ubiquitous in our society. Furthermore, such devices are increasingly Internet-connected, and therefore easily targeted by remote attackers. We must therefore develop security defenses for RTES that address the aforementioned challenges.

Towards that end, in this paper, we propose, implement, and evaluate a new defense for protecting RTES from control-flow hijacking attacks. Our defense, called Real-Time Embedded CFI for Secure Hardware (RECFISH), is inspired by past work on control-flow integrity but distinguishes itself from existing efforts in three key ways. First, RECFISH addresses the problem of custom toolchains by retrofitting binaries. This allows the developer to use existing toolchains without modification and even apply RECFISH protections to binaries without access to their source code. Second, we develop a new memory-isolation approach for ARM systems that does not rely on virtual memory. RECFISH provides the isolation between application and OS code needed to support secure context switching and enforce control-flow integrity. In particular, we modify a popular real-time operating system, FreeRTOS, to include RECFISH protections. Third, we provide a rigorous analysis of RECFISH's impact on real-time schedulablility and show that RECFISH can be applied to most systems without violating real-time requirements.

We evaluate the security and performance overhead of RECFISH using four broad classes of experiments. First, we perform the Basic Exploitation Test (BET) proposed by Carlini et al. [10] and demonstrate how RECFISH prevents various types of corruption used for control-hijacking, and how it secures the necessary CFI state from malicious modifications. Second, to evaluate the performance overhead, we run the CoreMark and BEEBS embedded-system benchmarks. Third, in order to better understand the sources of overhead, we conduct a series of microbenchmarks to quantify the CPU cycles necessary for each CFI operation.

Finally, based on the microbenchmark results, we empirically measure the effect on real-time *schedulability* [36], or the ability to analytically guarantee all deadlines will be satisfied, a fundamental metric in work on RTES. To our knowledge, we are the first to analyze the effect of a memory-corruption defense on analytical schedulability – this analysis provides a significant distinction from previous work in the embedded space (e.g., EPOXY [12]). Our contributions are summarized as follows:

- **Binary instrumentation for ARM:** We develop a CFI scheme, RECFISH, that protects both ARM-based bare-metal applications and those that run on FreeRTOS.
- **Protection mechanisms for CFI data structures:** We protect the instrumentation required for CFI as well as the shadow stack on low-resource ARM-based systems that lack native capabilities for such protections.
- **Process isolation without virtual memory:** We devise a low-overhead method for isolating critical parts of a process on ARM systems where all processes run in the same address space.
- **A binary-patching framework for ARM:** We create a binary-patching framework that rewrites precompiled ARM binaries to add CFI protection.
- **Evaluation:** We conduct both a security evaluation of RECFISH using BET and a performance evaluation using benchmarks, microbenchmarks, and schedulability.

2 Background and Related Work

2.1 Control Flow Integrity

CFI-based defenses check, at runtime, if program execution follows a legal control flow. Broadly, CFI schemes modify the target binary in three ways. First, at each indirect branch target, they insert a label to encode legal control-flow transfers. Second, at each indirect branch instruction, they insert instrumentation to verify the target has the expected label. Third, at each function return, they insert instrumentation to ensure control returns to the calling function.

Legal control flow is defined by the program's control-flow graph (CFG). Typically computed at compile-time, the CFG is a directed graph where the nodes represent *basic blocks* – i.e. sequences of program instructions ending in a branch – and the edges represent legal control-flow transfers between basic blocks. There are broadly two classes of branches: *direct branches* statically specify the target, while *indirect branches* depend on a register or memory value to specify the target at runtime. The latter are the target of control-flow hijacking attacks [37] and the focus of CFI. Note, checks are not needed for direct jumps when the code section of memory is read-only as the attacker cannot modify the target.

CFI implementations vary primarily in the choice of labeling scheme and the approach to protecting function returns. For performance reasons, some CFI approaches ignore function returns and only protect the other indirect branches. Other CFI-based defenses – including the original implementation by Abadi et al. [5] and the system proposed in this paper – rely on a runtime data structure, called a *shadow stack*, to securely store return addresses. This structure increases the precision of CFI and, by extension, the security of the instrumented program [17]; the tradeoff is higher overhead. See the survey by Burow et al. for a more comprehensive treatment of prior work on control-flow integrity [9].

Compared to earlier control-flow defenses (e.g., StackGuard [14], RAD [11], and DISE [13]), CFI implementations often consider a stronger threat model and provide stronger security guarantees. Specifically, CFI-based defenses must ensure that control-flow integrity is enforced even against adversaries that have full control of the data memory. In contrast, these earlier defenses do not protect all code pointers (only return address) and the shadow stack is either left unprotected from attackers with the ability to arbitrarily write to memory or the defense adds significant overhead by interposing on all (or a large subset of) memory writes.

2.2 Real-Time Embedded Systems

To facilitate writing real-time software, embedded-system designers often use a *real-time operating system* (*RTOS*). In a real-time OS, *tasks* are the rough equivalent of a process in a general purpose system. A *scheduler* is used to switch between executions of each task to meet pre-defined timing constraints.

RTOSes vary greatly in their complexity. On more powerful hardware, RTES can leverage versions of Linux compiled with `SCHED_DEADLINE` or `SCHED_RT`, which replace Linux's default scheduler with a real-time scheduler. On processors designed for embedded use – like those targeted for this work – the hardware typically does not meet the minimum requirements for Linux. For reference, in 2014, a minimally configured Linux kernel required at least 8 MB of program flash and 1.6 MB of RAM [40], whereas the test device for this work has only 1.25 MB of flash and 192 KB of RAM. The alternative to real-time Linux is using an embedded RTOS such as FreeRTOS or μC/OS, which are designed to run on devices with storage space and memory on the scale of kilobytes, rather than megabytes or gigabytes. One of the most common RTOSes is FreeRTOS. Designed to be as small as possible, this free and open source RTOS fits in as little as 5 KB of program flash and under 1 KB RAM, depending on the features used [4]. FreeRTOS is highly portable, with ports for most major architectures. FreeRTOS, while minimal in nature, provides a few rich features such as mutexes, semaphores, shared queues, and software timers.

2.3 Real-time Security

There has been some prior work on providing increased security to real-time systems. However, most of this work has focused on different attack classes or adopt weaker threat models than considered here. For example, Hasan et al. [22] considered how to schedule security monitoring into real-time scheduling while respecting legacy real-time constraints. Others have considered information-leakage attacks via cache-based and other side channels [29, 35], and how schedule randomization can be applied to defend against such threats [39]. In this work, we consider a much stronger, more pernicious threat model, that of memory corruption and control-flow hijacking.

EPOXY [12] targets the same class of attacks as RECFISH, but the underlying approach is significantly different. First, EPOXY does not guarantee control flow integrity, i.e., EPOXY does not check the target of indirect branches. Second, EPOXY is compiler-based whereas RECFISH retrofits existing binaries. It is unlikely that EPOXY could be re-engineered to work on existing binaries, e.g., EPOXY's code diversification presents significant challenges if implemented outside of a compiler. Third, EPOXY only targets bare-metal applications whereas RECFISH is implemented for both bare-metal and FreeRTOS. As we explain later sections, context switching introduces additional security challenges, which EPOXY does not address; notably, EPOXY does nothing to protect the stack in a multi-task environment.

2.4 ARM Architecture

Our work focuses on ARM's Cortex-R architecture for high performance real-time systems. Most Cortex-R processors are single core and they have special interrupt controllers and caching mechanisms to support the low latency required by real-time systems. Unlike x86-based hardware, ARM Cortex-R does not support virtual memory. Consequently, all realtime tasks share the address space. The lack of a memory-management unit and high-quality entropy sources [19], coupled with a small address space, mean it is especially challenging to implement randomization-based defenses (e.g., ASLR). Further, these challenges also complicate the implementation of secure runtime data structures (e.g., the shadow stack).

Another important complication is that Cortex-R chips operate on several different instruction sets, such as the ARM and Thumb instruction sets. It is common for a single ARM binary to include instructions from multiple sets and switch among them during execution. Broadly, the Thumb instruction set and its variations are used to reduce code size with minimal reduction in performance. We further describe on the details of the Cortex-R architecture and its implications for the design of RECFISH in Section 3.

3 Design of RECFISH

RECFISH is a software defense for embedded ARM architectures. RECFISH takes a control flow graph and program binary as input, adds security instrumentation, and produces a protected binary. We divide the discussion of RECFISH into four components: *(i)* basic memory protections, *(ii)* forward-edge CFI, *(iii)* shadow stack operations, and *(iv)* secure context switching. The first three are presented in the context of bare-metal execution and the last in the context of FreeRTOS.

3.1 Threat Model

We assume a powerful adversary able to modify anything in writeable memory at any time, including all data on the stack or heap. The attacker cannot, however, modify read-only memory such as program code. Unlike other software defenses, we also assume that writeable memory is, by default, executable. Consequently, we must implement basic memory protections as part of RECFISH.

The attacker's goal is to subvert the control-flow of a program by modifying the target of an indirect branch. In ARM, an indirect branch is either *(i)* a branch instruction with a register operand, or *(ii)* any operation with the program counter register as the destination. These instructions are enumerated in Appendix A. RECFISH is charged with thwarting such attacks. As with previous work, RECFISH does not prevent memory corruption, but it does prevent corrupted code pointers from hijacking control-flow.

In a system executing without RECFISH modifications all of RAM is configured, by default, to be readable, writeable, and executable. The code is stored in ROM which is only readable and executable. Discussed in detail below, RECFISH uses binary instrumentation to check the targets of indirect branches and leverages the MPU to disable the execute permissions for RAM and to create a region of protected memory for the shadow stack (and other security-critical structures in FreeRTOS). Most code executes in an unprivileged mode and this protected memory region is only accessible from privileged modes.

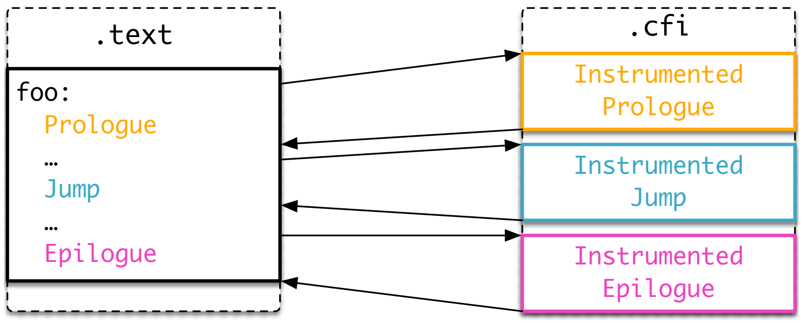


Figure 1 RECFISH uses trampolines to add CFI instrumentation to binaries without access to the source code.

3.2 RECFISH for Bare-Metal Execution

RECFISH patches pre-compiled binaries to add security instrumentation. Binary patching promotes broad adoption of the defense as it allows developers to employ RECFISH without modifying existing toolchains. This capability is important for retrofitting security to existing devices that may otherwise never receive updates.

However, binary patching is more complicated than simply inserting additional instructions into the code section. Namely, the inserted instructions can break the relative addressing common in the ARM Thumb instruction set. For example, the instruction `ldr r1, [pc, #32]` loads data from an address 32 bytes after the program counter. With in-line checks, we must update this instruction (and likely many others) to point to the new location of the data. To avoid this issue, we instead instrument instructions by replacing them with *trampolines*, i.e., direct branches to CFI code appended to an unused memory section. At a high level, the patched binary follows the format shown in Figure 1. The original program code is in the `.text` section, and the CFI instrumentation goes into a new `.cfi` section.

3.2.1 Basic Memory Protections

Like previous CFI implementations, RECFISH depends on two basic memory invariants. First, code regions must be read-only. Second, writeable regions must be non-executable. Unlike x86, Cortex-R does not offer virtual memory support, so these protections must be implemented using the limited functionality of the memory protection unit (MPU) and privileged processing modes.

The MPU, included by most Cortex-R processors, supports developer-defined permissions for up to 12 memory regions. For each region, there are three sets of permissions to be set: user mode, privileged mode, and execute permissions. For example, a memory region can be configured to be read-only for user mode, read and write for privileged mode, and non-executable (in any mode). In addition to the basic memory protections described above, we also leverage the MPU to create a protected region for shadow stack operations (see Section 3.2.3). MPU violations result in a data abort and any attempts by an adversary to modify program code or execute from writeable memory will be prevented.

To enforce basic memory protections, RECFISH requires the device to have an MPU and two available MPU regions. This includes many Cortex-M, Cortex-R, and RISC-V devices. The primary difference between Cortex-M and Cortex-R, in the context of this work, is that the former uses memory-mapped registers that must also be protected – though this can be done in the same manner that RECFISH protects the shadow stack.

Listing 1 Example function that uses function pointers.

```
1  int foo(int a, int b) {
2      int (*func[2])(int, int) = {add, sub};
3      static unsigned int i = 0;
4      return func[i++
```

Listing 2 Disassembly of the `foo()` function.

```
0x192:    push {r4, r7, lr} #
0x194:    sub sp, 20        # Function Prologue
0x196:    add r7, sp, 0     #
...
0x1e6:    add r3, r3, r4    #
0x1e8:    blx r3            # Indirect Call
...
0x1f0:    mov sp, r7        #
0x1f2:    pop {r4, r7, pc}  # Function Epilogue
```

Cortex-R processors have seven processing modes: User, System, Supervisor, Interrupt, Fast Interrupt, Abort, and Undefined. We leverage these different modes to perform operations at different privilege levels. Specifically, we use the *unprivileged* User mode for normal code execution and forward-edge CFI checks and the *privileged* System, Supervisor, and Interrupt modes for other CFI functionality such as shadow stack operations and context switching. The most relevant distinction between privileged and unprivileged modes is that the former can access memory marked as privileged-only.

3.2.2 Forward-Edge Checks

We use a simple example to illustrate how RECFISH handles forward-edge checks with binary patching. Consider the function `foo()` given in Listing 1 and its disassembly shown in Listing 2. RECFISH instruments three components of this function: the prologue, the indirect call resulting from the function pointer usage on line 4, and the epilogue. Listing 3 shows the resulting instructions after RECFISH is applied; namely, each component is overwritten with trampolines to RECFISH instrumentation.

Listing 3 Instrumented version of the `foo()` function.

```
0x192:    b.w 0x13f60       # Branch to new prologue

0x196:    <label>           # Insert label
...
0x1e6:    bl 0x13f80        # Replace indirect call
                            # with CFI check
...
0x1f0:    b.w 0x13f98       # Branch to new epilogue
...
```

Listing 4 Function prologue instrumentation.

```
0x13f60:  push {r4, r7}      # Copy displaced instructions
0x13f62:  sub sp, 20         #   from the orig. prologue
0x13f64:  add r7, sp, 0      #   with modifications
0x13f66:  svc 0              # Call ss_push
0x13f68:  b.w 0x198          # Branch back, skipping label
```

Listing 5 Indirect call instrumentation.

```
0x13f80:  add r3, r3, r4     # Copied instruction
0x13f82:  push {r0, r1}      # Save registers
0x13f84:  ldrh r0, [r3, 3]   # Load target's CFI label
0x13f86:  movw r1, <label>   # Load expected label
0x13f88:  cmp r0, r1         # Compare the labels
          error:
0x13f8a:  bne error          # Error if mismatch
0x13f8c:  pop {r0, r1}       # Restore registers
0x13d8e:  bx r3              # Perform indirect jump
```

3.2.2.1 Function Prologue

RECFISH instruments the function prologue to embed the appropriate CFI label for `foo()`. This allows any calling function to verify that `foo()` is a legal target. As mentioned previously, RECFISH cannot simply insert this label without breaking relative addressing. Instead, RECFISH replaces 6 bytes of the original function prologue with a 4-byte branch to the CFI section (i.e., the trampoline) and a 2-byte label. The CFI section for the function prologue, shown in Listing 4, includes the instructions replaced in the original prologue, adds some shadow stack operations (discussed later), and returns to the instruction following the original function prologue.

3.2.2.2 Indirect Branches

RECFISH ensures that the target of the indirect branch is legal by checking the value of the target's label against the expected value. As with the prologue, RECFISH inserts these checks into a separate CFI code region (Listing 5) and uses a trampoline to jump to the check. Note that the target's label is stored in a function prologue – similar to what was discussed above for `foo()` – and the expected label is hard-coded into the instruction at `0x13f86`. In the event of a label mismatch, the instruction at `0x13f8a` will branch to error handling code, which in the current implementation will result in an infinite loop.

RECFISH must replace the 16-bit indirect branch instruction with a 32-bit direct branch to the CFI check. To make space, RECFISH replaces both the indirect branch and the preceding `add` instruction. The displaced `add` is moved to the start of the appropriate CFI code region. We use a branch-and-link operation as the direct branch to copy the return address into the link register for use by the called function.

3.2.2.3 Function Epilogue

The modified function epilogue reverses the operations performed during the new prologue. Namely, RECFISH restores the registers previously pushed to the normal stack and pops the return address from the shadow stack. To do this, RECFISH again replaces two 16-bit

Listing 6 Function epilogue instrumentation.

```
0x13f98:  mov sp, r7         # Execute displaced operation
0x13f9a:  svc 1              # Call ss_pop
0x13f9c:  pop {r4, r7}       # Perform pop without PC
0x13f9e:  bx lr              # New return instruction
```

instructions with a 32-bit trampoline. In the instrumentation shown in Listing 6, the modified epilogue includes code to retrieve the return address from the shadow stack and move it into the link register. The original `pop` instruction is also modified such that the link register is no longer included in operands. This modification is necessary as the link register is not pushed to the normal stack in the new prologue. Finally, the code returns to the calling function using a branch-and-exchange to the link register.

3.2.3 Shadow Stack

The shadow stack is a region of memory, separate from the normal stack, used to securely store return addresses and increase the runtime precision of RECFISH checks. RECFISH makes the shadow stack inaccessible from the User processing mode using the MPU, but allows reading and writing from the Supervisor mode. Consequently, performing shadow stack operations (e.g., push and pop) requires RECFISH to jump into a privileged mode, modify the shadow stack, and jump back to an unprivilege mode. To implement this, we created a system call interface using the ARM Supervisor call (`svc`) instruction.

The Supervisor call instruction takes one operand, an immediate value representing the function number. When it executes, the `svc` instruction triggers an interrupt on the processor. The handler for this interrupt determines the function number by reading the opcode of the software interrupt instruction. The ARM assembly code function to do this is shown in the Appendix in Listing 7. The short ARM assembly code functions for the shadow stack operations are shown in the Appendix in Listing 8.

Procedure calls are handled differently in ARM than x86. In x86, procedure calls are generally implemented with pairs of `call` and `ret` instructions. The `call` instruction is used to branch to the target procedure, saving the return address onto the stack. The associated `ret` instruction is later used to return to the caller, popping the return address off the stack. In ARM, procedure calls are implemented using a branch-link-exchange instruction [16]. Branch-link-exchange instructions atomically branch to the target location stored in the link register and then update the link register to store the return address. We leverage this behavior to reduce the number of writes to the shadow stack. Specifically, we only need to push `LR` to the shadow stack if the link register is spilled to the normal stack, e.g. when a procedure calls another procedure.

3.2.4 Implementation

Our prototype implementation uses the Capstone disassembly engine [1], the pyelftools [3] ELF file parser, and the Keystone assembler [2]. Capstone provides a powerful disassembly and instruction decomposition framework that makes it possible to identify the registers modified by any instruction. We search the executable for indirect branches and instructions that indirectly modify the program counter register (such as a load multiple operation where PC is a destination register). After enumerating the instructions that need instrumentation, we follow the procedures outlined earlier in this section to generate the instrumentation.

Finally, we use the Keystone assembler to write the patched code to a new binary. We follow the same procedure for function prologues, epilogues, and indirect branch targets until we have a fully instrumented binary.

3.2.4.1 Limitations

One limitation of our current implementation is that the size of the shadow stack must be manually configured. However, RECFISH uses additional instrumentation to ensure that the stack does not overflow.

Uncommon C features such as `setjmp` and `longjmp` pose additional challenges that our current implementation does not directly address. Though such functionality was not employed by any of the binaries we evaluated, we can extend RECFISH to support `setjmp/longjmp` without a substantial impact on the security or performance results presented. For example, we can adopt an approach similar to that used by DISE [13] and push the current stack pointer along with the return address to the shadow stack.

3.3 RECFISH for FreeRTOS

The primary challenge of extending RECFISH to the FreeRTOS real-time operating system is supporting context switching and multithreading. As we discuss below and in the following section, task preemption and the lack of memory isolation introduces the possibility of a memory error in one task being used to corrupt the memory of another.

Each task has its own stack. This stack is also used by the scheduler to save and restore state when switching from one task to another. Under our threat model, RECFISH must assume any information stored on the stack during a context switch could be modified by the attacker. Consequently, RECFISH cannot consider CFI checks as atomic operations as any context-switches that preempt a CFI check could introduce a time-of-check to time-of-use vulnerability. Specifically, CFI labels are loaded from read-only program code into registers. In the presence of context switching, however, registers with CFI-critical information – i.e., the two registers with labels and the register storing the branch target – could be saved to the task stack at any point during the CFI check. With careful timing, the attacker could overwrite these saved register values. Defenses on general-purpose systems do not have to address this challenge (for process threads) because register values are saved to kernel memory during a context switch.

To avoid this vulnerability, RECFISH saves task state in the task's shadow stack rather than on the task's unprotected regular stack. For FreeRTOS, this necessitates modification of the Task Control Block (TCB) structure, the task creation procedure, and the scheduler. The scheduler already runs in privileged mode with interrupts disabled, so we do not incur additional overhead from the Supervisor call instruction.

One alternative to using the shadow stack for context switching would be to disable interrupts during the CFI checks. However, we avoid this approach as it introduces additional scheduling challenges, i.e., it introduces a new source of latency for real-time tasks. Even in non-preemptive systems (where the scheduler only runs when a task yields), other real-time sensitive hardware interrupts could be negatively impacted by disabling interrupts.

3.3.1 Task Creation Modifications

We modified the FreeRTOS task creation procedure to assign a shadow stack to each task when it is created. Specifically, we extended the Task Control Block (TCB) structure to add a field for a shadow stack. We also modified the functions that initialize this structure, the FreeRTOS function `prvInitialiseNewTask` and the port-specific FreeRTOS function `pxPortInitialiseStack`. The `prvInitialiseNewTask` simply assigns the next available shadow stack to the TCB of the newly created task. The port-specific `pxPortInitialiseStack` function required more complicated changes. When FreeRTOS creates a task, it sets up the stack such that the task appears to have been switched out by the scheduler. This is an optimization that allows FreeRTOS to simply use its restore context routine to start a task, rather than needing a special procedure.

3.3.2 Scheduler Modifications

Most of the scheduler is written in ARM assembly, and the instruction set makes it easy to save context to the unprotected stack. Normally, to save the register information to the stack from the scheduler, only two instructions are needed: `srs` and `push`. The `srs` mnemonic is the *store return state* instruction, which pushes the IRQ return address and the saved process state register to the system or user mode stack. After saving the return state, the scheduler switches to system mode and pushes the rest of the registers to the stack. To modify this to use the shadow stack, we need to get the pointer to the top of the shadow stack, and use this like the stack pointer.

4 Evaluation

To evaluate RECFISH, we conducted three broad classes of experiments. First, we evaluated the security provided by RECFISH to show that the instrumentation will enforce the CFI policy, even in the presence of a powerful attacker. Second, we conducted a series of microbenchmarks to understand and quantify the overheads and costs associated with specific functionality within our instrumentation, and how commonly such functionality is invoked among many benchmark applications. Finally, the results of these microbenchmark experiments informed the design of a large scale *schedulability study*, or effectively a macrobenchmark study. This study demonstrates how the RECFISH overheads affect the guarantees that a given task system will meet all deadlines.

The reference system used for this work is a Texas Instruments Hercules RM46L852, an ARM Cortex-R4F processor. This processor has 1.25 MB of non-volatile program flash, 192 KB of RAM, and an additional 64 KB of flash for emulated EEPROM storage.

4.1 Security Evaluation

The security benefits of control-flow defenses are difficult to describe quantitatively. While measurements of ROP gadget reduction and Average Indirect Target Reduction (AIR) have been used in previous work, Carlini et al. have discussed how these measurements are misleading and reflect CFG precision more so than security [10]. Instead of using the aforementioned metrics, we adopt the standard qualitative analysis used in prior work on control-flow defenses for general-purpose systems; we show that RECFISH checks that all branches are legal with respect to the control-flow graph, those checks cannot be bypassed, and the shadow stack cannot be modified by an attacker. Because our focus is on the security of the proposed CFI instrumentation rather than the precision of the control flow graph, attacks that follow a legal control flow are out of the scope of this evaluation.

In addition to the qualitative analysis, we also tested RECFISH using the empirical methodology proposed by Carlini et al. – the Basic Exploitation Test (BET) – where a minimal, representative program is written with a known vulnerability (such as a buffer overflow) to show that a defense prevents an attacker from achieving their specific goal (i.e. arbitrary code execution) [10]. We elide further discussion of the BET results as RECFISH successfully prevented the attack.

4.1.1 Basic Memory Protections

RECFISH leverages the MPU to set memory as either writeable or executable, but not both. This prevents an attacker from inserting and executing their own code to bypass the CFI checks. Further, the attacker cannot modify program code to disable CFI checks. These two basic protections ensure the attacker's only attack vector is to modify writeable memory.

4.1.2 Label Assignment

The labels for CFI instrumentation must be chosen to satisfy the *global uniqueness assumption*. This assumption states that the byte sequence representing a label only appears in the code section as part of the CFI instrumentation. If this assumption does not hold and the label coincidentally appears somewhere else in code memory (e.g. as an instruction opcode), an attacker could circumvent CFI by overwriting a code pointer with an address that is the correct offset from the location where the erroneous label appears. Given that RECFISH patches pre-compiled binaries and that there is no dynamic linking in our target system, we can use static analysis to verify that all labels are globally unique.

Further, because the label is stored in executable code, the instrumentation should either ensure that the label is never executed, or it should be a side-effect free instruction. In the original CFI implementation, the side-effect-free x86 `prefetch` instruction was used to encode the label [5]. In RECFISH, the forward-edge and shadow stack protections to prevent the label from being executed.

4.1.3 Forward-edge Instrumentation Without Context Switching

Each forward-edge check has two parts: the source and destination instrumentation. The source instrumentation replaces the indirect branch and its preceding instruction with a direct branch to the correct location in the `.cfi` section. The general format of the indirect call instrumentation is shown in Listing 5. All critical operations of the CFI check are performed entirely in registers and thus are protected in the shadow stack if the check is pre-empted.

The source label is hardcoded in a `mov` instruction, so that cannot be modified. Consider the case where the target label matches the expected label. In this scenario, the target label either resides in read-only program code, or it has been inserted into writeable memory by the attacker. If the label is in program code and the labels are globally unique, the label must be valid and it precedes a legal branch target. If the label was maliciously inserted into writeable memory, the CFI check will allow the branch to be taken, but the MPU will prevent the processor from executing the code at the target. In summary, there are three possible outcomes from the label checking code: the branch is taken and execution continues, the branch is taken and the MPU prevents execution, or execution enters an infinite loop. In any of the three cases, the attacker cannot achieve arbitrary code execution.

4.1.4 Backward-edge Instrumentation With Shadow Stack

Each backward-edge check has two parts: function prologue and function epilogue instrumentation. In ARM, we do not need to consider the backward-edge in leaf functions, that is, functions at the end of a call tree that do not call any other functions. Leaf functions do not push the return address to the stack; they keep the return address in the link register and end the function with a `bx lr` instruction. In non-leaf functions, however, the compiler will generate a matching pair of `push {<reglist>, lr}` and `pop {<reglist>, pc}` instructions to store the return address on the unprotected stack. All non-leaf functions are instrumented by RECFISH.

The instrumentation for non-leaf functions has a single goal – protect the return address by saving the link register on the shadow stack rather than on the unprotected regular stack. The general form for this instrumentation is shown in Listings 4 and 6. Importantly, memory accesses always occur at the current hardware privilege level (i.e., privileged or unprivileged). Most program code executes in unprivileged mode, but RECFISH places the shadow stack in a memory region accessible only during privileged mode execution. RECFISH uses the software interrupt (`svc`) instruction to change the processing mode from user to supervisor mode, allowing it to modify the shadow stack.

To manipulate the shadow stack, the attacker must exploit a memory corruption vulnerability while the processor is in privileged mode. Interrupt handlers execute in privileged mode, and thus are a potential avenue of attack; however, interrupt handlers in real-time systems are designed to be short and deterministic and can be designed without arbitrary memory writes. Finally, the shadow stack code itself is effectively atomic. If a context switch occurs during shadow stack operations, RECFISH pushes all of the context to the shadow stack, so there is no time-of-check to time-of-use vulnerability.

4.1.5 Forward-edge Instrumentation With Context Switching

The only time that the CFI checks can be tampered with is when context switching is possible. If the scheduler interrupts the CFI check and puts CFI-critical registers into memory, our threat model dictates that the attacker could use this as an opportunity to corrupt the saved CFI-critical registers. When context is restored, the corrupted values will be loaded into the registers, potentially allowing the attacker to bypass CFI. As stated previously, we combat this issue by storing all saved context in the shadow stack.

Since the scheduler runs with interrupts disabled, the scheduler operations are atomic from the perspective of program code. This means that there is no opportunity for an attacker to corrupt the context before it gets pushed to the protected shadow stack. Further, an attacker cannot change the pointer to a task's shadow stack because RECFISH protects the entire Task Control Block using the same privileged MPU region as the shadow stack.

4.2 Performance Impact

To measure the performance impact of RECFISH, we look at four different measurements. First, we use an embedded system benchmark to determine the overhead associated with the bare metal instrumentation. Second, we look at the additional latency added to FreeRTOS context switching by adding the shadow stack. Third, we analyze the resource requirements for RECFISH via microbenchmarking. Finally, we perform a large-scale schedulability analysis to assess the suitability of RECFISH for real-time systems.

4.2.1 CPU Benchmarks

RECFISH is designed to work on embedded systems without a traditional operating system, thus benchmarks designed for general-purpose machines such as the SPEC CPU2006 benchmark are not appropriate for this evaluation. To measure the raw overhead associated with

CFI checks on a realistic workload, we used the CoreMark embedded system benchmark [18] and the BEEBS benchmark suite [34]. These easily portable applications run on a variety of embedded architectures. CoreMark performs various common embedded tasks, like matrix manipulation, linked list manipulation, state machine operations, and cyclic redundancy check (CRC) calculation. BEEBS combines benchmarks from MiBench [21], WCET [20], and DSPStone [42].

On our TI RM46L852 evaluation board, we measured the CoreMark score both with and without CFI using the default settings and 1000 iterations. Without CFI, the recorded CoreMark score was 97.371, which is a reasonable score for that hardware running in Thumb mode. With CFI, we recorded a score of 76.767, a decrease of about 21% compared to the non-CFI score. Additionally, we recorded an approximately 30% increase in total execution time for the benchmark code. In this evaluation, all default settings were used, and 1000 iterations were run of the benchmark.

In the BEEBS benchmarks, over 70% of the applications saw less than 25% overhead. On benchmarks with few or no function calls and no indirect branches, we see no significant difference in execution time. However, in benchmarks like `recursion` and `fac` which both use recursive function calls, we see up six times slowdown. In practice, real-time embedded systems avoid using recursion because it introduces nondeterminism and can result in quickly running out of memory, so a slowdown of this magnitude is unlikely to occur in production systems. The `trio-snprintf` and `mergesort` benchmarks both have many indirect branches, but they only see about 0.5 times slowdown. The CFI checks are fast relative to other computation performed by these benchmarks.

4.2.2 Additional Resource Use

RECFISH requires an additional 10 bytes of storage per indirect branch and 8 bytes per non-leaf function prologue and epilogue. While we cannot generalize the number of non-leaf functions and indirect branches in any given program, the CoreMark benchmark required 964 bytes of instrumentation code for a 10 KB binary – just under a 10% increase in binary size.

RAM usage depends on the system being instrumented. On bare metal systems, a single shadow stack is required, which on our evaluation system, we used a shadow stack size of 256 bytes plus 12 bytes for the shadow stack structure – a total of 268 additional bytes of RAM for the shadow stack. In FreeRTOS, however, we used a larger shadow stack, since context information is stored on the stack, so we required 528 bytes per task, which encompassed a 512-byte shadow stack, 12-byte shadow stack structure, and 4-byte pointer to the shadow stack stored in each Task Control Block.

Finally, our implementation depends on some additional resources. We need at least two MPU regions to prevent execution from RAM and to protect the shadow stack. On our hardware, a maximum of 12 regions could be configured, so our utilization was minimal. Also, we require two supervisor calls out of a possible 256 available in Thumb mode.

4.3 Microbenchmarks

To better understand the overhead introduced by RECFISH, we measured each component of the instrumentation separately. In this section, we examine the number of CPU cycles RECFISH adds to indirect branches, function prologues, and non-leaf function epilogues.

4.3.1 Microbenchmark Design

We measured CPU cycles using the ARM Performance Monitoring Unit (PMU), configuring the PMU to count three events: CPU cycles, predictable branches, and incorrectly predicted branches. For each component, we took these measurements for a few different scenarios: no instrumentation, inline instrumentation, and the trampoline-based instrumentation used by RECFISH. We ran these microbenchmarks under different configurations of the branch predictor. By default, ARM uses a 256-entry, 2-bit history-based branch predictor with a hardware return stack [6]. The branch predictor can be configured to use static policies rather than the history-based policy and the return stack can be disabled. For the schedulability study, we also measured the context switch overhead in FreeRTOS, both with and without the shadow stack. For this measurement, we used the default branch predictor configuration.

4.3.2 Results

Under default CPU settings, we found that unconditional indirect branches without RECFISH instrumentation take 11 CPU cycles to execute. When adding inline checks to these branches, we saw a varied number of cycles. In the worst-case, the forward-edge CFI check takes 41 cycles, although we only see this worst-case result in the first iteration of the experiment. In later iterations, the branch predictor determined that the conditional branch inside the CFI check was not likely to be taken, so the CFI check sped up to 26 cycles after 5 iterations. For the trampoline method used in RECFISH, we saw a worst-case forward-edge check of 61 cycles, which sped up after 5 iterations to 44 cycles. By examining the execution time under different branch predictor settings, we could account for the majority of variability in execution time. By disabling the branch predictor's dynamic history function and return stack, only the first iteration of each microbenchmark was slower than the rest. We were unable to determine the cause of this slowdown, but potential causes could include pipeline stalls, pipeline flushes, or a conflict that prevents the CPU from dual-issuing instructions.

The other component that RECFISH affects is non-leaf function calls, since these functions must save the return address at the start of the function and restore it at the end. RECFISH requires that the return address be stored in the shadow stack, rather than on the unprotected user stack. Since there are no conditional branches in the shadow stack operations, we saw a constant increase from 19 cycles for the combined function prologue and epilogue without CFI to 275 cycles with RECFISH. Most this overhead is associated with changing the execution mode from User mode to Supervisor mode. Additionally, during these 275 cycles, interrupts were disabled twice for 11 cycles during the handling of the `svc` instruction, which was used once in the prologue and once in the epilogue. Since the combined function prologue and epilogue in RECFISH requires significantly more CPU cycles than the unmodified binary, we expect more performance degradation in binaries with many calls to short non-leaf functions. By contrast, programs with many calls to longer functions (or leaf functions) will see less performance degradation from shadow stack operations. Indeed, this matches our previous observations of the macrobenchmarks.

The final microbenchmark that we measured was FreeRTOS context switch overhead, which is critical to the schedulability study in Section 4.4. Without RECFISH, FreeRTOS context switches take a total of 120 cycles, 57 for saving context and 63 for restoring it. With RECFISH, we saw a moderate increase of context switch time to 159 cycles, 80 for saving and 79 for restoring.

Table 1 Microbenchmark results for individual components with the default branch predictor. All units are CPU cycles.

Component	CFI Type	Worst Case	Best Case
Indirect Branch	No CFI	11	11
Indirect Branch	Inline CFI	41	26
Indirect Branch	RECFISH	61	44
Function Call	No CFI	19	19
Function Call	Inline CFI	237	237
Function Call	RECFISH	275	275

4.4 Schedulability Study

Next, we incorporate the microbenchmark results into a large-scale schedulability study, which demonstrates the effect RECFISH has on the ability to ensure that all deadlines will be satisfied.

4.4.1 Schedulability

We begin our schedulability discussion with the *periodic task model* [28], which is implemented in FreeRTOS. In this model, a *task system* τ is composed of a set of n *tasks*, denoted $\tau = \{T_1, \ldots, T_n\}$. Each task is mathematically modeled as a tuple, $T_i = (e_i, p_i)$, and is comprised of a (potentially infinite) sequence of *jobs*, which are common invocations of the same logic. Each job of T_i executes for at most e_i time units, or its *worst-case execution time* (*WCET*). Jobs of T_i are *released* or made ready for execution every p_i time units, and must complete by their *deadline*. We assume the deadline of each job is p_i time units after it is released, i.e., when the next job of the task is released. The utilization of T_i is given by $u_i = e_i/p_i$, and the utilization of the task system U, is the sum of all tasks' utilizations, $U = \sum_{T_i \in \tau} u_i$. We assume fixed-priority scheduling, and evaluated schedulability using standard fixed-priority time-demand analysis [27].

4.4.2 RECFISH Schedulability

RECFISH introduces sources of overhead that do not exist in an unprotected system. In the rest of this section, we demonstrate how these overheads affect schedulability. To do so, we must first consider how the RECFISH overheads should be incorporated into the time-demand analysis. In our overhead analysis, we only consider overheads incurred on normal control-flow paths. While detecting and triggering an exception on invalid control flow incurs overhead, the code that must execute to handle such an exception is highly application specific and we thus exclude them from the scope of this study.

RECFISH introduces several sources of overhead, which are described more completely and quantified in Section 4.3. These overheads fall into two distinct categories: runtime checks, which occur at indirect branches and function prologues/epilogues, and context-switch-related overheads. These two types of overheads are accounted for analytically using different techniques.

We can account for the time spent in CFI checks by inflating the execution time of the task.[1] We assume that each CFI check at an indirect branch (respectively, function epilogue and prologue) imposes an overhead of Δ^{b} (respectively, Δ^{f}), and that each job of T_i executes

[1] We note that for a mere 11 cycles, interrupts are disabled. This is handled as a *priority inversion* and is incorporated into our analysis.

at most C_i^b indirect branches and C_i^f functions. To incorporate the overhead of all of the CFI checks, we simply inflate the execution time of each task to account for the time spent in CFI checks, $e_i' = e_i + C_i^b\Delta^{\mathrm{b}} + C_i^f\Delta^{\mathrm{f}}$.

The second source of overhead in RECFISH is the additional context-switch overhead associated with handling the shadow stack. We denote this overhead by Δ^{c}. Specifically, Δ^{c} includes both the time to save the context of one process, as well as restore the context of the next. We leverage existing techniques for handling context-switch overhead [8]. Instead of charging the overhead of the context switch to the task whose context is being saved or restored, instead, analytically, we charge the overhead to the preempting, higher-priority task, as is commonly done for analyzing cache-related preemption delays [38]. Notably, each job can only preempt at most one other job. Therefore, we analytically inflate each task's WCET to account for this overhead as, $e_i' = e_i + \Delta^{\mathrm{c}}$.

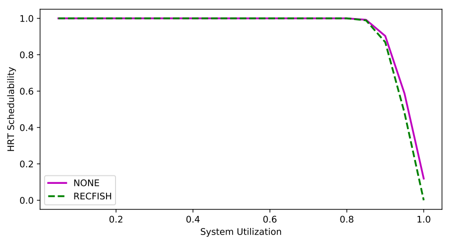
The other modifications to FreeRTOS required from RECFISH, such as augmenting the TCB and the task initialization procedures do not affect schedulability. It is quite common for a real-time system to startup and initialize the real-time tasks before entering a real-time mode, during which all deadlines must be satisfied. The performance implications of the remaining aspects of RECFISH fall within this initialization mode, and therefore do not affect schedulability. As such, in our schedulability experiments, we consider that when RECFISH is enabled, the execution time of each task is analytically treated as $e_i' = e_i + \Delta^{\mathrm{c}} + C_i^b\Delta^{\mathrm{b}} + C_i^f\Delta^{\mathrm{f}}$. In the context of this study, these are the only overheads considered so as to focus on the specific effects the RECFISH-specific overheads have on schedulability.

4.4.3 Experimental Design

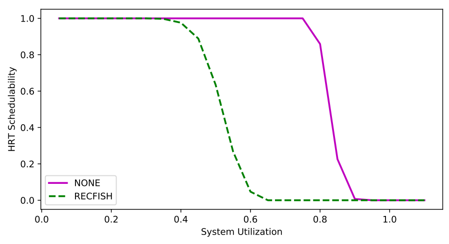
We conducted a large-scale schedulability study to evaluate the tradeoff between security and schedulability enabled by RECFISH. Practical real-time applications have varying task-system parameters, and the interplay among these parameters, analysis pessimism, and implementation overheads can have significant schedulability implications. Also, an overhead may be observed to be minor, but if it is unpredictable, difficult to incorporate into schedulability, or otherwise subject to analysis pessimism, it may significantly affect schedulability. Accordingly, we consider many classes of real-time task systems in our experimental design.

Overall, our schedulability study is conducted as follows. Using several different random distributions, we generate over six million analytical task systems with different parameter values. We then evaluate the schedulability of each task system both with and without RECFISH applied.

We randomly generated sporadic task systems using a similar experimental design as previous studies [8]. We generated task systems with a total system utilization in $U \in \{0.05, 0.1 \ldots, 1.0\}$. Per-task utilizations in each task system were chosen to be *light*, *medium*, or *heavy*, which correspond to uniformly distributed utilizations in the range $[0.001, 0.1]$, $[0.1, 0.4]$ and $[0.5, 0.9]$, respectively. Tasks were randomly generated using the chosen distribution until the desired system utilization was reached. The periods of all tasks were chosen uniformly from either $[3, 33]$ ms (*short*), $[10, 100]$ ms (*moderate*), or $[50, 250]$ ms (*long*). Based on the micro-benchmark experiments presented previously, we assume $\Delta^{\mathrm{c}} = 39$ cycles. We also considered two different values for $\Delta^{\mathrm{b}} \in \{33, 50\}$ cycles, which reflect the overhead at an indirect branch if branch correctly predicted or not. (In provisioning a hard real-time system, one may assume branches are always mispredicted, whereas in a soft real-time system, less analysis pessimism may be necessary.) We also measured $\Delta^{\mathrm{f}} = 256$ cycles.



(a) Bimodal indirect branches, few functions, short periods, heavy utilizations.



(b) Common indirect branches, frequent functions, moderate periods, light utilizations.

Figure 2 Example schedulability graphs.

Based on our microbenchmark results, we considered several distributions for the frequency of indirect branches and functions within each task. We considered that either no indirect branches were taken (*None*), or the number of indirect branches were uniformly chosen at a rate of one indirect branch among $[10^3, 10^5]$ cycles (*common*), $[10^6, 10^7]$ cycles (*rare*), or *bimodally* between the two distributions none (90%) and common (10%). Similarly, we considered the following distributions for the number of of functions: the total number of functions per task is chosen uniformly among $[1, 100]$ (*few*), or uniformly at a rate of one function among $[10^2, 10^3]$ cycles (*frequent*), $[10^3, 10^4]$ cycles (*moderate*), or bimodally between the two distributions moderate (90%) and few (10%).

We considered the cross product of these possible system parameters, resulting in 5,760 unique configurations. For each configuration, we generated and evaluated 1,000 task systems for schedulability, for a total of over six million task systems.

The chosen taskset parameters were pioneered by Brandenburg [8] and have been widely used in the community. We extended the taskset generation models to account for the frequency of indirect branches and functions, based upon data from the benchmarks we measured. To our knowledge, our work is the first to consider the effect of a control-flow hijacking defense on schedulability, and therefore we could not compare against other defenses from a schedulability perspective.

4.4.4 Schedulability Results and Observations

Based on these experiments, we generated 288 *schedulability graphs*, two of which, which demonstrate the performance extremes, are depicted in Figure 2. From these figures, we draw several observations.

Observation 1: RECFISH has a negligible impact on schedulability for some classes of task systems. This observation is supported by inset (a) of Figure 2. In this system configuration, there are relatively few tasks (because of the heavy utilizations) and relatively few CFI checks on indirect branches or functions. As a result, RECFISH has a negligible effect on schedulability, while improving security.

Observation 2: RECFISH has a significant impact on schedulability for some classes of task systems. This observation is supported by inset (b) of Figure 2. In this system configuration, there are many tasks given the light task utilizations, and each of those tasks has many CFI checks on both indirect branches as well as function calls. In such a system configuration, we would expect RECFISH to have a more significant effect on schedulability.

In this case, there is roughly 30% *utilization loss* due to RECFISH, i.e., CFI checks and their impact in overhead analysis cause 30% of the available system utilization to be sacrificed in order to meet all deadlines.

Observation 3: Across all generated task systems, 85% of those that were schedulable without RECFISH, were schedulable with RECFISH. This aggregate statistic demonstrates the practical applicability of RECFISH. In 85% of the generated task systems, RECFISH could be applied without compromising schedulability. Only those task systems that stress the available computing resources are unlikely to be schedulable in the presence of RECFISH. From these results, we believe that RECFISH can be deployed in most RTES with only a minimal increase in system size, weight, and power.

4.4.5 Optimization Opportunities

While a real-time system is composed of many tasks, only a subset of those tasks typically take external input, i.e. directly interact with the adversary. Any exploitation must involve one of those tasks. Specifically, a task is *unsafe* if it accepts external inputs from the user, and *safe* otherwise. We denote safe (resp. unsafe) tasks with the superscript T^S (resp. T^U). We can reduce the overhead of RECFISH and improve schedulability on many task systems by controlling the execution order of safe and unsafe tasks.

Consider the following example with two tasks T_i^S and T_j^U. Let us assume that T_j^U takes external input, contains a memory error, and the attacker can leverage that error to corrupt memory. Let us also assume that T_i^S does not contain any errors nor does it take external input. Without RECFISH, if T_j^U has a higher priority than T_i^S then T_i^S can be pre-empted by T_j^U. Thus, the attacker can leverage the bug in T_j^U to manipulate memory used by T_i^S. Therefore, RECFISH checks are needed in both T_i^S and T_j^U to provide control-flow integrity in this situation. If we prevent the preemption of T_i^S by T_j^U, then we need not conduct the CFI checks on T_i^S, only T_j^U. This can be realized either by marking such tasks as non-preemptive, or in fixed-priority scheduling which is supported in FreeRTOS and our RECFISH implementation, by increasing the priority of T_i^S over that of all unsafe tasks. Therefore, by carefully choosing priorities, we can eliminate the need for some CFI checks and reduce the security overhead, while still providing the same security guarantees.

This raises the question, how should tasks be prioritized to minimize the number of RECFISH-related CFI checks? We consider a technique we call *task pushing* in which the priority of otherwise safe tasks are increased above the unsafe tasks. While *task pushing* initially seems to have a negative impact on schedulability, we find that the resulting reduction in RECFISH overhead actually increases the number of schedulable task sets.

To test task pushing, we generated additional task sets using the same configurations as discussed earlier, and added an additional parameter for the probability of a task being labeled safe. We considered several distinct values for this probability, 0%, 25%, 50%, and 75%, which was constant for each generated task system. We then used a brute-force algorithm to test all possible combinations of pushed tasks. We find that with task pushing, the percentage of schedulable task sets with RECFISH increases from 85% to 88%, 91%, and 95%, respectively. We also measured through simulation[2], the total amount of overhead observed during a hyperperiod, or the point at which the schedule repeats (the least-common

[2] We assumed for this simulation that the execution time e_i was exact.

multiple of all periods). The average RECFISH overhead observed across all generated task systems was 12%, 8%, and 4%, for safe-task probability of 25%, 50%, and 75%, respectively. This is down from 16% overhead when RECFISH checks are applied to all tasks.

While these results are promising, there are still many open questions. For example, for now we assume the developer provides the safe/unsafe label, but can automated mechanisms (e.g., static program analysis) be leveraged to provide these labels? Can RECFISH-related overheads be further reduced under different schedulers (e.g., non-preemptive unsafe tasks, or other more dynamic scheduling policies)? How should information passing between safe and unsafe tasks be handled? Intuitively, we believe it is simpler and more efficient to secure a few well-defined interfaces between tasks rather than allowing an attacker unfettered access to memory. Further, our analysis assumes that state from one execution of a task does not carry over to subsequent executions. How do we use secure memory regions, e.g., the shadow stack, to safely and efficiently persist that state? Finally, there is potential for greater optimization through careful design by the application developer. For instance, if the developer designs the tasks such that external input is always handled in low priority tasks then the process of task pushing is greatly simplified.

5 Conclusions

CFI schemes are only as secure as the CFG is precise [41, 10, 17, 15]. There are two sources of imprecision: the difficulty of sound and complete CFG generation and the labeling scheme extracted from the CFG. Sound and complete CFG generation is believed to be undecidable [10, 17], so to preserve functionality of programs, CFI schemes often use a more permissive CFG, potentially allowing some unintended indirect branch targets. On top of the inherent imprecision, the labeling scheme itself often introduces more imprecision for performance reasons. For example, coarse-grained approaches assign a single label to all legal targets. This imprecision can allow an attacker to achieve Turing-complete computation in the presence of certain instruction sequences [10, 17]. RECFISH mitigates these attacks by using fine-grained labeling and a shadow stack to increase precision.

One potential limitation of RECFISH is application-specific uses of privileged mode execution in tasks, i.e., privileged code that is written by the developer and is not part of RECFISH. In practice, this issue is unlikely to become a barrier to adoption. First, it is uncommon for tasks themselves to have privileged sections (outside of handling hardware interrupts). In the evaluated benchmarks, privileged code was limited to the RECFISH and FreeRTOS code. Second, privileged code in tasks may not be an issue as long as that code omits MPU-sensitive instructions. Specifically, the MPU in Cortex-R can only be modified in privileged mode using the `mcr` and `mrc` instructions. As long as those two instructions only appear in RECFISH code – this is statically verifiable – then RECFISH can rely on its own CFI checks to prevent MPU instructions from being executed outside of normal control flow and, consequently, prevent unwanted modification of the MPU. The proposed ARMv8-R architecture provides another mechanism to address this issue with its bare metal hypervisor mode, but these processors are not widely available yet, and existing systems with ARMv7-R processors will likely not be upgraded.

In summary, RECFISH can be applied to both baremetal and FreeRTOS applications. The defense introduces a minimal amount of program storage and RAM overhead, requiring only 10 bytes of program storage per indirect branch and just 8 bytes per shadow stack operation, and a constant, configurable block of memory for the shadow stacks. Further, in the 85% of task systems where RECFISH can be applied without compromising schedulability, there is no impact on real-time performance.

While this work makes a significant step towards hardening real-time embedded systems, there are many directions for future work. Beyond optimization, future work would benefit from the use of formal methods to analyze the correctness of the CFI instrumentation. Finally, new features in the upcoming ARMv8-R architecture could be leveraged to provide stronger performance and security guarantees for RECFISH as well as other embedded system hardening techniques.

References

1 The Capstone Disassembly Engine. `http://www.capstone-engine.org/`.

2 The Keystone Assembler. `http://www.keystone-engine.org/`.

3 pyelftools. `https://github.com/eliben/pyelftools`.

4 FreeRTOS FAQ relating to memory management and usage. `http://www.freertos.org/FAQMem.html`, 2017. Accessed: 2017-03-28.

5 Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4, 2009.

6 ARM Limited. Cortex-R4 and Cortex-R4F Technical Reference Manual, 2011.

7 Michael Backes and Stefan Nürnberger. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *23rd USENIX Security Symposium*, 2014.

8 B. Brandenburg. *Scheduling and Locking Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

9 Nathan Burow, Scott A Carr, Stefan Brunthaler, Mathias Payer, Joseph Nash, Per Larsen, and Michael Franz. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys*, 50(1), 2017.

10 Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium*, 2015.

11 Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *21st International Conference on Distributed Computing Systems(ICDCS)*. IEEE, 2001.

12 Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting Bare-metal Embedded Systems With Privilege Overlays. In *IEEE Symposium on Security and Privacy*, 2017.

13 Marc L. Corliss, E. Christopher Lewis, and Amir Roth. Using DISE to Protect Return Addresses from Attack. *SIGARCH Computer Architecture News*, 2005.

14 Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Battie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptative Detection and Prevention of Buffer-Overflow Attacks. In *7th USENIX Security Symposium*, 1998.

15 Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium*, 2014.

16 Richard Earnshaw. Procedure call standard for the ARM architecture. *ARM Limited, October*, 2003.

17 Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

18 Shay Gal-On and Markus Levy. Exploring CoreMark—A benchmark maximizing simplicity and efficacy. *The Embedded Microprocessor Benchmark Consortium*, 2012.

19 Jacob Grycel and Robert J. Walls. A Random Number Generator Built from Repurposed Hardware in Embedded Systems. *CoRR*, abs/1903.09365, 2019. `arXiv:1903.09365`.

20 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks: Past, present and future. In *OASIcs-OpenAccess Series in Informatics*, volume 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

21 Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE International Workshop on Workload Characterization*. IEEE, 2001.

22 Monowar Hasan, Sibin Mohan, Rakesh Bobba, and Rodolfo Pellizzoni. Exploring opportunistic execution for integrating security in legacy hard real-time systems. In *37th IEEE Real-Time Systems Symposium*, RTSS, 2016.

23 J. Hiser, A. Nguyen, M. Co, M. Hall, and J.W. Davidson. ILR: Where'd my gadgets go. In *IEEE Symposium on Security and Privacy*, 2012.

24 T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz. Compiler-Generated Software Diversity. In *Moving Target Defense*, Advances in Information Security. Springer, 2011.

25 Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2014.

26 Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated software diversity. In *35th IEEE Symposium on Security and Privacy*, S&P, 2014.

27 J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm, Exact characterization and average case behavior. In *1989 IEEE Real-Time Systems Symposium (RTSS'89)*, December 1989.

28 C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973.

29 Sibin Mohan, Man-ki Yoon, Rodolfo Pellizzoni, and Rakesh Bobba. Real-time systems security through scheduler constraints. In *26th Euromicro Conference on Real-Time Systems*, ECRTS, 2014.

30 Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *ACM Sigplan Notices*, PLDI, 2009.

31 Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. CETS: compiler enforced temporal safety for C. In *ACM Sigplan Notices*, 2010.

32 Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan Notices*, volume 42 (6), 2007.

33 Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

34 James Pallister, Simon Hollis, and Jeremy Bennett. BEEBS: Open benchmarks for energy measurements on embedded platforms. *arXiv preprint*, 2013. `arXiv:1308.5174`.

35 Rodolfo Pellizzoni, Neda Paryab, Man-Ki Yoon, Stanley Bak, Sibin Mohan, and Rakesh Bobba. A generalized model for preventing information leakage in hard real-time systems. In *21st Real-Time and Embedded Technology and Applications Symposium*, RTAS, 2015.

36 Danbing Seto, John P Lehoczky, Lui Sha, and Kang G Shin. On task schedulability in real-time control systems. In *17th IEEE Real-Time Systems Symposium*, 1996.

37 Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *14th ACM conference on Computer and communications security*. ACM, 2007.

38 Bryan Ward, Abhilash Thekkilakattil, and James Anderson. Optimizing Preemption-Overhead Accounting in Multiprocessor Real-Time Systems. In *22nd International Conference on Real-Time and Network Systems*, RTNS, 2014.

39 Man-Ki Yoon, Sibin Mohan, Chien-Ying Chen, and Liu Sha. TaskShuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems. In *22nd Real-Time embedded Technology and Applications Symposium*, RTAS, 2016.

40 Tom Zanussi. microYocto and the internet of tiny. Embedded Linux Conference, 2015.

41 Mingwei Zhang and R Sekar. Control Flow Integrity for COTS Binaries. In *USENIX Security Symposium*, volume 13, 2013.

42 Vojin Zivojnovic, Harald Schraut, M Willems, and R Schoenen. DSPs, GPPs, and multimedia applications-an evaluation using dspstone. In *International Conference on Signal Processing Applications and Technology*, 1995.

A ARM Indirect Jumps

The following table lists all indirect branch operations in ARM. All such branches much be instrumented to enforce forward-edge control flow integrity.

Table 2 Indirect jump operations in ARM.

Mnemonic	Instruction	Description
`bx` *`Rm`*	Branch and exchange	Branch to target address *Rm*, and exchange instruction set based on least significant bit (LSB) of *Rm*. If LSB is set, switch to Thumb mode, else switch to ARM mode.
`blx` *`Rm`*	Branch, link, and exchange	Branch to target address *Rm*, set link register, and exchange instruction set based on LSB of *Rm*.
`ldm{`*`mode`*`}` *`Rm{!}, reglist`*	Load multiple	Load into registers in *reglist*, starting at address in *Rm*. If *Rm!* is specified, write back the final address into *Rm*. Mode specifies the addressing order: *ia* (increment after), *ib* (increment before), *da* (decrement after), *db* (decrement before). The pseudo instruction `ldmfd` is for loading from a full-descending stack. It is the same as `ldmia`.
`pop` *`reglist`*	Pop from stack	Same as `ldmfd sp!,` *`reglist`*
`rfe` *`Rn{!}`*	Return from exception	Pop PC and CPSR off of the stack pointer specified by *Rn* to return from an exception state. If *Rn!* is specified, write back new stack top to *Rn*.

B Selected Source Code

The following source code details the instrumentation used by RECFISH to handle shadow stack operations. Note, this code will jump into the higher privilege mode needed to access shadow stack memory.

Listing 7 Supervisor call handler.

```
do_syscall:
  cpsie   aif                       # Re-enable interrupts
  stmfd   sp!, {r9,r10,r12,lr}      # Store registers
  mrs     r9, spsr                  # Working register
  tst     r9, 0x20                  # Test if thumb state
  ldrneh  r9, [lr, -2]              # Yes: load halfword
  bicne   r9, r9, 0xFF00            #   and get func num
  ldreq   r9, [lr, -4]              # No:  load word and
  biceq   r9, r9, 0xFF000000        #   and get func num
  ldr     r10, table                # Load address of table
  ldr     pc, [r10, r9, lsl 2]      # Jump to routine

table:
  .word jump_table

jump_table:
  .word ss_push
  .word ss_pop
```

Listing 8 Shadow stack operations.

```
# Input: User lr containing value to push to shadow stack
# Returns: void
.type ss_push, ss_push:
        ldr r9, current_ss_const  # Load stack pointer
        ldr r10, [r9]             # Load stack top
        stmfd r10!, {lr}^         # Push lr to stack
        str r10, [r9]             # Store new top
        exit_syscall              # Syscall exit macro

# Input: void
# Returns: value at top of shadow stack -> lr
.type ss_pop, ss_pop:
        ldr r9, current_ss_const  # Load stack pointer
        ldr r10, [r9]             # Load stack top
        ldmfd r10!, {lr}^         # Pop into user mode lr
        str r10, [r9]             # Store new stack top
        exit_syscall              # Syscall exit macro

# Constant pointer reference to current_ss
current_ss_const    .word   current_ss
```

Simultaneous Multithreading Applied to Real Time

Sims Hill Osborne
University of North Carolina, Chapel Hill, North Carolina, USA
`http://shosborn.web.unc.edu`
shosborn@cs.unc.edu

Joshua J. Bakita
University of North Carolina, Chapel Hill, North Carolina, USA
`https://jbakita.me/`
jbakita@cs.unc.edu

James H. Anderson
University of North Carolina, Chapel Hill, North Carolina, USA
`http://jamesanderson.web.unc.edu/`
anderson@cs.unc.edu

Abstract

Existing models used in real-time scheduling are inadequate to take advantage of simultaneous multithreading (SMT), which has been shown to improve performance in many areas of computing, but has seen little application to real-time systems. The SMART task model, which allows for combining SMT and real time by accounting for the variable task execution costs caused by SMT, is introduced, along with methods and conditions for scheduling SMT tasks under global earliest-deadline-first scheduling. The benefits of using SMT are demonstrated through a large-scale schedulability study in which we show that task systems with utilizations 30% larger than what would be schedulable without SMT can be correctly scheduled.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Real-time system specification; Software and its engineering → Scheduling; Software and its engineering → Multithreading

Keywords and phrases real-time systems, simultaneous multithreading, soft real-time, scheduling algorithms

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.3

Related Version Longer version with all graphs at `http://jamesanderson.web.unc.edu/papers/`.

Supplement Material ECRTS 2019 Artifact Evaluation approved artifact available at `https://dx.doi.org/10.4230/DARTS.5.1.8`

Funding Work supported by NSF grants CNS 1409175, CNS 1563845, CNS 1717589, and CPS 1837337, ARO grant W911NF-17-1-0294, and funding from General Motors.

1 Introduction

Simultaneous multithreading (SMT) is a technology developed in the 1980s and 90s that allows multiple processes to issue instructions to different processor contexts, or threads, on a single physical computing core, creating the illusion of multiple cores for every one core that is actually present. It was designed to increase system utilization, particularly in the presence of memory latency [6, 26]. SMT became widely available in 2002, when it was made available on Intel processors [18]. Early experiments on the Pentium 4 showed that SMT could increase throughput by a factor of more than 1.5 in the best case [1, 2, 25]. The first attempt to utilize SMT in a real-time context was made in 2002 by Jain et al. [15], who showed that, by enabling SMT and making every thread available for real-time work, it is possible to schedule workloads with total utilizations up to 50 percent greater than what would be possible on the same platform without SMT. While Jain et al. gave ample experimental evidence that SMT can enable systems with higher utilization to be supported, neither they nor anyone else, to our knowledge, has provided a schedulability test that takes SMT into account.

Unfortunately, SMT's increase in throughput comes at the cost of longer and less predictable execution times, caused by contention for limited hardware resources. Apparently, the real-time systems community decided that this uncertainty makes SMT inappropriate for real-time work. We question the validity of this assessment for soft real-time (SRT) systems that may tolerate some tardiness. Evidence suggest that even others begin to question this assessment in the context of safety-critical domains. In particular, the U.S. Federal Aviation Administration has received requests to certify safety-critical applications that use SMT, though they currently lack adequate techniques for doing so [21]. (We defer considerations of safety-critical applications to future work.)

As evidence of the potential benefits of SMT, we present a sample of our results in Fig. 1; a platform with 16 cores is capable of scheduling task systems with total utilizations exceeding 20. We discuss this graph and others in Section 5.

Considered problem. We consider the problem of defining a scheduler for SRT systems that reaps the benefits of SMT without sacrificing execution-cost predictability. Existing models for analyzing real-time workloads do not allow us to specify how enabling SMT affects a task, so to quantify the per-task effects of SMT, we introduce a new task model, SMART (**S**imultaneous **M**ultithreading **A**pplied to **R**eal **T**ime). Using the SMART model, we attack our problem by dividing it into three sub-problems:

- **Sub-Problem 1:** Determine execution costs for tasks with SMT enabled. "Costs" is plural for each task; one worst-case execution cost is not enough to define a task.
- **Sub-Problem 2:** Decide which tasks should use SMT. How using SMT will affect any given task is a function of what other tasks are using SMT.
- **Sub-Problem 3:** Schedule so tasks using SMT do not interfere with tasks not using SMT.

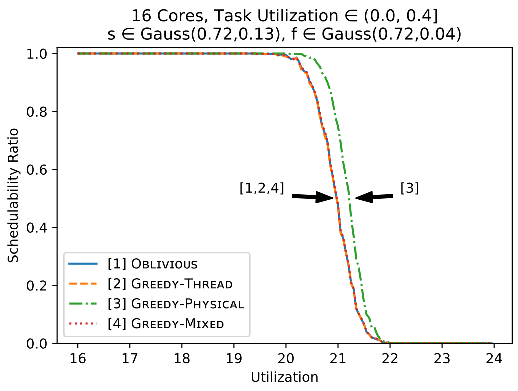


Figure 1 Schedulability on 16 cores with SMT. Note that the horizontal axis begins at utilization 16 and that schedulability does not begin to drop until utilization 20. Effectively, more than 20 cores worth of capacity can be had on a 16-core platform. We discuss this graph and others like it in detail later.

The second sub-problem is particularly interesting. In general, allowing a task to execute with SMT will decrease the demand the task places on the hardware platform but increase the time needed for the task to execute. To address our problem, we need to balance the advantages of decreasing platform demand with the disadvantages of increasing task execution time. It is not enough to evaluate a task in isolation; every task that uses SMT may influence every other task that uses SMT.

Motivation. Processors are expensive. For any workload, real time or not, it is desirable to minimize the hardware cost needed to obtain a given level of performance. SMT is a means to get the most work out of a given processor. Presently, SMT is widely implemented, meaning there is a high chance that users are paying for SMT even if they are not using it. A better understanding of SMT would allow for better use of existing hardware resources.

Related works. Snavely and Tullsen demonstrated that SMT performance is dependent on which tasks share a core and introduced the term "symbiosis" to describe this concept [24]. We have already mentioned Jain et al.'s work on SMT and real-time scheduling from 2002 [15]. Since then, Cazorla et al. [3], Gomes et al. [12, 13], and Zimmer et al. [28] have proposed ways to eliminate the timing uncertainties associated with SMT by means of detailed control over program execution and, in the case of Zimmer et al., a purpose-built processor, FlexPRET. Cazorla et al. [3] and Lo et al. [17] gave methods to limit real-time work to a small number of threads, leaving the remaining threads to execute only when doing so will not interfere with real-time work. Mische et al. [20] proposed to use SMT to hide context-switch times by using threads to switch task state in and out in the background. Early work on the performance of tasks executed by hardware threads was done by Bulpin [1], Bulpin and Pratt [2], Huang et al. [14], and Tuck and Tulsen [25]. Detailed analysis of Intel's microarchitecture, including the resource constraints that are relevant with SMT, have been performed by Fog [11]. A preliminary version of our paper was presented as a work in progress at RTSS 2018 [22].

Contribution and organization. We introduce the SMART task model, a method for scheduling SMART tasks, and a related schedulability test. While other works focus on modifying hardware to make SMT more predictable, our work allows for SMT-supported real-time work to run on existing hardware and operating systems. We give results of benchmark tests measuring the performance impacts of SMT with regard to execution times. We show, using a schedulability study based on our benchmark results, that it is possible to correctly schedule task systems with utilizations more than 30% greater than what would be schedulable on the same platform without SMT enabled.[1]

The rest of this paper is organized as follows. In Section 2, we give a brief overview of SMT technology, discuss the shortcomings of the sporadic task model with regard to SMT, and introduce the SMART model. In Section 3, we address Sub-Problems 2 and 3, showing how SMT can be used to schedule otherwise unschedulable task systems. In Section 4, we address Sub-Problem 1, how to determine appropriate costs. (Note that we address our sub-problems in reverse order.) In Section 5, we present our schedulability experiments and results. In Section 6, we conclude and suggest future directions for our research.

[1] While Jain et al. [15] were able to schedule systems with up to 50% greater utilization, they define a "correctly scheduled system" as one having a low number of observed deadline misses, whereas we define correctness as all tasks having analytically guaranteed bounded tardiness.

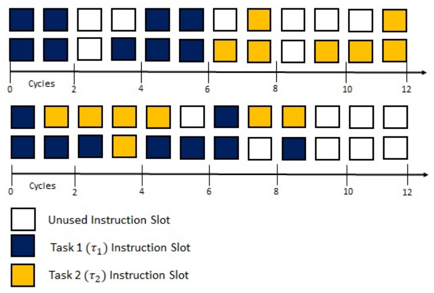

Cycles
0
2
4
6
8
10
12
Cycles
0
2
4
6
8
10
12
Unused Instruction Slot
Task 1 (τ_1) Instruction Slot
Task 2 (τ_2) Instruction Slot

Figure 2 Top: task execution without SMT. Bottom: task execution with SMT.

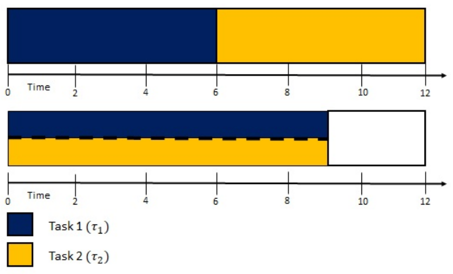


Figure 3 Two tasks executing without SMT (top) and with SMT (bottom). With SMT, each task requires more time to complete individually, but time for both tasks to complete is reduced.

2 What is a SMART Task?

Here we give a brief overview of SMT technology alongside the sporadic task model and its limitations. We introduce SMART as an alternative model to address SMT.

2.1 SMT Basics

Cores with SMT enabled accept multiple instructions per cycle from multiple tasks, reducing wasted instructions per cycle. A detailed explanation is available in Eggers et al.[6], but we illustrate the essentials in Example 1.

▶ **Example 1.** Fig. 2 shows the effect of enabling SMT. At the top of the figure, tasks τ_1 and τ_2 execute sequentially without SMT on a processor that can accept two instructions per cycle. When less than two instructions are available for execution, as at times 2, 3, and elsewhere, processor cycles are lost. τ_1 finishes at time 6 and τ_2 at time 12. At the bottom of the figure, the same tasks execute in parallel with SMT enabled, reducing the number of lost processor cycles. Both tasks finish at time 9. In this case, SMT has the effect of delaying the completion of τ_1, but speeding up the completion of τ_2, since it does not have to wait for τ_1 to complete before beginning its own execution. ◇

Fig. 3 gives a more task-centric view of the two tasks seen in Fig. 2. For the remainder of this paper, we will conceptualize tasks as seen in Fig. 3; we are interested in how long a task takes to execute and how much of a core it uses, not an exact cycle-by-cycle accounting. As shown in Fig. 3, SMT can cause individual tasks to take longer to complete, but total throughput is potentially increased, since the number of wasted instruction slots can be decreased. The challenge for real-time scheduling is to take advantage of this increased throughput without allowing increased execution costs to render the system unschedulable. The effect of SMT on task execution times is not constant across tasks; how much a task's execution time is increased by SMT depends on both the task itself and on other tasks that might be executing on the same core.

To discuss SMT more easily, we make a distinction between a core and a processor. A *core* is the hardware unit responsible for executing instructions. A *processor* is a single instruction context on a core. Every computer core, by definition, supports at least one processor, but computer cores capable of SMT may support multiple processors. We define a *physical processor* as a processor that occupies an entire core, while a *threaded processor* corresponds to a single hardware thread. Different threaded processors on the same core are *sibling processors*. Tasks scheduled on sibling processors are said to be *co-scheduled*.

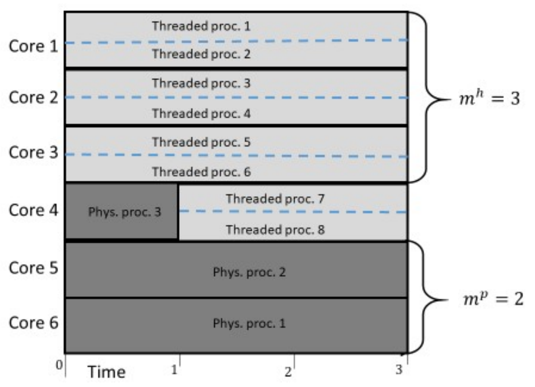


Figure 4 Example of cores supporting threaded processors only, physical processors only, or both.

We focus on a platform π that has m cores where every core supports one physical processor or two threaded processors at a time. For example, Fig. 4 shows a system of six cores. Cores 1-3 have SMT enabled and support two threaded processors each. Cores 5 and 6 have SMT disabled and support one physical processor each. Core 4 initially has SMT disabled and supports one physical processor, but at time 1, SMT is enabled on core 4, causing the single physical processor to be replaced by two threaded processors. We only consider two threads per core because this is what Intel currently supports.

2.2 Task Model

In the traditional implicit-deadline sporadic task model, a task $\tau_i = (T_i, C_i)$ is defined by its *period,* T_i, and its worst-case execution *cost,* C_i. The *utilization* of τ_i is given by $u_i = \frac{C_i}{T_i}$. Every task releases an unlimited number of *jobs*, with the k^{th} job released by τ_i denoted by $\tau_{i,k}$. Jobs of τ_i are released at least T_i units of time apart and have an implicit deadline of T_i. If the jobs of each task τ_i are released exactly T_i units apart, then the task system is *periodic.* We consider only SRT systems here, in which some deadline misses are acceptable. In our model, a job's *tardiness* is the difference between its completion time and deadline, if the job completes after its deadline, and zero otherwise. A task's tardiness is the maximum tardiness of any of its jobs. We define an SRT system as being *correctly scheduled* if all tasks have guaranteed bounded tardiness. A task system is *SRT-schedulable* under scheduling algorithm A if it can be correctly scheduled by the specific algorithm A, and *SRT-feasible* if it is SRT-schedulable by some algorithm A. An algorithm is *SRT-optimal* if it can schedule all SRT-feasible task systems.

Given a platform π consisting of m identical cores and no SMT, a task system τ is SRT-feasible if and only if

$$\forall \tau_i \in \tau \; u_i \leq 1 \qquad \text{and} \qquad \sum_{i=1}^{n} u_i \leq m \tag{1}$$

both hold [4].

The SMART model. The shortcoming of the sporadic model in regard to SMT is that it only allows one worst-case execution cost per task, and therefore cannot adequately characterize a task system's behavior in the presence of SMT. For example, it is not possible to specify the task behavior seen in Fig. 3 using the sporadic model. To address this shortcoming, we introduce the SMART model. In this model, every task is modeled as $\tau_i = (T_i, (C_{i:j}))$. All parameters must be rational. As in the sporadic model, T_i is the period of τ_i. The parameter $(C_{i:j})$ is a list of costs that indicate the worst-case execution cost of a job of τ_i given that the entire job is co-scheduled with one or more jobs of τ_j. We define $C_{i:i}$ to be τ_i's cost when it executes on a normal physical processor. For all $i \neq j$, $C_{i:j} \geq C_{i:i}$.[2] We define $u_{i:j} = \frac{C_{i:j}}{T_i}$.

Notice that we are implicitly making four simplifying assumptions here: **(i)** τ_i's worst-case execution time can be determined by examining how it is interfered with when co-scheduled with each other task *individually*; **(ii)** when τ_i is co-scheduled with τ_j, every portion of τ_i receives the same amount of interference from every portion of τ_j; **(iii)** the two threads of a given core are identical; and **(iv)** the hardware-level priority of τ_i and τ_j, when co-scheduled, is fixed. In practice, (i) and (ii) will not necessarily hold, but we maintain that our model is sufficient for non-safety critical SRT workloads. Currently, (iii) and (iv) hold on Intel architectures [11]. We discuss (i) and (ii) further in Section 4 when we delve into the issue of how to actually determine execution costs.

▶ **Definition 2.** *The execution* rate *of τ_i given that it is co-scheduled with τ_j is given by $r_{i:j} = \frac{C_i}{C_{i:j}}$, where both C_i and $C_{i:j}$ are maximum observed execution times.*

We assume no relationship between $r_{i:j}$ and $r_{j:i}$; in fact, as we show in our benchmark experiments, the two can differ significantly. Our definition assumes two hardware threads per core, but could be expanded to allow for additional threads. In general, $r_{i:j} > 0.5$ indicates that τ_i could benefit from being co-scheduled with τ_j assuming that $C_{i:j} \leq T_i$ and $C_{j:i} \leq T_j$ hold.

▶ **Example 3.** Suppose Fig. 3 depicts one job each of SMART tasks τ_1 and τ_2. $C_1 = 6$ and $C_2 = 6$, but $C_{1:2} = 9$ and $C_{2:1} = 9$, giving $r_{1:2} = r_{2:1} = \frac{2}{3}$. Task τ_2 benefits from SMT; the job completes at time 9 with SMT as opposed to time 12 without. If both jobs are released at time 0 and have a deadline at time 10, then SMT allows for both jobs to complete on time, whereas without SMT, τ_2's job misses its deadline. ◇

Scheduling SMART tasks. We need to schedule n tasks that have n costs each; this problem poses obvious difficulties. In the next section, we show how we can schedule SMART tasks similarly to traditional sporadic tasks without sacrificing the advantages of SMT.

3 Scheduling Physical and Threaded Tasks

Not all tasks will benefit from SMT. We label tasks that should and should not use SMT as *threaded tasks* and *physical tasks*, respectively. Physical tasks can execute only on physical processors and threaded tasks only on threaded processors. To keep the task types separate, we divide them into two task subsystems, τ^p and τ^h, that we schedule separately.

▶ **Definition 4.** *Subsystem τ^p is the set of all physical tasks in τ. $n^p = |\tau^p|$. Subsystem τ^h is the set of all threaded*[3] *tasks in τ. $n^h = |\tau^h|$.*

[2] In the rare event that $C_{i:j} < C_{i:i}$ holds, the two are likely close in value, and we can simply redefine $C_{i:j}$ to equal $C_{i:i}$.
[3] We use h rather than t for t<u>h</u>readed so as to avoid confusion with t for time.

After partitioning τ into τ^p and τ^h, physical tasks have cost C_i^p and utilization $u_i^p = \frac{C_i^p}{T_i}$; threaded tasks have cost C_i^h and utilization $u_i^h = \frac{C_i^h}{T_i}$. Costs for physical tasks are no different than costs in a sporadic task system, but costs for threaded tasks are a function of how the task system is divided. These cost parameters are a simplification of the full SMART parameters; we will show how to obtain them in Section 3.2.

▶ **Definition 5.** *The total utilizations of τ^p and τ^h are given by $U^p = \sum_{i=1}^{n^p} u_i^p$ and $U^h = \sum_{i=1}^{n^h} u_i^h$ respectively. To measure the total demand placed on the platform, we define* effective utilization, *$U^E = U^p + \frac{U^h}{2}$. U^h is halved in the sum to reflect the fact that each threaded task requires only half a core at a time to execute.* ◇

3.1 Sub-Problem 3: Scheduling Task Subsystems

In this section, we give conditions for scheduling τ^p and τ^h on π. We assume the decision of which tasks should be physical and which should be threaded has already been made. Our current problem is how to schedule those tasks, but the best way to do so is not clear.

▶ **Example 6.** Suppose we attempt to schedule a task system τ using global earliest-deadline-first scheduling (GEDF). Let τ_1 be a threaded task and τ_2 a physical task such that at time t, a job of $\tau_1$with a deadline of $t+1$ is contending for a single core with a job of τ_2 with a deadline of $t+2$. Following GEDF, the job of τ_1 should be given priority over that of τ_2. However, if no other threaded task has an active job at time t, then doing so will cause the second threaded processor of a core in π to be unused, negating any advantage gained by having τ_1 be threaded. If we avoid this problem by giving priority to τ_2, then we are not wasting processor capacity, but we are violating EDF priority rules. If we co-schedule the tasks on threaded processors despite τ_2 being a physical task, then unanticipated task interference may ensue, potentially invalidating assigned per-task worst-case execution costs. None of these approaches is particularly satisfactory. ◇

To address the problems raised in Example 6, we divide π into *sub-platforms* π^p and π^h.

▶ **Definition 7.** *π^p is the sub-platform of π that schedules only tasks in τ^p. It includes $m^p = \lfloor U^p \rfloor$ fully available cores and one partially available core. Given a length-W interval, denoted a* window*, the partially available core belongs to π^p for a^pW time units per window, where $a^p = U^p - \lfloor U^p \rfloor$. π^p can exist only if $U^p \le m$.*

▶ **Definition 8.** *π^h is the sub-platform of π that schedules only tasks in τ^h. It has $m^h = m - \lceil U^p \rceil$ fully available cores and one core available for a^hW time units per window, where $a^h = \lceil U^p \rceil - U^p$. Consequently, $m^h + a^h = m - U^p$. If $a^p > 0$, then $a^h = 1 - a^p$.*

We refer to the core shared by both platforms as the *shared core*. If there is no shared core, then $a^p = a^h = 0$. Note that $m^p + a^p + m^h + a^h = m$ must hold.

▶ **Example 9.** In Fig. 4, π^p is shown in dark gray and π^h in light gray. The sub-platforms are defined by $m^p = 2$, $m^h = 3$, $W = 3$, $a^p = \frac{1}{3}$, and $a^h = \frac{2}{3}$. ◇.

We now give schedulability results for τ^p and τ^h individually and then combine those conditions to get an overall schedulability result. For the most part, we will focus on the case where a shared core exists. Our results are based on Devi and Anderson's EDF-high-low (EDF-hl) algorithm [5]. EDF-hl gives schedulability conditions and tardiness bounds for "low" SRT tasks that are scheduled according to GEDF but are subject to interruption from periodic "high" hard real-time tasks, with at most one such task fixed on each processor.

For our purposes, we can view τ^p as a set of low tasks scheduled on $m^p + \lceil a^p \rceil$ processors and subject to preemption by a single high task with period W and cost $a^h W$. This reflects the fact that, from the perspective of τ^p, work on the shared core is periodically preempted. Likewise, we can view τ^h as a set of low tasks scheduled on $2(m^h + 1)$ processors that are periodically preempted by two high tasks, both with period W and cost $a^p W$. The following definitions apply to the EDF-hl results.

▶ **Definition 10.** *Devi and Anderson define τ_H as the set of all high tasks, τ_L as the set of all low tasks, $u_{max}(\tau_L)$ as the highest-utilization task within τ_L, U_{sum} as the total utilization of both τ_H and τ_L, U_H is the sum of all the utilizations of all tasks in τ_H, and U_L is the sum of the $min(\lceil U_{sum} \rceil - 2, n)$ largest utilization of tasks in τ_L.*

We state an abridged version of Theorem 1 in [5] here. The full version defines the tardiness bound B as a function of the task system and platform. We omit that portion of the theorem due to space constraints.

▶ **Theorem 11.** *EDF-hl ensures a tardiness bound of at most B to every task τ_i of τ_L if $|\tau_H| \leq m$ and $U_{sum} \leq m$ and at least one of (2) or (3) holds.*

$$m - |\tau_H| - U_L > 0 \tag{2}$$

$$m - \max(|\tau_H| - 1, 0)u_{max}(\tau_L) - U_L - U_H > 0 \tag{3}$$

Returning to our problem, our schedulabilty conditions rely on the following assumptions. These assumptions allow us to schedule τ^p and τ^h as if they both consisted of standard sporadic tasks. We will show how to support Assumptions 1 and 2 in Section 3.2.

▶ **Assumption 1.** *Tasks have been divided into threaded and physical tasks such that $\forall \tau_i^p \in \tau^p, u_i^p \leq 1$ and $\forall \tau_i^h \in \tau^h, u_i^h \leq 1$ both hold. Without loss of generality, we assume that the tasks in each of the sets τ^p and τ^h are indexed in decreasing-utilization order, e.g., u_1^p (resp., u_1^h) is the largest utilization in τ^p (resp., τ^h).*

▶ **Assumption 2.** *Worst-case costs for physical and threaded tasks have been determined.*

▶ **Assumption 3.** *Physical tasks are not permitted to execute on threaded processors.*[4]

▶ **Lemma 12.** *τ^p is schedulable on π^p under GEDF such that all tasks have guaranteed bounded tardiness if (4) holds.*

$$U^p \leq m^p + a^p. \tag{4}$$

Proof. If $a^p = 0$, then the result restates the SRT feasibility condition for m identical, fully available processors from (1). GEDF is known to be SRT-optimal [4], so the result follows.

If $m^p = 0$, then it can easily be shown that the system is schedulable only if $U^p \leq a^p$.

In the rest of the proof, we consider the remaining possibility, i.e., that $a^p > 0$ and $m^p > 0$ both hold. For this case, we show that Theorem 11 can be applied.

From the perspective of τ^p, there exists a set of low tasks τ^p with total utilization U^p, one high task with utilization a^h, and $m^p + 1$ processors. Thus, we want to apply Theorem 11 with the substitutions $m \leftarrow m^p + 1$, $\tau_L \leftarrow \tau^p$, $U_{sum} \leftarrow U^p + a^h$, and $|\tau_H| = 1$. With

[4] When the shared core belongs to π^p, it supports a physical processor, not a threaded processor.

these substitutions, (4), and Def. 8, it is straightforward to see that both $|\tau_H| \leq m$ and $U_{sum} \leq m$ hold, as required by Theorem 11. We now show that (2) holds, from which bounded tardiness for the tasks in τ_L, i.e., those in τ^p, follows. To see this, note that from Def. 8 and $U_{sum} = U^p + a^h$, we have

$$\begin{aligned}
&U_L = \sum_{i=1}^{min(\lceil U^p+a^h \rceil -2, n^p)} u_i^p \\
&= \{\text{by Defs. 7 and 8, } U^p + a^h = m^p + 1\} \\
&U_L = \sum_{i=1}^{min(m^p-1, n^p)} u_i^p \\
&\Rightarrow \{\text{because } u_i^p \leq 1 \text{ holds, by Assumption 1}\} \\
&U_L < m^p.
\end{aligned}$$

From this inequality, we have $m - |\tau_H| - U_L = m^p + 1 - 1 - U_L > 0$, as required by (2). ◀

The schedulability condition for τ^h is slightly more complicated, due to it potentially having two partially available processors.

▶ **Lemma 13.** *τ^h is schedulable on π^h under GEDF such that all tasks have guaranteed bounded tardiness if (5) and at least one of (6) or (7) hold, where $u_{max}(\tau^h)$ denotes the maximum task utilization in τ^h.*

$$U^h \leq 2(m^h + a^h) \tag{5}$$

$$2m^h > \sum_{i=1}^{\min(2m^h, n^h)} u_i^h \tag{6}$$

$$2(m^h + a^h) - u_{max}(\tau^h) > \sum_{i=1}^{\min(2m^h, n^h)} u_i^h \tag{7}$$

Proof. As in the prior proof, the proof is straightforward if either $a^h = 0$ holds or $m^h = 0$ holds, so we focus on the remaining possibility, i.e, $m^h > 0$ and $a^h > 0$ both hold; note that the latter implies that $a^p > 0$ holds as well. As before, we will use Theorem 11. In this case, we are attempting to schedule a set of low tasks τ^h with total utilization U^h on $2(m^h + 1)$ processors given two high tasks, each with utilization a^p. Thus, we want to apply Theorem 11 with the substitutions $m \leftarrow 2(m^h + 1)$, $\tau_L \leftarrow \tau^h$, $U_{sum} \leftarrow U^h + 2a^p$, and $|\tau_H| = 2$. With these substitutions, (5), and Def. 8, it is straightforward to see that both $|\tau_H| \leq m$ and $U_{sum} \leq m$ hold, as required by Theorem 11. In the rest of the proof, we show that, with these substitutions, (6) implies (2) and (7) implies (3), from which bounded tardiness for the tasks in τ_L, i.e., those in τ^h, follows.

To see that (6) implies (2), first note that, because m^h is an integer, we have $\lceil U_{sum} \rceil - 2 \leq \lceil m \rceil - 2 = \lceil 2(m^h+1) \rceil - 2 = \lceil 2m^h \rceil = 2m^h$. Therefore,

$$2m^h > \sum_{i=1}^{\min(2m^h, n^h)} u_i^h$$

$\Rightarrow$ {because $\lceil U_{sum} \rceil - 2 \leq 2m^h$}

$$2m^h > \sum_{i=1}^{\min(\lceil U_{sum} \rceil - 2, n^h)} u_i^h$$

$=$ {by the definition of U_L in Def. 10}

$$2m^h > U_L,$$

i.e., $2m^h - U_L > 0$ holds, which is equivalent to (2), since $m = 2(m^h+1)$ and $|\tau_H| = 2$.

To see that (7) implies (3), observe that

$$2(m^h + a^h) - u_{max}(\tau^h) > \sum_{i=1}^{\min(2m^h, n^h)} u_i^h$$

$\Rightarrow$ {reasoning as above}

$$2(m^h + a^h) - u_{max}(\tau^h) > U_L$$

$=$ {because $a^h = 1 - a^p$}

$$2(m^h + 1 - a^p) - u_{max}(\tau^h) > U_L,$$

$=$ {in our context $u_{max}(\tau^h) = u_{max}(\tau_L)$, $|\tau_H| - 1 = 2$, $U_H = 2a^p$, and $m = 2(m^h+1)$}

$$m - \max(|\tau_H| - 1, 0) u_{max}(\tau_L) - U_H > U_L,$$

which is equivalent to (3). ◀

A special case applies when there is no shared core.

▶ **Lemma 14.** *If $a^h = 0$, then τ^h is schedulable on π^h under GEDF if and only if $U^h \leq 2m^h$ holds.*

Proof. With no shared core, the platform consists of $2m^h$ identical cores. The standard SRT feasibility test given by (1) applies. ◀

Our next step is to give a schedulability condition for τ^p and τ^h combined on π. This condition is a straightforward extension of the preceding lemmas, but it has the benefit of letting us focus on τ rather than on how π is partitioned.

▶ **Theorem 15.** *Platform π can be partitioned such that τ^p is schedulable on π^p and τ^h is schedulable on π^h, both under GEDF, if (8) and at least one of (9) or (10) hold.*

$$U^E \leq m \tag{8}$$

$$2(m - \lceil U^p \rceil) > \sum_{i=1}^{\min(2(m - \lceil U^p \rceil), n^p)} u_i^h \tag{9}$$

$$2(m - U^p) - u_1^h > \sum_{i=1}^{\min(2(m - \lceil U^p \rceil), n^p)} u_i^h \tag{10}$$

Proof. In order to define m^p and a^p so that $m^p + a^p = U^p$ holds, as in Def. 7, we merely require $U^p \leq m$ to hold, and by Def. 5, this is implied by (8). Note that $m^p + a^p = U^p$ satisfies Condition (4) in Lemma 12.

Schedulability of τ^p on π^p is implied by (8):

$$\begin{aligned} &U^E \leq m \\ &\quad = \{\text{by Def. 5, } U^E = U^p + \frac{U^h}{2}\} \\ &U^p \leq m \\ &\quad = \{\text{by Def. 7, } m^p + a^p = U^p\} \\ &U^p = m^p + a^p, \end{aligned}$$

which is the condition for τ^p per Lemma 12.

We next show that (8) implies Condition (5) of Lemma 13. To see this, observe that, by Def. 5, $U^E \leq m \Rightarrow \frac{U^h}{2} \leq m - U^p$. Also, by Def. 8, $m^h + a^h = m - U^p$. Putting these facts together, we have $U^h \leq 2(m^h + a^h)$, which is (5).

We conclude the proof by showing that (9) is equivalent to Condition (6) of Lemma 13, and that (9) is equivalent to Condition (7) of Lemma 13. To see the former, note the following.

$$\begin{aligned} 2(m - \lceil U^p \rceil) > &\sum_{i=1}^{\min(2(m-\lceil U^p \rceil), n^h)} u_i^h \\ = &\{\text{by Def. 8, } m - \lceil U^p \rceil = m^h\} \\ 2m^h > &\sum_{i=1}^{\min(2m^h, n^h)} u_i^h \end{aligned}$$

Similarly, to see that (10) holds, note the following.

$$\begin{aligned} 2(m - U^p) - u_1^h > &\sum_{i=1}^{\min(2(m-\lceil U^p \rceil), n^h)} u_i^h \\ = &\{\text{by Def. 8, } m - \lceil U^p \rceil = m^h.\} \\ 2(m^h + a^h) - u_1^h > &\sum_{i=1}^{\min(2m^h, n^h)} u_i^h. \end{aligned}$$

Having verified all conditions of Lemmas 12 and 13, we conclude that τ^p is schedulable on π^p and τ^h is schedulable on π^h. ◀

Again, a special case applies if U^p is integral.

▶ **Corollary 16.** *If U^p is integral, then both τ^p and τ^h are schedulable on their respective sub-platforms under GEDF so long as $U^E \leq m$ holds.*

Proof. Similar to the proof of Lemma 14. ◀

It is not strictly necessary that π^p be defined as we do here. If we allow other design considerations, such as maximizing cache affinity or minimizing tardiness, different platform definitions may be preferable, but we defer those possibilities to future work.

By themselves, the results of this section are not very useful, since there are an exponential number of possible ways to partition π. In the next section, we show how to efficiently find τ^p and τ^h that will be schedulable under Theorem 15.

3.2 Sub-Problem 2: Dividing the Tasks

We have addressed how to schedule a task system τ for a given pair of subsystems τ^p and τ^h. Here, we show how we arrive at Assumption 1 – τ has already been divided – and weaken Assumption 2, which states that all execution costs have been determined, to the following:

▶ **Assumption 4.** *If τ_i is a threaded task, then $C_i^h = \max_{\forall \tau_j \in \tau} C_{i:j}$.*

Oblivious scheduling. We first work through a simple example of dividing a task system and then formalize that approach into what we term *symbiosis-oblivious partitioning.*[5] We then show how our approach can be improved by modifying Assumption 4.

▶ **Example 17.** Let τ consist of four SMART tasks,

$$\tau_1 = \left(8, \left(7, 10, 10, 9.\overline{3}\right)\right), \qquad \tau_2 = \left(4, \left(4, 1, 2, 1.\overline{3}\right)\right),$$
$$\tau_3 = \left(4, \left(3, 2.\overline{6}, 2, 2.5\right)\right), \qquad \tau_4 = \left(8, \left(6, 6, 5.\overline{3}, 4\right)\right).$$

Under traditional sporadic scheduling, where we consider only physical costs, τ has total utilization $\frac{7}{8} + \frac{1}{4} + \frac{2}{4} + \frac{4}{8} = 2.125$ and will require three cores to be feasibly scheduled (recall that $C_{i:i}$ gives τ_i's cost with nothing co-scheduled, i.e., without SMT). Based on Assumption 4, we see that $C_1^h = 10$ if τ_1 is threaded. Because $T_1 = 8$, making τ_1 threaded would give $u_1^h = \frac{10}{8}$, making the system unschedulable. For τ_2, C_2^h would be at most τ^2's period, but $C_2^h = 4$ would be more than twice $C_2^p = 1$. Part of the schedulability condition given in Theorem 15 is that $U^E \leq m$. Because U^E is defined as $U^E = U^p + \frac{U^h}{2}$ (Def. 5), placing τ_2 in τ^h would increase U^E more than placing τ_2 in τ^p, so we do not wish for τ_2 to be threaded. For both τ_3 and τ_4, $\max C_{i:j} \leq T_i$ and $\min(\frac{C_i}{C_{i:j}}) \geq .5$ both hold, so letting those tasks be threaded would decrease U^E compared to placing them in τ^p without violating $u_i^h \leq 1$, so we allow those tasks to be threaded, giving $u_3^h = \frac{3}{4}$ and $u_4^h = \frac{6}{8}$. The resulting partition has $U^p = \frac{7}{8} + \frac{1}{4}$, $U^h = \frac{3}{4} + \frac{6}{8}$, and $U^E = 1.875$. It can, per Theorem 15, be scheduled on only two cores. ◇

We formally state the steps we just took in Algorithm 1, which partitions τ into τ^p and τ^h so as to minimizes U^E subject to $u_i^h \leq 1$ for all threaded tasks and $|\tau^h| \geq 2$. The resulting partition is then schedulable if Theorem 15 holds. We require that $|\tau^h| \geq 2$ holds since allowing a single threaded task will give no schedulability advantage compared to letting all tasks by physical. We refer to partitions of τ that obey both these constraints as *legal.* We will examine the effectiveness of Algorithm 1 in our schedulability study.

▶ **Definition 18.** *A partition of τ into τ^p and τ^h is* legal *if and only if $\forall\ \tau_i^h \in \tau^h$, $u_i^h \leq 1$ and $|\tau^h| \neq 1$ hold.*

A more complex cost model. Under Assumption 4, the only variable that influences the cost of τ_i is whether τ_i is physical or threaded. However, Assumption 4, and consequently Algorithm 1, is highly pessimistic with regard to assigning C_i^h values. Returning to Example 17, we declared $C_3^h = 3$ on the grounds that $\forall j$, $\max C_{3:j} = 3$ holds. However, there is a limitation to that logic; $C_3^h = 3$ is based on the assumption that τ_1 can interfere with τ_3, but in our example, we decided that τ_1 should not be threaded. We can remove this limitation, thereby improving our model, by replacing Assumption 4 with Assumption 5. The difference is that under Assumption 5, C_i^h is only based on other threaded tasks, not on all tasks in τ.

[5] The terms symbiosis-oblivious and symbiosis-aware scheduling were previously used by Jain et al. [15].

Algorithm 1: Oblivious Partitioning.

```
1: for all $\tau_i \in \tau$ do
2:     $C_i^h \leftarrow \max_{\forall j \leq n} C_{i:j}$
3:     if $C_i^h \leq T_i$ and $\frac{C_i}{C_i^h} \geq 2$ then
4:         $\tau^h \leftarrow \tau^h \cup \tau_i$
5:     else
6:         $C_i^p \leftarrow C_{i:i}$
7:         $\tau^p \leftarrow \tau^p \cup \tau_i$
8:     end if
9: end for
10: if $|\tau^h| < 2$ then
11:     $\tau^p \leftarrow \tau^p \cup \tau^h$
12:     $\tau^h \leftarrow \emptyset$
13: end if
14: return $\tau^p, \tau^h$
```

▶ **Assumption 5.** *If τ_i is threaded, then $C_i^h = \max_{\forall \tau_j \in \tau^h} C_{i:j}$.*

The difference is that while Assumption 4 considers interference from all tasks in τ, Assumption 5 considers interference only from other tasks in τ^h. While this approach removes some of the pessimism present in symbiosis-oblivious scheduling, it has the disadvantage that every time a task is added to or removed from τ^h, C_i^h may change for all tasks in τ^h. We refer to task-partitioning algorithms that incorporate Assumption 5 as *symbiosis-aware partitioning.* We give a brief demonstration of symbiosis-aware partitioning in Example 19, using the same task set as in Example 17.

▶ **Example 19.** We first decide that τ_1 must be physical, since $\forall\ j \neq 1, C_{1:j} > T_1$. Knowing that no task will be co-scheduled with τ_1, we have $C_2^h = 2$ and $C_3^h = 2.\overline{6}$, giving $u_2^h = \frac{2}{4}$ and $u_3^h = \frac{2.\overline{6}}{4}$, but leaving u_4^h unchanged. (In Example 17, we made τ_2 a physical task and τ_3 a threaded task with $u_3^h = \frac{3}{4}$.) Now we make all of τ_2, τ_3, and τ_4 threaded, with τ_3 having a lower utilization than before. We now get $U^p = \frac{7}{8}$ and $U^h = \frac{2}{4} + \frac{2.\overline{6}}{4} + \frac{6}{8}$, so that $U^E = 1.8\overline{3}$. a reduction from $U^E = 1.875$ in Example 17. Again, τ^p and τ^h are schedulable on two cores per Theorem 15.

A greedy approach to schedulability. We propose to use Algorithm 2 to partition τ. The algorithm seeks to minimize U^E by repeatedly moving a task from τ^p to τ^h, or vice versa, to give the greatest decrease in U^E. It does so until either a specified maximum number of attempts has been made or it reaches a partition that cannot be improved by the movement of any single task. The algorithm is not optimal, even given an unlimited number of attempts, as there may exist partitions of τ that cannot be improved by moving any one task but can be improved by moving two or more tasks.

The **for** loop of lines 3 through 16 determines, for every τ_i in τ^p, the benefit of moving that task to τ^h. Line 4 tests what C_i^h would be if τ_i were in τ^h. Lines 10 through 13 calculate the change to tasks already in τ^h caused by moving τ_i, and line 15 gives the total change to U^E caused by moving τ_i to τ^h.

Similarly, the **for** loop of lines 19 through 23 determines the benefit of moving τ_j to τ^p, for every τ_j currently in τ^h. Line 20 gives the change to tasks remaining in τ^h caused by moving τ_j, and line 22 gives the total change to U^E caused by moving τ_j to τ^p. The **if** of line 25 guarantees that no task will be moved unless moving that task will decrease U^E, preventing the algorithm from placing τ into any one partition more than once.

Algorithm 2: Greedy Partitioning.

```
Require: $\tau$ partitioned such that $\forall \tau_i \in \tau^h, u_i^h \leq 1$ and $|\tau^h| \geq 2$
 1: for $\ell \leftarrow 1...maxLoops$ do
 2:   ▷ Identify best move from $\tau^p$ to $\tau^h$
 3:   for all $\tau_i \in \tau^p$ do
 4:     $C_i^h = \max_{\tau_i \in \tau^h} C_{i:j}$
 5:     $u_i^h = \frac{C_i^h}{T_i}$
 6:     if $u_i^h > 1$ then
 7:       continue
 8:     end if
 9:     ▷ Calculate how adding $\tau_i$ to $\tau^h$ will affect tasks already in $\tau_h$
10:     if moving $\tau_i$ to $\tau^h$ will cause $u_j^h \geq 1$ for any $\tau_j \in \tau^h$ then
11:       continue
12:     end if
13:     $I(\tau_i^h) \leftarrow$ total increase in util. of tasks already in $\tau^h$ caused by moving $\tau_i$
14:     ▷ $\Delta(i)$ gives decrease to $U^E$ caused by moving $\tau_i$.
15:     $\Delta(i) \leftarrow u_i^p - \frac{u_i^h + I(\tau_i^h)}{2}$
16:   end for
17:   ▷ Identify best move from $\tau^h$ to $\tau^p$
18:   if $|\tau^h| > 2$ then
19:     for all $\tau_j \in \tau_h$ do
20:       $D(\tau_j^h) \leftarrow$ total decrease in util. of tasks already in $\tau^h$ caused by moving $\tau_j$
21:       ▷ $\Delta(j)$ gives decrease to $U^E$ caused by moving $\tau_j$.
22:       $\Delta(j) \leftarrow \frac{u_j^h + D(\tau_j^h)}{2} - u_j^p$
23:     end for
24:   end if
25:   if no task has a positive $\Delta$ value then
26:     break
27:   end if
28:   Move task with maximum $\Delta$ to other subsystem and update threaded costs
29: end for
30: return($\tau^p, \tau^h$)
```

The algorithm returns a partition that can be tested for schedulability by Theorem 15.

Algorithm 2 assumes, and maintains as an invariant, that the partition is legal, as defined in Def. 18. To begin Algorithm 2, τ must already be in a legal partition. We propose three ways to achieve this. First, in the *greedy-threaded* approach, we begin with all tasks in τ^h and then place into τ^p all tasks for which any possible C_i^h value will give $u_i^h > 1$. Intuitively, putting tasks in τ^h whenever possible should be beneficial, so we should start with as many tasks in τ^h as possible.

Second, in the *greedy-physical* approach, we start with all tasks in τ^p apart from the single pair of tasks that will give the greatest decrease to U^E. This can be done by defining the decrease to U^E associated with a single pair of tasks (τ_i, τ_j) as

$$\forall\ (i,j), \Delta(i,j) = u_i^p + u_j^p - \frac{1}{2}\left(\frac{C_{i:j}}{T_i} + \frac{C_{j:i}}{T_j}\right)$$

and adding to τ^h the pair of tasks that maximize $\Delta(i, j)$ subject to $\frac{C_{i:j}}{T_i} \leq 1$ and $\frac{C_{j:i}}{T_j} \leq 1$. When τ_i and τ_j are placed into τ^h, u_i^p and u_j^p are no longer part of U^p and can be subtracted from U^E. However, we must add half of the new U^h value, $\left(\frac{C_{i:j}}{T_i} + \frac{C_{j:i}}{T_j}\right)$, to U^E. We expect this approach will be more efficient than the first one in task systems where u_i^p is typically large or $\frac{C_i}{C_{i:j}}$ is typically small, since there will be relatively few tasks that can be placed in τ^h, making it more efficient to begin with the majority of tasks in τ^p. If no satisfactory pair of tasks exists, then we conclude that SMT should not be used with this task system.

Third, in the *greedy-mixed* approach, we first run Algorithm 1 and use the partition given by doing so as our starting point. Intuitively, Algorithm 1 by itself should give a partition with a lower U^E value than either of the other two approaches, so using it is a starting point should yield better results. As with the greedy-physical approach, if Algorithm 1 places no tasks in τ^h, then we conclude that SMT should not be used. We compare these three approaches in our schedulability experiments, presented in Section 5. We found that for all three versions of Algorithm 2, there existed task systems that were schedulable according to that version alone. In fact, the greedy-physical approach seemed to find more schedulable task systems than the other two.

4 Sub-Problem 1: SMT and Execution Times

Current literature does not address how SMT affects worst-case execution costs. While the early 2000s saw multiple detailed analyses of the performance effects of SMT [1, 2, 25], little work of this type has been done since then. While ongoing research into scheduling with SMT exists outside of real time [7, 8, 10, 23], this current research does not suit our needs for two reasons. First, it tends to be oriented towards total throughput and average execution costs, whereas we need information on worst-case execution costs. Second, the current works we are aware of compare different methods of implementing SMT, but do not compare systems that use SMT to those that do not use it.

4.1 Benchmark Experiments

To analyze the effects of SMT on worst-case execution costs, we ran a series of experiments using the TACLeBench sequential benchmarks [9], which consist of 23 C implementations of functions commonly found in embedded and real-time systems. All of our experiments were conducted in Linux on an Intel Xeon Silver 4110 2.1 GHz CPU with eight cores, each capable of supporting two threaded processors, running Linux.[6]

To get baseline results for execution times without SMT enabled, we looped each benchmark 1,000 to 100,000 times – lower cost benchmarks got more loops – and timed the execution of each loop using a nanosecond resolution timer. Between loops, an array the size of the L3 cache was allocated and set, so that every execution started with a cold cache. Benchmarks were assigned a Linux real-time priority, prioritizing them above all normal tasks, pinned to a single processor, and executed sequentially. We excluded four benchmarks from the set – anagram, audiobeam, g723_enc, and huff_dec – as they would not correctly execute in a loop. Results of our baseline experiments are summarized in Table 1. The last column gives the coefficient of variation, defined as the standard deviation divided by the mean.

[6] Code for these experiments is available at `https://github.com/JoshuaJB/SMART-ECRTS19`, `https://jamesanderson.web.unc.edu/papers/`, and `https://doi.org/10.4230/DARTS.5.1.8`.

Table 1 Baseline Execution Times (ns).

Benchmark	max	mean	$CV \left(\frac{\text{std. dev.}}{\text{mean}}\right)$
adpcm_dec	167,380	151,914	0.006659
adpcm_enc	158,053	147,394	0.006463
ammunition	47,979,870	47,899,553	0.001589
cjpeg_transupp	844,791	827,661	0.002087
cjpeg_wrbmp	32,420	26,712	0.010552
dijkstra	15,740,782	15,719,309	0.000445
epic	665,837	649,170	0.002284
fmref	154,776	99,280	0.068863
gsm_dec	470,193	463,592	0.002546
gsm_enc	1,337,465	1,320,787	0.001934
h264_dec	93,361	82,045	0.006340
huff_enc	247,232	234,213	0.005431
mpeg2	135,009,849	134,898,300	0.000248
ndes	21,600	15,426	0.015071
petrinet	3,682	62	0.215268
rijndael_dec	965,022	940,081	0.007688
rijndael_enc	872,400	858,645	0.002224
statemate	11,928	6,495	0.026602
susan	10,958,260	10,932,188	0.000379

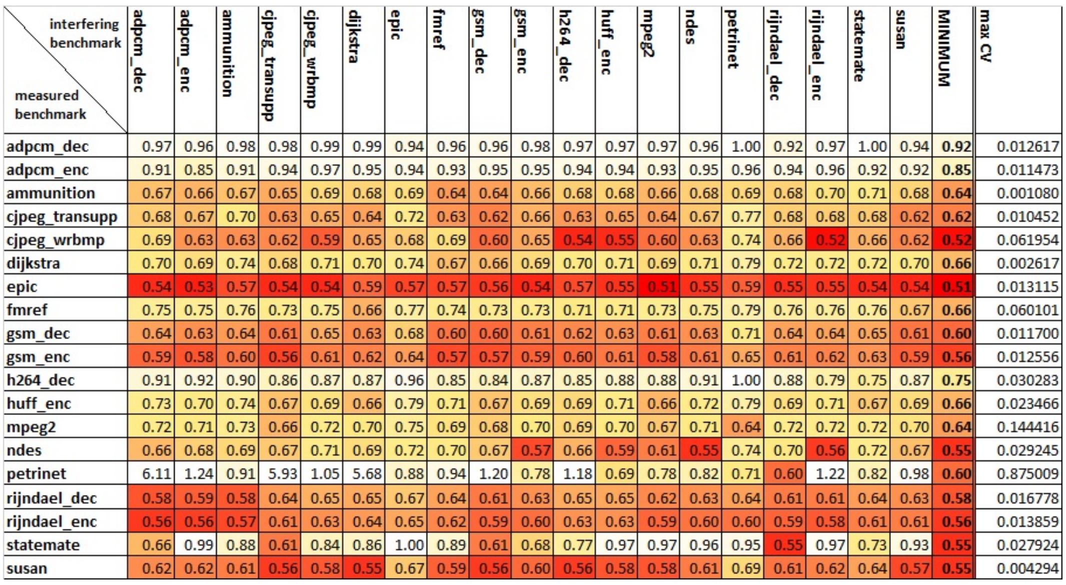
For threaded execution times, every task was executed alongside every other task. For each pair, the measured task was executed the same number of times as in the baseline experiments while an interfering task executed continuously at equal priority on the second thread of the same core. Our results are summarized in Fig. 5, which shows $r_{i:j}$ for every pair of tasks, with the measured task as τ_i and the interfering task as τ_j. Observed rates ranged from 0.51 (mpeg2 interfering with epic) to 1.00, with the exception of values involving petrinet. Petrinet has an extremly short execution time, as indicated in Table 1; we suspect its strange behavior is merely random noise.

We cannot guarantee that our experiments captured the maximum interference to τ_i caused by τ_j. However, the low coefficients of variation recorded in Fig. 5 imply that different interleavings of τ_i and τ_j will cause only minor variations in the cost of τ_i. As discussed in Section 4.3 below, SRT systems may tolerate some cost overruns.

While we have defined $C_{i:i}$ as the cost of τ_i with no co-schedule, the main diagonal of Fig. 5 shows how much slower a task runs when executed with a second copy of itself. This is irrelevant for real-time systems in which task parallelism is forbidden, but is relevant to systems in which different jobs of the same task may execute in parallel, as discussed by Voronov, Anderson, and Yang [27]. Prior to performing our experiments, we had expected that tasks executed alongside copies of themselves would have very low $r_{i:j}$, values, due to competing for the same resources, but our experiments show this is not necessarily the case.

4.2 Benchmark Characterization

In our results, we observe that tasks are relatively consistent both in how vulnerable they are to interference from other tasks and in how much interference they cause to other tasks. This is similar to other results in the literature [1, 2, 14, 25]. We say that tasks that experience



interfering benchmark / measured benchmark	adpcm_dec	adpcm_enc	ammunition	cjpeg_transupp	cjpeg_wrbmp	dijkstra	epic	fmref	gsm_dec	gsm_enc	h264_dec	huff_enc	mpeg2	ndes	petrinet	rijndael_dec	rijndael_enc	statemate	susan	MINIMUM	max CV
adpcm_dec	0.97	0.96	0.98	0.98	0.99	0.99	0.94	0.96	0.96	0.98	0.97	0.97	0.97	0.96	1.00	0.92	0.97	1.00	0.94	**0.92**	0.012617
adpcm_enc	0.91	0.85	0.91	0.94	0.97	0.95	0.94	0.93	0.95	0.95	0.94	0.94	0.93	0.95	0.96	0.94	0.96	0.92	0.92	**0.85**	0.011473
ammunition	0.67	0.66	0.67	0.65	0.69	0.68	0.69	0.64	0.64	0.66	0.68	0.68	0.66	0.68	0.69	0.68	0.70	0.71	0.68	**0.64**	0.001080
cjpeg_transupp	0.68	0.67	0.70	0.63	0.65	0.64	0.72	0.63	0.62	0.66	0.63	0.65	0.64	0.67	0.77	0.68	0.68	0.68	0.62	**0.62**	0.010452
cjpeg_wrbmp	0.69	0.63	0.63	0.62	0.59	0.65	0.68	0.69	0.60	0.65	0.54	0.55	0.60	0.63	0.74	0.66	0.52	0.66	0.62	**0.52**	0.061954
dijkstra	0.70	0.69	0.74	0.68	0.71	0.70	0.74	0.67	0.66	0.69	0.70	0.71	0.69	0.71	0.79	0.72	0.72	0.72	0.70	**0.66**	0.002617
epic	0.54	0.53	0.57	0.54	0.54	0.59	0.57	0.57	0.56	0.54	0.57	0.55	0.51	0.55	0.59	0.55	0.55	0.54	0.54	**0.51**	0.013115
fmref	0.75	0.75	0.76	0.73	0.75	0.66	0.77	0.74	0.73	0.73	0.71	0.71	0.73	0.75	0.79	0.76	0.76	0.76	0.67	**0.66**	0.060101
gsm_dec	0.64	0.63	0.64	0.61	0.65	0.63	0.68	0.60	0.60	0.61	0.62	0.63	0.61	0.63	0.71	0.64	0.64	0.65	0.61	**0.60**	0.011700
gsm_enc	0.59	0.58	0.60	0.56	0.61	0.62	0.64	0.57	0.57	0.59	0.60	0.61	0.58	0.61	0.65	0.61	0.62	0.63	0.59	**0.56**	0.012556
h264_dec	0.91	0.92	0.90	0.86	0.87	0.87	0.96	0.85	0.84	0.87	0.85	0.88	0.88	0.91	1.00	0.88	0.79	0.75	0.87	**0.75**	0.030283
huff_enc	0.73	0.70	0.74	0.67	0.69	0.66	0.79	0.71	0.67	0.69	0.69	0.71	0.66	0.72	0.79	0.69	0.71	0.67	0.69	**0.66**	0.023466
mpeg2	0.72	0.71	0.73	0.66	0.72	0.70	0.75	0.69	0.68	0.70	0.69	0.70	0.67	0.71	0.64	0.72	0.72	0.72	0.70	**0.64**	0.144416
ndes	0.66	0.68	0.69	0.67	0.71	0.69	0.72	0.70	0.67	0.57	0.66	0.59	0.61	0.55	0.74	0.70	0.56	0.72	0.67	**0.55**	0.029245
petrinet	6.11	1.24	0.91	5.93	1.05	5.68	0.88	0.94	1.20	0.78	1.18	0.69	0.78	0.82	0.71	0.60	1.22	0.82	0.98	**0.60**	0.875009
rijndael_dec	0.58	0.59	0.58	0.64	0.65	0.65	0.67	0.64	0.61	0.63	0.65	0.65	0.62	0.63	0.64	0.61	0.61	0.64	0.63	**0.58**	0.016778
rijndael_enc	0.56	0.56	0.57	0.61	0.63	0.64	0.65	0.62	0.59	0.60	0.63	0.63	0.59	0.60	0.60	0.59	0.58	0.61	0.61	**0.56**	0.013859
statemate	0.66	0.99	0.88	0.61	0.84	0.86	1.00	0.89	0.61	0.68	0.77	0.97	0.97	0.96	0.95	0.55	0.97	0.73	0.93	**0.55**	0.027924
susan	0.62	0.62	0.61	0.56	0.58	0.55	0.67	0.59	0.56	0.60	0.56	0.58	0.58	0.61	0.69	0.61	0.62	0.64	0.57	**0.55**	0.004294

Figure 5 Effect of SMT on execution times. Measured benchmarks execute with the listed $r_{i:j}$ values when sharing a thread with a given interfering benchmark. Shading is darkest on smallest values. Right column shows the maximum coefficient of variation experienced by each measured benchmark over all interfering benchmarks.

little interference from other tasks – i.e. tasks τ_i for which $r_{i:j}$ tends to be high – are *strong*, and that tasks which cause little interference to other tasks – i.e. τ_i for which $r_{j:i}$ tends to be high – are *friendly*. When we define a *strength score* $s_i = mean_j(r_{i:j})$ and *friendliness score* $f_i = mean_j(r_{j:i})$, no task has a Pearson correlation[7] with absolute value greater than 0.14 between s_i and f_i values. Bulpin's work on the behavior of threaded tasks discusses this lack of correlation further [1, 2].

For both values, we centered and standardized each row and column before fitting them to several common statistical distributions via a log-likelihood maximization. We found the Gaussian distribution to best approximate the results from our experiments. Applying a maximum likelihood (MLE) estimation, we found that mean 0.72 and standard deviation 0.13 were the best for s_i while mean 0.72 and standard deviation 0.04 were best for f_i.

4.3 Reliability of Measured Worst-Case Costs

We stated in Assumption 4 that C_i^h is no more than $\max_{\tau_j \in \tau^h} C_{i:j}$. While we are confident that violations will be rare, we cannot guarantee there will not be any. In particular, our assumption that all portions of τ_i receive the same amount of interference from all portions of τ_j is a potential source of timing violations. For example, let $\tau^h = \{\tau_1^h, \tau_2^h, \tau_3^h\}$ be such that $C_{1:2} = C_{1:3} = 6$. Under Assumption 5, the worst-case execution time for τ_1 is 6. Suppose τ_1 can be broken into two segments, τ_{1a} and τ_{1b}, such that $C_{1a:2} = 4$, $C_{1b:2} = 2$, $C_{1a:3} = 2$, and $C_{1b:3} = 4$. If τ_{1a} is co-scheduled with τ_2 and τ_{1b} is co-scheduled with τ_3, τ_1's total execution time would be 8, violating our stated worst-case execution costs. At present, our benchmark tests and model do not discover or account for task inter-leavings as in this scenario. In the

[7] A Pearson correlation of ± 1 indicates total positive or negative linear correlation; 0 indicates no correlation.

future, we would like to resolve this with finer-grained timing analysis and a model that does not assume task interference is independent from location within the task. In particular, breaking tasks into segments, determining execution costs per segment, as in our example, and conducting an analysis similar to this paper, but at a finer granularity, seems like a promising way forward. For now, we reiterate that we are only considering applications that are not safety-critical and where some tardiness is acceptable.

Generally, precise timing analysis on multicore is hard and contains uncertainty regardless of the added SMT challenge. Fortunately, Mills and Anderson have shown SRT systems to have expected tardiness bounds based on average rather than worst-case execution times [19]. Their approach relies on designating per-task execution budgets so that if any one job overruns its budget, it will not receive further execution time until a subsequent job of the same task could have been executed had the first job completed. These budgets come from average execution times. Therefore, so long as our costs are greater than the true average costs, any system τ that can be scheduled as we have described will remain so, though possibly with increased tardiness, even if our stated costs are not true worst-case costs.

Concerning our results here, our true interest is not in these specific times, but rather in developing a sense of how SMT-enabled tasks behave so that we can create synthetic tasks for our schedulability study that are good representations of reality.

5 Schedulability Experiments

Having shown how to schedule SMT-enabled systems and analyzed the behavior of our benchmark tasks when using SMT, it remains to be seen whether we can schedule otherwise unschedulable systems. To answer this question, we ran a series of schedulabilty experiments.

5.1 Experimental Procedure

To run our experiments, we created synthetic task systems to be scheduled on platforms with m cores, $m \in \{4, 8, 16\}$ such that the total system utilization ranged from m to $2m$. For each task system, we partitioned the system into τ^p and τ^h using Algorithm 1 and all three versions of Algorithm 2. We then tested for schedulability per Theorem 15. We created enough task systems that each data point in our graphs represents the composite schedulability of approximately 1,000 task systems. We created over 300 graphs, with a few thousand to hundreds of thousands of task systems per graph. Creating task sets, partitioning task sets, and testing for schedulability consumed over 30 days of CPU time.

We plotted our results on a series of schedulability graphs with total utilizations on the horizontal axis and the proportion of systems that were schedulable on the vertical axis. Since we started at utilization m, and the standard SRT feasibility condition given by (1) requires that $\sum_{i=1}^{n} u_i \leq m$ hold, every system we created was infeasible without using SMT. Every system that we could schedule is an argument for adapting SMT in real-time systems.

Each graph shows results for tasks created using a common set of utilization and $r_{i:j}$ values. Task utilizations were assigned from one of four ranges: the uniform distributions (0, .4], [.3, .7], [.6, 1], and (0, 1]. We used two approaches for determining $r_{i:j}$ values. In the *Gaussian-average* approach, we drew s_i and f_i from the Gaussian distributions with mean 0.72 for both values and standard deviations ranging from 0.13 to 0.39 for s_i and from 0.04 to 0.12 for f_i. These parameters are based on distributions we fitted to our models, as discussed in the previous section. We allowed larger standard deviations than we obtained from our benchmarks to make our results more widely applicable.

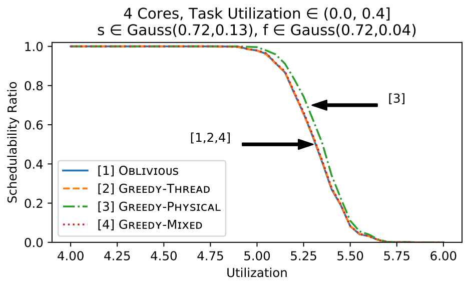


Figure 6 Graph shape is similar to Fig. 1, which has more cores.

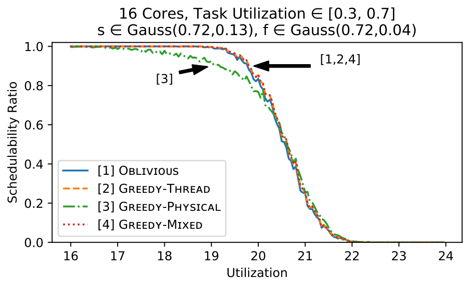


Figure 7 Schedulability similar to Figs. 1 and 6, despite higher task utils.

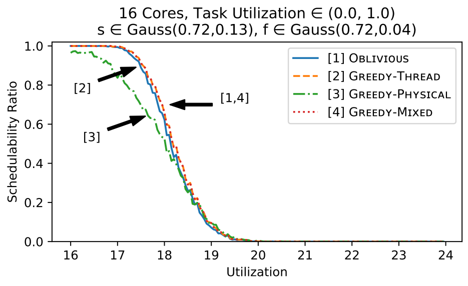


Figure 8 Despite same expected per-task util. as Fig. 7, schedulability is reduced.

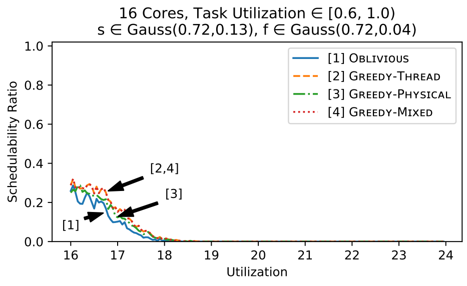


Figure 9 Given high per-task utilizations, only small schedulability gains can be achieved.

In the *uniform-normal* approach, both s_i and f_i come from one of four uniform distributions: [.65, 1], [.7, 1], [.75, 1], or [.8, 1]. The two ranges may differ for a given graph. Each $r_{i:j}$ value was then chosen from a normal distribution with mean $s_i f_i$ and standard deviation σ, where σ is .01, .05, or .1. Negative values or those greater than 1 are clamped to 0 or 1 respectively. The intuition behind the uniform-normal approach is to create $r_{i:j}$ values broadly similar to the benchmark values we obtained, but via different methods than Gaussian-average so as to avoid having our results be overly dependent on that model. While high s_i values in this context still indicate tasks that receive little interference from other tasks, and high f_i values indicate tasks that cause little interference to others, they are used differently here than in the Gaussian average approach and should not be directly compared.

5.2 Results

Due to space constraints, we present only a small portion of our graphs to highlight general trends. A full set of graphs is available in our online appendix.[8] For all of our graphs, the horizontal axis begins at m; all of our task systems would be infeasible without SMT.

▶ **Observation 1.** *Given favorable task parameters, virtually all task systems with utilizations as high as* $1.25m$, *and roughly half of task systems with utilizations of* $1.33m$, *are schedulable. Favorable task parameters are high means and low standard deviations for friendliness and strength values combined with low per-task utilizations. Examples of these results are seen in Figs. 1, 6, and 7.*

[8] Appendix and code is available at `https://jamesanderson.web.unc.edu/papers/`. Code is also available at `https://github.com/JoshuaJB/SMART-ECRTS19` and `https://doi.org/10.4230/DARTS.5.1.8`.

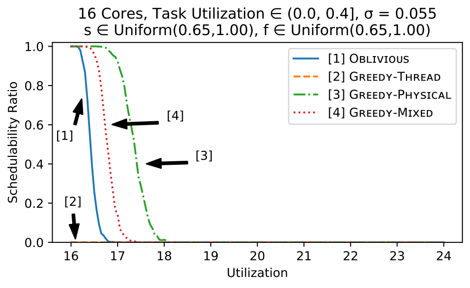


Figure 10 Uniform-normal $r_{i:j}$ values on 16 cores. Note variations in algorithm performance.

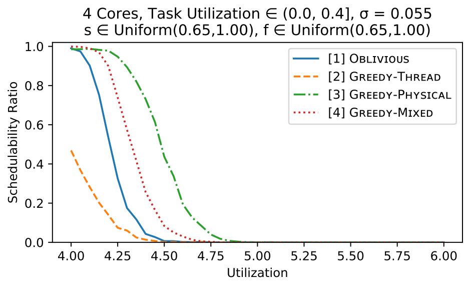


Figure 11 Uniform-normal $r_{i:j}$ on 4 cores. Unlike the Gaussian model, core count influences gains from SMT here.

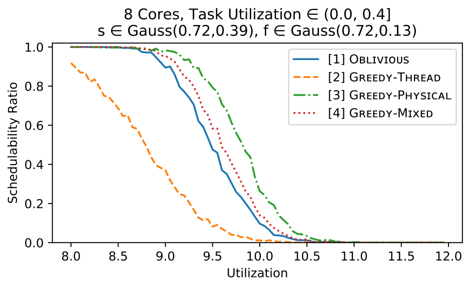


Figure 12 Gaussian approach with higher variance. Gains from SMT are reduced compared to Figs. 1, 6, and 7.

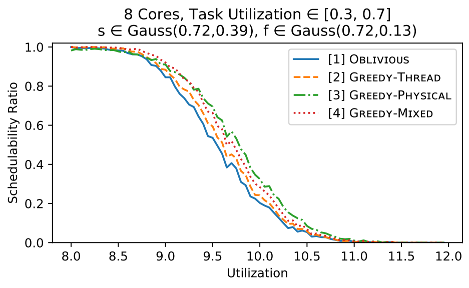


Figure 13 Underperformance of Greedy-Thread as in Fig. 12 disappears as utilizations increase.

▶ **Observation 2.** *Task systems with low per-task utilization received the greatest improvement in schedulability, and task systems with high utilization saw the least. Since threading tasks increases individual execution costs, it will typically not be possible to thread tasks that already have high utilizations. Fig. 6, in our introduction, shows schedulability for task systems with individual utilizations drawn from the uniform distribution* $(0, 0.4]$*, and shows that the majority of systems considered are schedulable with utilizations as high as* 5.34*. Fig. 9 has the same parameters as Figs. 1 and 6, but draws utilizations instead from the range* $[.6, 1]$*. This graph shows virtually no improvement when run with SMT.*

▶ **Observation 3.** *Algorithm 1, oblivious partitioning, competes with the more complex algorithms. In our best results, such as Figs. 1, 6, 7, and 13, Algorithm 1 is indistinguishable from the greedy algorithms. When Algorithm 1 does not perform as well as the variants of Algorithm 2, the difference is small enough that the lower algorithm complexity might still make it a better choice.*

▶ **Observation 4.** *Lower* $r_{i:j}$ *variability yields improved schedulability. In Fig. 12, the task systems sample from the same utilization range as those of Figs. 1 and 6, but here the standard deviation of the distribution from which* s_i *and* f_i *are sampled is larger. This increased variance causes fewer task sets to be schedulable Fig. 12 in than in Figs. 1 and 6.*

▶ **Observation 5.** *Schedulability benefits of our methods are not limited to task systems generated using a single model. While the Gaussian approach created systems that saw more improvement from SMT, the benefits of SMT are not limited to task systems created under that model, suggesting that SMT can benefit a wide variety of task systems.*

6 Conclusion

We have given a task model, SMART, that allows for reasoning about SMT-enabled task systems by defining multiple cost parameters per task. We have shown how to decide which tasks should and should not use SMT and how to take advantage of SMT to schedule otherwise unschedulable task systems. We measured the execution times of benchmark tasks with and without SMT enabled, with the SMT-enabled case covering interference from all other tasks in the set. We conducted an extensive schedulability study using synthetic tasks modeled on our benchmark tasks and showed that for task systems consisting of low utilization tasks, it is possible to schedule virtually all systems with utilization as large as $1.25m$ and to schedule many task systems with utilizations approaching $1.33m$.

In the future, we plan to improve our timing analysis to the point that hard real-time systems, where no tardiness is permitted, becomes an option. In addition, we want expand our soft real-time work by partitioning both task systems and hardware platforms to minimize tardiness, rather than simply maximizing schedulability. Making tasks threaded tends to decrease demand on the platform, potentially reducing tardiness, but will increase execution costs, potentially increasing tardiness [4, 5, 16]. While the potential gains shown in this paper are substantial, we have only begun to expose the potentials of hardware multithreading.

References

1 J. Bulpin. *Operating system support for simultaneous multithreaded processors*. PhD thesis, University of Cambridge, King's College, 2005. URL: `http://www.cl.com.ac.uk/TechReports/`.

2 J. Bulpin and I. Pratt. Multiprogramming Performance of the Pentium 4 with Hyperthreading. In *Third Annual Workshop on Duplicating, Deconstruction and Debunking*, pages 53–62, June 2004.

3 F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. Predictable performance in SMT processors: synergy between the OS and SMTs. *IEEE Transactions on Computers*, 55(7):785–799, July 2006. `doi:10.1109/TC.2006.108`.

4 U. M. C. Devi and J. H. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. In *RTSS'05*, pages 330–341, December 2005. `doi:10.1109/RTSS.2005.39`.

5 U. M. C. Devi and J. H. Anderson. Flexible tardiness bounds for sporadic real-time task systems on multiprocessors. In *20th IEEE International Parallel Distributed Processing Symposium*, pages 10 pp.–, April 2006. `doi:10.1109/IPDPS.2006.1639265`.

6 S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: a platform for next-generation processors. *IEEE Micro*, 17(5):12–19, September 1997. `doi:10.1109/40.621209`.

7 S. Eyerman and L. Eeckhout. The Benefit of SMT in the Multi-core Era: Flexibility Towards Degrees of Thread-level Parallelism. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 591–606, New York, NY, USA, 2014. ACM. `doi:10.1145/2541940.2541954`.

8 S. Eyerman, P. Michaud, and W. Rogiest. Revisiting symbiotic job scheduling. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 124–134, March 2015. `doi:10.1109/ISPASS.2015.7095791`.

9 H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sorensen, P. Wagemann, and S. Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:10, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/OASIcs.WCET.2016.2`.

10 J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. Perf Fair: A Progress-Aware Scheduler to Enhance Performance and Fairness in SMT Multicores. *IEEE Transactions on Computers*, 66(5):905–911, May 2017. `doi:10.1109/TC.2016.2620977`.

11 A. Fog. *The Microarchitecture of Intel, AMD, and VIA CPUs: an optimization guide for assembly programmers and compiler makers*. Technical University of Denmark, 2018. URL: `https://www.agner.org/optimize/microarchitecture.pdf`.

12 T. Gomes, P. Garcia, S. Pinto, J. Monteiro, and A. Tavares. Bringing Hardware Multithreading to the Real-Time Domain. *IEEE Embedded Systems Letters*, 8(1):2–5, March 2016. `doi:10.1109/LES.2015.2486384`.

13 T. Gomes, S. Pinto, P. Garcia, and A. Tavares. RT-SHADOWS: Real-time system hardware for agnostic and deterministic OSes within softcore. In *ETFA '15*, pages 1–4, September 2014. `doi:10.1109/ETFA.2015.7301572`.

14 W. Huang, J. Lin, Z. Zhang, and J.M. Chang. Performance Characterization of Java Applications on SMT Processors. In *ISPASS '05.*, pages 102–111, March 2005. `doi:10.1109/ISPASS.2005.1430565`.

15 R. Jain, C. J. Hughes, and S. V. Adve. Soft real-time scheduling on simultaneous multithreaded processors. In *RTSS '02*, pages 134–145. Institute of Electrical and Electronics Engineers Inc., 2002. `doi:10.1109/REAL.2002.1181569`.

16 H. Leontyev and J. H. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. *Real-Time Systems*, 44(1):26–71, March 2010. `doi:10.1007/s11241-009-9089-2`.

17 S. Lo, K. Lam, and T. Kuo. Real-time task scheduling for SMT systems. In *RTCSA'05*, pages 5–10, August 2005. `doi:10.1109/RTCSA.2005.77`.

18 D. Marr, F. Binns, D. Hill, G. Hinton, K. Koufaty, J. Miller, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. In *Intel Technology Journal*, volume 6, pages 4–15, February 2002.

19 A. F. Mills and J. H. Anderson. A Stochastic Framework for Multiprocessor Soft Real-Time Scheduling. In *RTAS '10*, pages 311–320, April 2010. `doi:10.1109/RTAS.2010.33`.

20 J. Mische, S. Uhrig, F. Kluge, and T. Ungerer. Using SMT to Hide Context Switch Times of Large Real-Time Tasksets. In *RTAS '10*, pages 255–264, August 2010. `doi:10.1109/RTCSA.2010.33`.

21 B. Ocker. FAA special topics. In *Collaborative Workshop: Solutions for Certification of Multicore Processors*, November 2018.

22 S. Osborne and J. H. Anderson. Work in Progress: Combining Real Time and Multithreading. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 139–142, December 2018. `doi:10.1109/RTSS.2018.00024`.

23 P. Radojković, P. M. Carpenter, M. Moretó, V. Čakarević, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero. Thread Assignment in Multicore/Multithreaded Processors: A Statistical Approach. *IEEE Transactions on Computers*, 65(1):256–269, January 2016. `doi:10.1109/TC.2015.2417533`.

24 A. Snavely and D. M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In *ASPLOS '2000*, ASPLOS IX, pages 234–244, New York, NY, USA, 2000. ACM. `doi:10.1145/378993.379244`.

25 N. Tuck and D. M. Tullsen. Initial Observations of the Simultaneous Multithreading Pentium 4 Processor. In *PACT '03*, PACT '03, pages 26–35, Washington, DC, USA, 2003. IEEE Computer Society. URL: `http://dl.acm.org/citation.cfm?id=942806.943857`.

26 D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA '95*, pages 392–403, 1995.

27 S. Voronov, J. H. Anderson, and K. Yang. Tardiness Bounds for Fixed-Priority Global Scheduling Without Intra-Task Precedence Constraints. In *RTNS '18*, RTNS '18, pages 8–18, New York, NY, USA, 2018. ACM. `doi:10.1145/3273905.3273913`.

28 M. Zimmer, D. Broman, C. Shaver, and E. A. Lee. FlexPRET: A processor platform for mixed-criticality systems. In *RTAS '14*, pages 101–110, April 2014. `doi:10.1109/RTAS.2014.6925994`.

PREM-Based Optimal Task Segmentation Under Fixed Priority Scheduling

Muhammad R. Soliman
University of Waterloo, Ontario, Canada
mrefaat@uwaterloo.ca

Rodolfo Pellizzoni
University of Waterloo, Ontario, Canada
rpellizz@uwaterloo.ca

Abstract

Recently, a large number of works have discussed scheduling tasks consisting of a sequence of memory phases, where code and data are moved between main memory and local memory, and computation phases, where the task executes based on the content of local memory only; the key idea is to prevent main memory contention by scheduling the memory phase of one task in parallel with computation phases of tasks running on other cores. This paper provides two main contributions: (1) we present a compiler-level tool, based on the LLVM intermediate representation, that automatically converts a program into a conditional sequence of segments comprising memory and computation phases; (2) we propose an algorithm to find optimal segmentation decisions for a task set scheduled according to a fixed-priority partitioned scheme. Our evaluation shows that the proposed framework can be feasibly applied to realistic programs, and vastly overperforms a baseline greedy approach.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases PREM, LLVM, scratchpad memory, scheduling, program segmentation

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.4

Funding This work was supported in part by NSERC and CMC Microsystems.

1 Introduction

Multi-Processor Systems-on-a-Chip (MPSoCs) are becoming increasingly popular in the real-time and embedded system community. MPSoCs are characterized by the presence of shared memory resources. In particular, a single main memory shared by all processing elements on the chip can constitute a significant performance bottleneck. Even worse, hardware arbitration schemes used in Commercial-Off-The-Shelf (COTS) systems are optimized for average-case performance, resulting in extremely high worst-case latency in the presence of contention for memory access among multiple processors [17, 30, 16].

Hence, there is a significant interest in the real-time community in controlling the pattern of accesses in memory to avoid worst-case scenarios. This can be difficult in cache-based systems, where main memory accesses are generated by misses in last level cache, as the precise pattern of cache hits and misses is hard to predict. The PRedictable Execution Model (PREM) first proposed in [22] attempts to solve this issue by dividing the execution of each software task in two different parts: memory phases where the data and instructions required by the task are loaded from main memory into local memory (cache or scratchpad), and computation phases where a processor executes the task based on the content of local memory only. Since the task does not need to access main memory during its computation phase, other processors are free to do so without suffering contention.

Based on this core idea, successive works [33, 31, 32, 2, 1, 3, 27, 34, 19, 20, 12, 7, 21, 9, 10, 23, 4] have proposed a variety of contentionless approaches [1] targeting different scheduling schemes (preemptive vs non-preemptive, partitioned vs global) and platforms (general purpose processors vs GPU). However, the key problem of how to compile a program to execute based on PREM has received significantly less attention. Due to the complexities inherent in each step, we strongly believe that an automated tool is required to remove the burden from the programmer.

The main contribution of this paper is a framework for automatically generating PREM-compatible code for sequential programs running on a general purpose processor; it is largely agnostic to the programming language being used since it operates on the intermediate representation of the LLVM compiler infrastructure [18]. In particular, we propose a set of program transformation constraints that allow us to convert a task into a conditional sequence of PREM segments. We use a region-based approach to simplify segment creation, in conjunction with loop splitting and tiling transformations [15] to split large loops into multiple segments. Based on the proposed framework, we then derive a task segmentation algorithm that enumerates the best possible conditional segments for a given task on a platform with fixed-length memory phases [27]. Furthermore, for the case of fixed-priority partitioned scheduling, we show that applying the algorithm to each task in priority order leads to a solution that is optimal for the task set.

The rest of the paper is organized as follows. Section 2 summarizes the required background on PREM and related work. Section 3 introduces our new conditional PREM model and extends the existing schedulability analysis to cover such model. Section 5 then shows how to obtain an optimal segmentation for a given task set, while Section 4 describes our employed compilation framework, and our program segmentation algorithm based on such framework. Section 6 compares our optimal segmentation approach versus both a previous greedy approach, and a simple heuristic, using task set parameters extracted from real programs. Finally, we conclude in Section 7.

2 Background and Related Work

In this section, we introduce existing research based on PREM and discuss required background and system assumptions. Note that while other predictable management approaches for local memories exist in the literature, we limit ourselves to PREM-based solutions due to space limitations. We consider a MPSoC platform comprising a set of possibly heterogeneous processors. Each processor has a fast private local memory in the form of a last level cache or ScrachPad Memory (SPM); all processors share the same main memory. As discussed in Section 1, the goal of PREM is to create a contentionless memory schedule. While the seminal work in [22] first proposed to split the execution of each application into a memory and a computation phase, the approach has been refined in successive works [32, 2] into a three-phase model. Here, two memory phases are considered: an acquisition (or load) phase that copies data and instructions from main memory into local memory, and a replication (or unload) phase that copies modified data back to main memory. Memory phases are scheduled such that a single memory phase is executed at any one time in the entire system.

When the data used by a program is small and deterministic, the task can comprise a single sequence of load-computation-unload phases. However, the code and data of the program might be too large to fit in one partition of local memory. Second, it might be

[1] Note that the model we are discussing is also referred to as three-phase model or acquisition-execution-replication model in related work.

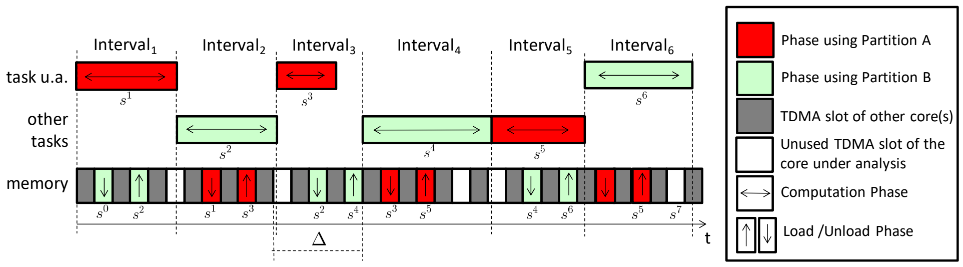


Figure 1 Example: TDMA memory schedule with $M = 2$ cores.

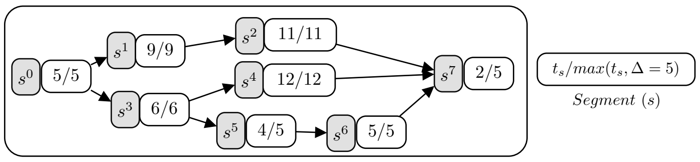


Figure 2 Example segment DAG (s^0 is s^{begin} and s^7 is s^{end}).

difficult to predict the data accessed by a job before it starts executing, as data accesses can be dependent on program inputs. To address such issue, the works in [22, 32, 9, 20] split a task into a sequence of PREM segments, where each segment has its own memory and computation phases and is executed non-preemptively.

2.1 Memory and Processor Schedule

The memory scheduling algorithm is different among related work, based on their specific goals and system assumption. Approaches targeted at multitasking systems optimize task execution by overlapping the computation of the current job with the memory phase for the next job to be scheduled on that processor. In essence, one can pipeline computation and memory phases using a double-buffering technique [32, 13, 12, 27], at the cost of halving the available local memory space. As an example, we detail the approach in [32, 27], which has been designed to schedule a set of fixed-priority, partitioned sporadic tasks, and fully implemented on an automotive COTS platform. The local memory of each processor is divided into two equal size partitions. Memory phases are executed by a dedicated DMA component using a TDMA memory schedule with fixed time slots; the size of each slot is sufficient to either load or unload the entirety of one partition. Figure 1 shows an example schedule on one processor; the task under analysis (u.a.) consists of three segments s^1, s^3 and s^6, while segments s^2, s^4 and s^5 belong to other tasks. The schedule consists of a sequence of scheduling intervals. Segments are scheduled non-preemptively. During each interval, a segment of a job (ex: s^2 in Interval$_2$) computes using data and instruction in one partition. At the same time, the DMA unloads the previous segment (s^1) and loads the next segment (s^3) in the other partition. Note that the length of each scheduling interval is the maximum of the computation time for the corresponding segment, and the time required for the load and unload operations. In the figure, Interval$_3$ is bounded by the memory time, while all other intervals are bounded by the computation time of the segment. Let M be the number of cores, and σ the size of each TDMA slot. Then as proven in [27], the worst-case memory time is equal to $\Delta = \sigma \cdot (2M + 1)$: as again shown in Interval$_3$, the previous interval can finish right after the beginning of a TDMA slot assigned to the core under analysis, forcing that slot to be wasted. To abstract from the details of the memory schedule, in the rest of the paper we assume a given bound Δ on the memory time for any interval. Hence, the length of an interval is the maximum of Δ and the computation time of the job in that interval. Finally, note that two segments of the same task cannot run back-to-back as the computation phase of a segment and the memory phase of the next one cannot be executed in parallel: in general, the data required by a segment might not be determined until the previous segment completes; furthermore, to load a segment we might need to first evict some data and code of the previous one. To avoid idling the processor while a task loads its next segment, one or more segments of (possibly lower priority) other tasks are instead scheduled.

A downside of the described approach is that a high priority job can suffer blocking by a low priority job due to the non-preemptive interval schedule. The works in [33, 34, 21] adopt preemptive scheduling, but this requires a number of local memory partitions equal to the number of tasks: otherwise, a memory phase could be "wasted" by loading a job that is immediately preempted by a higher priority one. Given that local memory is typically a limited resource, we will not consider such fully-preemptive approaches.

2.2 Program Transformation

We next discuss how a program can be transformed to be PREM-compliant. Most single-segment works do not require program transformation; instead, the entire memory region allocated by the OS to the program is loaded in local memory [32, 7, 27, 4]. The seminal work in [22] introduces a set of macros, which the programmer could add to the program to both segment it, and mark data structures to be loaded / unloaded. Our experience with programs of even medium complexity is that this places an undue burden on the programmer, and it is likely to lead to a sub-optimal transformation. The authors of [13, 12] discuss a compiler-based approach to transform a GPU kernel. The approach focuses on generating code for the memory phase. On the other hand, our focus in this paper is how to automate data usage analysis and task segmentation for sequential programs running on a general purpose processor. Light-PREM [19] uses run-time profiling to detect memory areas used by a program to load during memory phases. We find the approach suitable for programs with highly dynamic data structures, but since it is based on profiling rather than static program analysis, it cannot guarantee worst-case bounds. Also, it does not discuss how to segment a task. In our previous work [24], we proposed a program analysis and transformation technique that uses static analysis to determine data accesses and predictably load/unload data from SPM while the program is executing. We reuse the same compiler framework in Section 4 to determine the data to load in each segment. Note that [24] only deals with a single-task, single processor case, and does not segment the program based on PREM.

The closest related work is [20], where the authors introduce an automated task compilation and segmentation tool. The approach is similar to our work in that is relies on the LLVM compiler infrastructure, and employs loop splitting and tiling [15] to break loops that are too large to fit in local memory. However, the paper is focused on the case of a parallel, single-task system, and the tool employs a "greedy" segmenting approach that results in the longest possible segments. As we discuss in Section 3 and show in Section 6, such greedy approach is not suitable for multi-tasking systems where blocking time due to non-preemptive segments of lower priority tasks is a concern.

Finally, all related work assumes that a task comprises a single segment or a fixed sequence of segments. However, a program can have multiple execution paths whereas it accesses different data along each path, and must be PREM-compliant along all valid paths. Therefore, in Section 3 we introduce a new conditional PREM model in which the fixed segment sequence is replaced by a Directed Acyclic Graph (DAG) of segments, and we then show how to compile the program to execute segments conditionally.

3 Task Model and Schedulability Analysis

We consider scheduling a set of sequential, conditional PREM tasks on a multiprocessor. We assume non-preemptive segment execution, with a fixed memory time Δ to load/unload each segment. In details, we consider a set of sporadic tasks $\Gamma = \{\tau_1, \ldots, \tau_N\}$. We use T_i to denote the period (or minimum inter-arrival time) of task τ_i, and D_i for its relative deadline. We assume constrained deadline: $D_i \leq T_i$. τ_i is further characterized by a DAG of segments $G_i = (S_i, E_i)$, where S_i is a set of nodes representing segments, and E_i is a set of edges representing precedence constraints between segments. We assume that the set S_i contains unique source and sink segments s^{begin}, s^{end}, as we consider programs with a single entry and exit point. We define the length $s.l$ of a segment $s \in S_i$ as the maximum length of any scheduling interval for the segment, that is, the maximum between the worst-case computation time t_s of s (including context-switch overheads) and the memory time Δ. In the remaining of the paper, we use p to denote a DAG path, that is, an ordered sequence of segments; $p.I$ is the number of segments in the path, $p.L$ the sum of their lengths, and $p.end$ the length of the last segment in the path. We say that a path is maximal if its first segment is s^{begin} and its last segment is s^{end}. To avoid confusion, in the rest of the paper we use uppercase letters (P) to denote maximal paths. Note that by definition $P.end = s^{end}.l$. Figure 2 shows an example DAG with three maximal paths: $P = \{s^0, s^1, s^2, s^7\}$, $P' = \{s^0, s^3, s^4, s^7\}$, and $P'' = \{s^0, s^3, s^5, s^6, s^7\}$. Note that we have $P.L = 30, P.I = 4, P'.L = 28, P'.I = 4, P''.L = 26, P''.I = 5$, and $P.end = P'.end = P''.end = 5$. Finally, we will use the notation $p = \{p_1, ..., p_n\}$ to indicate that path p can be obtained as a sequence of n (sub-)paths. In general, a DAG could have many maximal paths, and a task could be segmented into many different DAGs. The following definitions will allow us to restrict the number of paths / DAGs to find a schedulable task system.

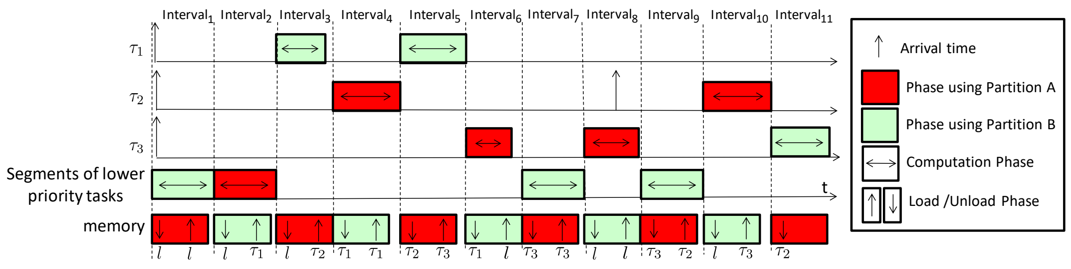
▶ **Definition 1.** *Given two maximal paths P, P', we say that P' dominates (is worse than or equal to) P and write $P' \succeq P$ iff: $P'.L \geq P.L$ and $P'.I \geq P.I$ and $P'.end \leq P.end$. If neither $P' \succeq P$ nor $P \succeq P'$ holds, we say that the two paths are incomparable.*

Since the $\succeq$ relation defines a partial order between maximal paths, we can characterize a task based on its set of dominating paths. Formally, given segment DAG G, we use $G.C$ to denote the Pareto frontier [2] of all maximal paths in G. Intuitively, for a task τ_i, we will show that the set $G_i.C$ replaces the concept of worst-case execution time. For example, for Figure 2, $G.C$ is the set P, P''; P' is not included since P dominates it; but both P and P'' are included since they are incomparable. While $P'.end = P.end$ for two paths belonging to the same DAG, we can also use Definition 1 to compare two DAGs for the same program.

▶ **Definition 2.** *Given two segment DAGs G, G', we say that G' dominates (is worse than or equal to) G and write $G' \succeq G$ iff: $\forall P \in G.C, \exists P' \in G'.C : P' \succeq P$. If neither $G' \succeq G$ nor $G \succeq G'$ holds, the two DAGs are incomparable.*

Note that since $G.C$ is the Pareto frontier, $G' \succeq G$ implies that for every path in G, there is a corresponding path in G' that dominates it.

[2] Given a partial order over a set of distinct elements, the Pareto frontier is the subset of elements that are not dominated by any other element.



Interval1 Interval2 Interval3 Interval4 Interval5 Interval6 Interval7 Interval8 Interval9 Interval10 Interval11
τ1
τ2
τ3
Segments of lower priority tasks
memory
t
l l l τ1 l τ2 τ1 τ1 τ2 τ3 τ1 l τ3 τ3 l l τ3 τ2 l τ3 τ2
Arrival time
Phase using Partition A
Phase using Partition B
Computation Phase
Load /Unload Phase

Figure 3 Example critical instant.

3.1 Schedulability Analysis and Preliminaries

We now consider a partitioned system with fixed per-task priority, and extend the analysis in [32, 27] to support conditional task execution. Since tasks are partitioned among cores and the effect of the memory schedule is captured by the memory time Δ, each core can be analyzed independently. Therefore, let $\Gamma = \{\tau_1, \ldots, \tau_N\}$ represent the set of tasks on the core under analysis, ordered by decreasing, distinct priorities, and assume that each task τ_i is associated with a given segment DAG G_i. The scheduling algorithm follows the scheme introduced in Section 2.1, where each SPM is divided in two partitions and the schedule is a sequence of scheduling intervals. In details: at the beginning of each scheduling interval, we execute on the processor the segment loaded during the previous interval (if any). In parallel, we unload and load the other local memory partition with the next segment of the highest priority ready task.

The critical instant for a task under analysis τ_3 (the task arrival pattern that leads to the worst case response time for the task under analysis), as derived in [32], is depicted in Figure 3. Since scheduling decisions are only made when an interval starts, the worst case arrival pattern corresponds to the task under analysis and all higher priority tasks arriving just after the beginning of an interval for a lower priority task (Interval$_1$ in the figure). As a consequence, the task under analysis suffers an initial blocking time B_i equal to two intervals: neither the task under analysis nor higher priority tasks can execute for the first two intervals, as another lower priority segment loaded during Interval$_1$ executes during Interval$_2$. More in general, let τ_i be the task under analysis, and let $l_i^{l\,\max}$ denote the maximum length of any segment of a lower priority task. Albeit pessimistically, we then bound the blocking time as:

$$l_i^{l\,\max} = \max(\Delta, \max_{j=i+1,N} \max_{s \in S_j} s.l) \tag{1}$$

$$B_i = \begin{cases} 2 \cdot l_i^{l\,\max}, & \text{if } i \le N-2. \\ l_{N-1}^{l\,\max} + \Delta, & \text{if } i = N-1. \\ \Delta, & \text{if } i = N. \end{cases} \tag{2}$$

For task τ_{N-1}, there is only one lower priority task; hence, the first blocking interval has only a memory phase and no task computation, while task τ_N computes in the second blocking interval. For τ_N, there is only one initial blocking interval consisting of a memory phase. Note that in the worst case, each successive segment of τ_i can suffer a blocking time equal to $l_i^{l\,\max}$ since two segments of τ_i cannot be executed back-to-back (Interval$_6$ and Interval$_8$ in the figure). For τ_N, we set $l_i^{l\,\max} = \Delta$ since there are no lower priority tasks, but a scheduling interval with memory only would be needed between successive segments of τ_N.

Since higher priority tasks arrive synchronously with the task under analysis, the interference suffered by τ_i in an interval of length t is equal to:

$$\text{Inter}_i(t) = \sum_{j=1}^{i-1} \lceil t/T_j \rceil \cdot L_j, \tag{3}$$

where L_j is the length of the path taken by τ_j. Since we cannot make any assumption on path execution, we maximize the interference by considering the path with maximum length:

$$L_j^{\max} = \max\{P.L \mid P \in G_j.C\}. \tag{4}$$

Note that it is sufficient to consider only the maximal paths in $G_j.C$ since each maximal path in G_j is dominated by a path in $G_j.C$, and by Definition 1 the dominating path has longer or equal L. Finally, since segments are executed non-preemptively, a task will complete by its deadline if its last segment starts execution $P.end$ time units before its deadline. Therefore, for a maximal path P, the response time $R_i(P)$ of τ_i up to its last segment can be computed as a standard iteration:

$$R_i(P) = B_i + (P.I - 1) \cdot l_i^{l\,\max} + P.L - P.end + \text{Inter}_i\big(R_i(P)\big), \tag{5}$$

and the task is schedulable along that path if:

$$R_i(P) \le D_i - P.end. \tag{6}$$

Here, $P.L - P.end$ represents the length of intervals where τ_i computes (excluding the last segment), B_i is the blocking suffered by the first segment, $(P.I - 1) \cdot l_i^{l\,\max}$ is the blocking suffered by other segments, and $\text{Inter}_i\big(R_i(P)\big)$ is the interference of higher priority tasks. We next prove three key properties of the analysis.

▶ **Property 1.** *Consider two paths P, P' with $P' \succeq P$. If Equation 6 holds for P', then it also holds for P.*

Proof. Note that Equation 3 is increasing in t, and Equation 5 is increasing in $P.I$ and $P.L$ and decreasing in $P.end$. Since it holds $P'.L \ge P.L$, $P'.I \ge P.I$, $P'.end \le P.end$, at convergence it must hold: $R_i(P') \ge R_i(P)$.

Now by hypothesis it holds: $R_i(P') \le D_i - P'.end$, which is equivalent to: $D_i \ge B_i + (P'.I - 1) \cdot l_i^{l\,\max} + P'.L + \text{Inter}_i\big(R_i(P')\big)$. But since we have: $B_i + (P'.I - 1) \cdot l_i^{l\,\max} + P'.L + \text{Inter}_i\big(R_i(P')\big) \ge B_i + (P.I - 1) \cdot l_i^{l\,\max} + P.L + \text{Inter}_i\big(R_i(P)\big)$, we obtain: $D_i - P.end \ge B_i + (P.I - 1) \cdot l_i^{l\,\max} + P.L - P.end + \text{Inter}_i\big(R_i(P)\big)$, completing the proof. ◀

Based on Property 1, to check the schedulability of τ_i it is sufficient to test the set of dominating maximal paths. Hence, the following lemma immediately follows, where $\bigwedge$ denotes a logical and.

▶ **Lemma 3.** *Task τ_i is schedulable if:*

$$\bigwedge_{P \in G_i.C} R_i(P) \le D_i - P.end. \tag{7}$$

▶ **Property 2.** *According to the analysis: (A) the schedulability of task τ_i depends on the maximum length $l_i^{l\,\max}$ of any segment of lower priority tasks $\tau_i + 1, \dots \tau_N$, but not on any other parameter of those tasks; (B) if τ_i is schedulable for a value l of $l_i^{l\,\max}$, then it is also schedulable for any other value $l' \le l$.*

Proof. Part (A): by definition of Equations 5, 7. Part (B): since R_i is increasing in $l_i^{l\max}$, the response time for $l_i^{l\max} = l'$ cannot be larger than the one for l. ◀

If the segment DAG G_i for each task $\tau_i \in \Gamma$ is known, then task set schedulability can be assessed by checking Equation 7 for all tasks in the order $\tau_1, \dots, \tau_N$. However, we are interested in using the response time of tasks $\tau_1, \dots, \tau_i$ in order to optimize the segmentation of task τ_{i+1}, hence $G_{i+1}, \dots, G_N$ are not known when analyzing τ_i. Based on Property 2, we instead use the analysis to determine the maximum value $\overline{l_i^{l\max}}$ of $l_i^{l\max}$ under which τ_i is still schedulable. Such value is then used by our segmentation algorithm working on τ_{i+1}, as we detail in the next section: the algorithm considers a segmentation of τ_{i+1} to be valid only if its maximum segment length is no larger than $\overline{l_i^{l\max}}$. Note that in theory, one could determine $\overline{l_i^{l\max}}$ by performing a binary search over Equation 7. However, we show in technical report [26] that an alternative formulation based on the concept of scheduling points used in [5] can be used to derive $\overline{l_i^{l\max}}$ directly.

▶ **Property 3.** *Consider two DAGs G_j, G'_j for task τ_j where $1 \le j \le i$ and $G'_j \succeq G_j$. If τ_i is schedulable for G'_j according to the analysis, then it is also schedulable for G_j.*

Proof. Case $j = 1, \dots i-1$: since $G'_j \succeq G_j$, the value of $L_j^{\max}$ for G_j is no larger than for G'_j. Since the interference $\text{Inter}_i(t)$ is increasing in $L_j^{\max}$, the resulting response time of τ_i for G_j cannot be larger than the one for G'_j.

Case $j = i$: since $G'_i \succeq G_i$, for each maximal path $P \in G_i.C$ there must exist a maximal path $P' \in G'_i.C$ such that $P' \succeq P$. Now since τ_i is schedulable for G'_i according to the analysis, by Equation 7 it must hold $R_i(P') \le D_i - P'.end$; then by Property 1, it must also hold $R_i(P) \le D_i - P.end$. This means that Equation 7 holds for G_i, concluding the proof. ◀

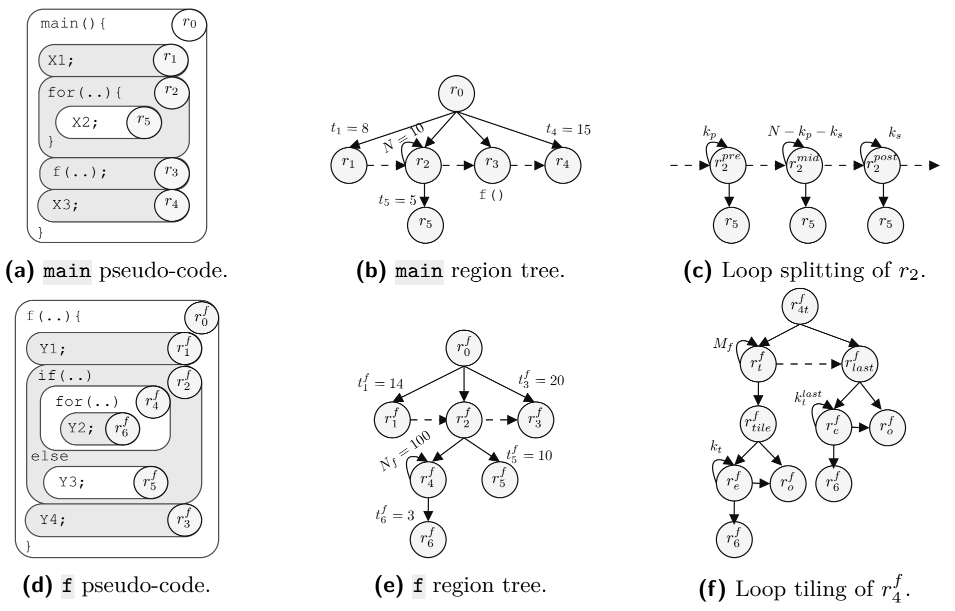
Property 3 shows that the dominance relation indeed corresponds to the notion of a DAG being better than another from a schedulability perspective. Hence, the objective of our segmentation algorithm is to find a set of "best" DAGs for a task based on Definition 2. Intuitively, the rest of the paper proceeds as follows. In Section 4 we present a segmentation algorithm that explores the set of all valid DAGs for a program, based on a set of constraints which include the maximum segment length, but quickly cuts dominating (i.e., worse) DAGs inspired by Property 3. Then, in Section 5 we show that, based on Property 2, we can invoke the segmentation algorithm on each task in priority order and obtain a set of DAGs (one for each task) that is optimal from a schedulability perspective.

4 Program Segmentation

In this section, we show how a task is compiled into segments. We start by discussing the program structure based on regions. After that, we define valid segmentations according to our compiler framework, which is based on LLVM [18] and the work in [24]. Finally, we detail our algorithm, which segments the program and returns the set of all DAGs that could be optimal. Similarly to [24], we assume that the program follows common real-time coding conventions. Therefore, the code should not use recursion or function pointers and all loops in the program are bounded. We also assume that the WCET and footprint of any part of the program are known either using static analysis or measurement.

4.1 Program Structure

We adopt the region-based program structure introduced in [24] which represents each function in the program as a tree where each node is a *region*. A region encompasses a sub-graph of the program control flow graph (CFG) with a single entry and a single exit.



(a) `main` pseudo-code. **(b)** `main` region tree. **(c)** Loop splitting of r_2.

(d) `f` pseudo-code. **(e)** `f` region tree. **(f)** Loop tiling of r_4^f.

Figure 4 Region representation ($\rightarrow \equiv$ parent-child / $\dashrightarrow \equiv$ sequential regions).

A leaf node in the region-tree is denoted as a *trivial region* and each trivial region comprises a single basic block or a single function call. Two regions r_1 and r_2 are *sequentially-composed* if the exit of r_1 is the entry of r_2. An internal node in the region-tree is a non-trivial region that can represent a loop, a condition, or a maximal set of sequentially-composed regions (i.e. a sequential region). A non-trivial region r_i is the *parent* of region r_j if r_i is the closest region containing r_j. Each loop region has one child that represents a single iteration of the loop. The top level region r_0^f of function f can either be a basic block or a sequential region. If r_0^f is sequential, then the last region in its children sequence must be a basic block that returns from f. Each region r in the region tree has WCET t_r and a data footprint.

Figure 4 shows an example of a program with two functions: `main()` in Figure 4a and `f()` in Figure 4d. Figure 4b shows the region tree of `main()`. Region r_0, which is the top level region of `main()`, is a sequential region with regions r_1 to r_4 as its children. Region r_2 is a loop with child r_5 representing one iteration. All leaf regions r_1, r_3, r_4 and r_5 are trivial regions. Region r_3 is a call to `f()` . Figure 4e is the region tree of `f()` where r_0^f is the top level region with r_1^f to r_3^f as its sequentially-composed children. Region r_2^f is an if-else statement with region r_4^f as the true path and region r_5^f as the false path.

Loop transformations can be applied to loop regions that otherwise could not fit in a segment. A loop transformation must be legal, i.e. it preserves the temporal sequence of all dependencies and hence the result of the program. We are interested in two transformations: loop splitting and loop tiling. *Loop splitting* breaks the loop into multiple loops which have the same bodies but iterate over different contiguous portions of the index range. Figure 4c shows an example of splitting loop region r_2 in `main()` that has N iterations by expanding the loop region into three nodes: pre-loop node with k_p iterations, mid-loop node with $N - k_p - k_s$ iterations, and post-loop node with k_s iterations. *Loop tiling* combines strip-mining and loop permutation of a loop nest to create tiles of loop iterations which may be executed together. A n-level tiled loop nest, which means that the n outer loops are tiled, is divided into n tiling loops that iterate over tiles and n element loops that execute a tile. Note that the data footprint of a tile is derived in terms of the tile sizes. An example for tiling a 1-level loop is depicted in Figure 4f. In the figure, r_4^f is a tiling loop region that has N_f iterations with tile size k_t. The number of tiles is $\lceil N_f/k_t \rceil$ with $M_f = \lceil N_f/k_t \rceil - 1$ complete tiles and a last tile $k_t^{last} \leq k_t$ such that $k_t^{last} = N_f - M_f * k_t$. In Figure 4f, r_t^f is the tiling loop with M_f iterations over the element loop r_e^f. Note that, adding a tiling loop adds an overhead which is represented as r_o^f; we use t_{tile} to denote the WCET of the overhead region. The last tile is separated in r_{last}^f, where a tile of size k_t^{last} is executed after all complete tiles.

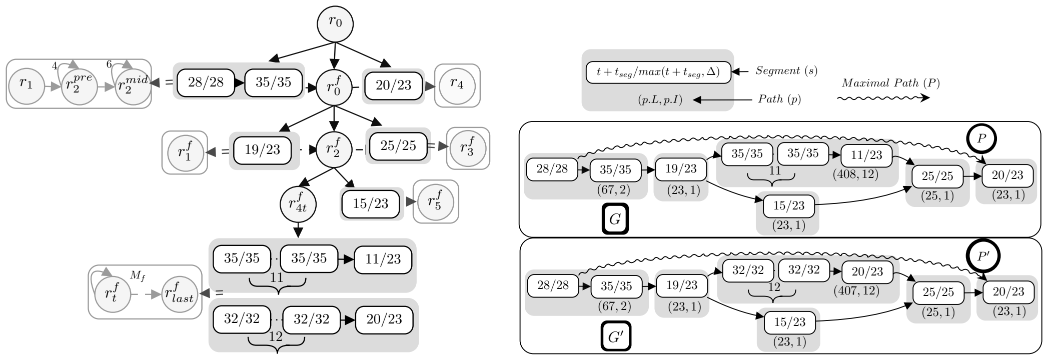
4.2 Valid Segmentation

Program segmentation is the process of assigning each part of the program code to a segment. In this paper, we restrict the parts of the program that can be assigned to a segment to be a region or a sequence of regions. A segmentation is valid if it satisfies the *footprint constraint*, the (optional) *length constraint* and the *compilation constraints*. The footprint constraint states that the footprint of each segment, i.e. the code and data of regions assigned to the segment must fit in the available SPM size. The length constraint states that the length of each segment must be at most $l^{\max}$. As discussed in Section 3, this is done to limit the blocking time imposed on higher priority tasks; setting $l^{\max} = +\infty$ is equivalent to removing the constraint. Note that creating a segment incurs a segmentation overhead t_{seg} which contributes to the segment length. That is, if region r with WCET t_r is assigned to segment s, then $s.l = \max(t_r + t_{seg}, \Delta)$. If multiple regions in sequence are assigned to a segment s, then $s.l = \max\left(\left(\sum_r t_r\right) + t_{seg}, \Delta\right)$. We further assume that the regions' WCETs satisfy the following property, which we argue is required for the WCET values to be sound:

▶ **Property 4.** *If r is a conditional region, then t_r is equal to the WCET of its longer children. If r is a sequential region or tiled loop, then its WCET is less than or equal to the sum of the WCETs of its children or tiles.*

The compilation constraints are related to how the code is modelled and transformed. A necessary compilation constraint on a segment is that the data used by the segment is known before executing the segment. This implies that if a pointer is used to access a data object in a segment, the object(s) that the pointer may refer to must be known before the segment. In this paper, we add the following compilation constraints based on the region structure to develop a systematic segmentation process:

- A region cannot be assigned to more than one segment. If a region is assigned to a segment, all its children are assigned to the same segment.
- Each basic block region must be assigned to a segment.
- For all regions except function calls, we say that a region is *mergeable* if it satisfies the footprint and length constraints and all the children of the region are mergeable.
- A function is *mergeable* if the top level region of the function is mergeable. Accordingly, a function call region is mergeable if the called function is mergeable.
- A set of mergeable regions that are sequentially-composed can be combined in a *multi-region* segment that satisfies the length and footprint constraints.
- A loop can be divided into multiple segments using loop tiling and loop splitting. A loop region is *splittable* if its child that represents a single iteration of the loop is mergeable. A loop region that represents the outermost loop of a loop nest is *tileable* if it is legal to tile and a single iteration of the innermost loop of the tiling loops is mergeable. Note that a splittable loop is always tileable based on this definition. If a loop is tiled, then each tile must be assigned to a segment that comprises that tile only and the loop node represents a sequence of segments. Tiling allows combining multiple loop iterations in a repeatable segment by inserting the segmentation instruction around the element loop.



(a) Segmented Tree. **(b)** DAGs.

Figure 5 Segmentation Example.

Based on the introduced constraints, we say that a set of regions in the tree constitute a *region sequence* if it comprises either: a single mergeable region, or a tiled loop, or a sequence of mergeable regions and/or splittable regions and tiles. Note that all regions in a sequence have the same parent. We say that a region sequence R is *maximal* if no children of its parent that is not in R can be merged with a region in R to form a segment. Our program segmentation produces a *segmented tree* $\mathcal{T}$, that is, a tree where every node is a set of segment paths $\mathcal{P}$. In particular, the segmented tree for a program is obtained by substituting region sequences in the region tree with sets of paths. A path $p \in \mathcal{P}$ for region sequence R is a sequence of segments, to which the regions and tiles in R are assigned. The segmented tree is derived inter-procedurally, i.e. for a call to a function that is not mergeable, the segmented tree of that function is duplicated in place of the call region. If there are multiple calls to the function, the segmented tree for all the calls must be the same. The segmented tree of the program is accordingly the segmented tree of the `main` function.

A segmented tree $\mathcal{T}$ implicitly generates a set $\mathcal{G}$ of segment DAGs: each DAG in $\mathcal{G}$ is constructed by taking one path out of each path set and joining them according to the segmented tree hierarchy. A maximal path in the DAG thus comprises a sequence of paths $\{p_1, p_2, ..., p_n\}$ for some n, where p_1 encompasses s^{begin} and p_n encompasses s^{end} and hence the last region in the program r_{end}. Note that for a function that has multiple calls, a path that is chosen to construct a DAG from the path set of a region sequence in the function must be used for all the function calls as the region sequence represents the same code.

Figure 5 illustrates an example segmentation of the program introduced in Figure 4. Let the maximum segment length be $l_{max} = 35$, the memory time $\Delta = 23$, the segmentation overhead $t_{seg} = 5$, and the tiling overhead $t_{tiling} = 3$. We assume for this example that all the data of the program fits in the SPM, so the footprint constraint is always satisfied. Given the times for each basic block t in Figure 4b and Figure 4d, regions $\{r_1, r_4, r_1^f, r_3^f, r_5^f\}$ are mergeable regions. However, loop regions $\{r_2, r_4^f\}$ are not mergeable. Assume that we applied loop splitting on r_2 that has 10 iterations such that it is split to two loops: pre-loop with 4 iterations and mid-loop with 6 iterations. In Figure 5a, the region sequence $\{r_1, r_2^{pre}, r_2^{mid}\}$ is replaced by a path set with a single path that has 2 segments and a total length 67. The first segment combines r_1 and r_{pre}^2 while the second segment is r_{mid}^2. As region r_3 is a call to a non-mergeable function, it is replaced by a duplicate of the segmented tree of f. The segmented tree of f has two regions r_1^f and r_3^f each wrapped in a segment. Region r_2^f is a conditional that is not mergeable, so the false path r_5^f is wrapped in a segment while the true path r_4^f which is a loop with 100 iterations is tiled. There are many possible tiling options that would satisfy the max segment length. We choose two of them based on the tiling algorithm in the next section. The first path has length $p.l = 408$ and number of segments $p.I = 12$. The first 11 segments are complete tiles each with size $k_t = 9$ and length $\max(9*3+t_{tiling}+t_{seg}, 23) = 35$, and the last segment is the last tile $k_t^{last} = 100-11*9 = 1$ with length $\max(1*3+t_{tiling}+t_{seg}, 23) = 23$. Similarly, the other path has length $p.l = 497$ and number of segments $p.I = 13$. The first 12 segments are complete tiles each with size $k_t = 8$ and with length $\max(8*3+t_{tiling}+t_{seg}, 23) = 32$, and the last segment is the last tile $k_t^{last} = 100-12*8 = 4$ with length $\max(4*3+t_{tiling}+t_{seg}, 23) = 23$. The two DAGs generated from the segmented tree are shown in Figure 5b.

4.3 Segmentation Algorithm

The example in Section 4.2 shows that different segmentation decisions can result in incomparable maximal paths according to Definition 1 as in Figure 5b: for the path P, we have $P.L = 547$, $P.I = 17$ and $P.end = 23$, while for the path P', we have $P'.L = 546$, $P'.I = 18$ and $P'.end = 23$. Since a DAG generated from the segmented tree $\mathcal{T}$ includes either P or P', the resulting two DAGs G and G' are also incomparable. This means that without considering the other tasks in the system, we cannot determine whether G or G' is better from a schedulability perspective. Hence, to guarantee that we can find an optimal segmentation for the task set, we need to consider both G and G'. On the other hand, if $G' \succeq G$, we can safely ignore G' based on Property 3. This is formally captured by the following definition.

▶ **Definition 4.** *Let $\mathcal{G}$ be the set of all valid DAGs for a program according to a set of constraints, and let $\mathcal{G}'$ be the set of DAGs returned by a segmentation algorithm for that program. We say that the algorithm preserves optimality iff for any program: $\mathcal{G}'$ is valid according to the constraints, and $\forall G \in \mathcal{G}, \exists G' \in \mathcal{G}' : G \succeq G'$.*

Based on Definition 4, a naive optimality-preserving algorithm could proceeds as follows: first, enumerate all valid DAGs in $\mathcal{G}$. Then, cut dominating DAGs based on the dominance relation. However, due to possible variations of loop tiling/splitting and multi-region segments, this is practically unfeasible as the set $\mathcal{G}$ is too large. Therefore, we propose a much faster segmentation Algorithm 1 that preserves optimality according to Definition 4 but removes dominating DAGs without enumerating $\mathcal{G}$; instead, the algorithm explores the segmented tree recursively and removes unneeded paths from the path set $\mathcal{P}$ of each region sequence R. Note that the length, footprint and compilation constraints are implied in all the following algorithms whenever a region is checked to be mergeable, splittable, or tileable and whenever a segment is checked to be valid.

Algorithm 1 starts with a call to SegmentTask function. Then Segment(r_0) is called on r_0, the top level region of `main`, hence returning the segmented subtree for the whole program. Finally, a DAG set $\mathcal{G}$ is generated from the segmented tree and returned as a result of SegmentTask. Note that if r_0 is mergeable, then the segmented tree is composed of a single, maximal region sequence R that comprises r_0 only; hence, in this case we simply return a DAG with r_0 as its single segment.

Function Segment(r) segments a subtree of the region tree and returns a segmented subtree with r as its root. The function traverses this subtree from its root r in depth-first order preserving the topological order between sequentially-composed children. If r is a sequential region, then a set of children in sequence that are mergeable or splittable loops may be combined in multi-region segments. This is achieved by adding these children to a region sequence R until a child that is not mergeable or splittable is found or until all children

Algorithm 1 Segmentation Algorithm.

```
1: function SEGMENTTASK($\tau$)
2:     if $r_0$ is mergeable then
3:         Create DAG $G$ with a single segment comprising $r_0$, return $\mathcal{G} = \{G\}$
4:     Generate DAG set $\mathcal{G}$ from $\mathcal{T}$ = SEGMENT($r_0$), return $\mathcal{G}$
5: function SEGMENT($r$)
6:     Initialize $R = \emptyset$                                      ▷ A set of sequential regions.
7:     Initialize $\mathcal{T}$ to be the subtree whose root is $r$
8:     for all $r_c \in children(r)$ do
9:         if $r$ is sequential and $r_c$ is mergeable or splittable loop then
10:            Add $r_c$ to $R$
11:        else if $r_c$ is mergeable then                              ▷ $r$ is not sequential
12:            Replace $r_c$ with $\mathcal{P} = \{p\}$, where $p$ is single-segment path
13:        else
14:            Replace regions in $R$ with SEGMENTSEQUENCE($R$), empty $R$
15:            if $r_c$ is a tileable loop then
16:                Replace $r_c$ with TILE($r_c$).
17:            else if $r_c$ is a call to $f$ then
18:                Replace $r_c$ with SEGMENT($r_0^f$)
19:            else
20:                Replace $r_c$ with SEGMENT($r_c$)
21:    If $R \neq \emptyset$, replace regions in $R$ with SEGMENTSEQUENCE($R$)
22:    return $\mathcal{T}$
```

are traversed. Note that based on the compilation constraints, no children outside R can be combined with a region in R to form a segment; hence, the obtained R is maximal. Then, the regions in R are replaced by a set of valid paths $\mathcal{P}$ that are generated using function SEGMENTSEQUENCE(R). If r is not sequential, a mergeable child r_c is directly replaced by a path of one segment, as r_c is a maximal region sequence by itself. If child r_c is not mergeable, then it has three cases: 1) r_c is a tileable loop, then a set of paths are generated by tiling the loop using function TILE(r_c); 2) r_c is a call to a function f, then the segmented tree of f is duplicated in place of r_c; 3) r_c is not a tileable loop or a function call, then it is segmented by recursively calling SEGMENT(r_c).

Since Algorithm 1 depends on SEGMENTSEQUENCE and TILE, we first state a key property of both functions, which will be detailed in Algorithms 2 and 3. Since the functions return a path set $\mathcal{P}$, we begin by defining a concept of domination among paths and path sets.

▶ **Definition 5.** *Given two paths p, p', we say that p' dominates p and write $p' \succeq p$ iff: $p'.L \geq p.L$ and $p'.I \geq p.I$.*

Note that Definition 5 is similar to Definition 1 for maximal paths, except that we do not consider the last segment, since its length is only relevant in the case of s^{end}. We can relate the two definitions through the following lemma.

▶ **Lemma 6.** *Consider two maximal paths $P = \{p_1, ..., p_k, ..., p_n\}, P' = \{p'_1, ..., p'_k, ..., p'_n\}$ obtained by joining n paths. If $p'_n.end = p_n.end$ and $\forall k = 1...n : p'_k \succeq p_k$, then $P' \succeq P$.*

Proof. Note by construction $P.L = \sum_{k=1...n} p_k.L, P'.L = \sum_{k=1...n} p'_k.L$. From $p'_k \succeq p_k$ it follows $p'_k.L \geq p_k.L$, hence $P'.L \geq P.L$. In the same manner, we obtain $P'.I \geq P.I$. Finally, since p'_n and p_n contain the last segments in their corresponding maximal paths P' and P, $p'_n.end = p_n.end$ implies $P'.end = P.end$. Then by Definition 1 we have $P' \succeq P$. ◀

▶ **Definition 7.** *Given two path sets $\mathcal{P}, \mathcal{P}'$ for the same region sequence R, we say that $\mathcal{P}'$ dominates $\mathcal{P}$ and write $\mathcal{P}' \succeq \mathcal{P}$ iff: $\forall p' \in \mathcal{P}', \exists p \in \mathcal{P} : p' \succeq p$, and if $r_{end} \in R$, then $p'.end = p.end$.*

▶ **Property 5.** *Let R be a region sequence and $\mathcal{P}'$ the set of all valid paths for R. Then* SegmentSequence(R) *returns a set of paths $\mathcal{P}$ such that $\mathcal{P} \subseteq \mathcal{P}'$ and $\mathcal{P}' \succeq \mathcal{P}$.*

▶ **Property 6.** *Let r_c be a tilable loop with N_r iterations and $\mathcal{P}'$ the set of all valid paths for r_c. Then* Tile(r_c) *returns a set of paths $\mathcal{P}$ such that $\mathcal{P} \subseteq \mathcal{P}'$ and $\mathcal{P}' \succeq \mathcal{P}$.*

Intuitively, this implies that Tile and SegmentSequence return a set of best path for the corresponding region sequence / loop. Based on Properties 5, 6, we next prove in Theorem 11 that Algorithm 1 preserves optimality. We start by showing that the algorithm can stop traversing the tree at mergeable regions, i.e. if a region is mergeable we do not need to segment its children.

▶ **Lemma 8.** *Consider a region r that is either mergeable (possibly after splitting) or a tile, and a valid DAG G' for the program where r is not assigned to a segment. Then there exists a valid DAG G where r is assigned to a segment and $G' \succeq G$.*

Proof. Consider any maximal path $\mathcal{P}'$ in G' of the form $P' = \{p_{begin}, p', p_{end}\}$, where p' is a path through the descendants of r (note that no path of the form $P' = \{p_{begin}, p'\}$ can exist, since the last region of `main` r_{end}, and thus the program, is a basic block with no descendants). Note that in case of conditional regions, there could be multiple such p', and hence maximal paths $\mathcal{P}'$ with the same p_{begin} and p_{end}. **Example:** consider the conditional region r_2^f in Figure 5; a valid DAG G' has two maximal paths P' through the descendants of r_2^f: one for the true path, and one for the false path.

Now consider a valid DAG G obtained by replacing all such maximal paths P' with a path $P = \{p_{begin}, p, p_{end}\}$, where p comprises a single segment that includes r only; note the DAG is valid since r is mergeable or a tile. Since $p.I = 1$, it immediately follows $p'.I \geq p.I$. Based on Property 4, there must also exist one path p' with $p'.L \geq p.L$. By Lemma 6, we then proved that there must exist a maximal path P' such that $P' \succeq P$. By definition, this implies $G' \succeq G$, completing the proof. ◀

▶ **Lemma 9.** *Consider a segmented tree $\mathcal{T}$ where all region sequences are maximal, and the path set $\mathcal{P}'$ for each region sequence R includes all valid paths for R. Then the DAG set generated from $\mathcal{T}$ preserves optimality.*

Proof. First note that by definition, each path $p \in \mathcal{P}'$ is a sequence of segments, to which the regions and tiles in R are assigned, i.e. $\mathcal{P}'$ does not include (still valid) paths that would segment the descendants of a region in R.

By the compilation constraints and definition of maximal region sequence R, it follows that any region that is in R cannot be merged in a segment with a region that is not in R. Hence, any valid maximal path for the program that includes segments of n region sequences can be constructed by joining n paths: $P = \{p_1, ..., p_k, ..., p_n\}$. By Lemma 8, we can restrict each p_k to be a path in $\mathcal{P}'$ (where each region $r \in R$ is assigned to a segment) and for each valid DAG G', generate a DAG G such that $G' \succeq G$. By Definition 4, this means that generating DAGs from $\mathcal{T}$ preserves optimality. ◀

Lemma 9 shows that to preserve optimality, it is sufficient to return a single segmented tree with maximal region sequences, which is what Algorithm 1 builds by construction. Finally, we show that instead of generating the set $\mathcal{P}'$ of all valid paths for each region sequence R, we can use a dominated subset $\mathcal{P}$.

Algorithm 2 1-Level Tiling.

```
1: function TILE($r$)
2:     Compute $k_t^{max}$, $\mathcal{P} = \emptyset$
3:     for all $k_t \leq k_t^{max}$ do
4:         Generate $p(k_t)$ and add it to $\mathcal{P}$ if it is valid
5:     Filter $\mathcal{P}$ by removing dominating paths based on Definition 5
6:     return $\mathcal{P}$
```

▶ **Lemma 10.** *Consider a segmented tree $\mathcal{T}$ as in Lemma 9. Let $\overline{\mathcal{T}}$ denote the segmented tree obtained by replacing, for each maximal region sequence R in $\mathcal{T}$, the set $\mathcal{P}'$ of all valid paths with a set $\mathcal{P}$ such that $\mathcal{P} \subseteq \mathcal{P}'$ and $\mathcal{P}' \succeq \mathcal{P}$. Then the DAG set generated from $\overline{\mathcal{T}}$ preserves optimality.*

Proof. Since for all regions $\mathcal{P} \subseteq \mathcal{P}'$, DAGs generated from $\overline{\mathcal{T}}$ are still valid. Consider any DAG G' generated from $\mathcal{T}$, and a maximal path P' of G' through n region sequences: $P' = \{p'_1, ..., p'_k, ..., p'_n\}$. Since for all regions $\mathcal{P}' \succeq \mathcal{P}$, then for every p'_k there exists another path p_k in $\overline{\mathcal{T}}$ such that $p'_k \succeq p_k$, and furthermore $p'_n.end = p_n.end$ since the last region sequence in any maximal path must include the last region in the program r_{end}. By Lemma 6, this means that we can find a maximal path $P = \{p_1, ..., p_k, ..., p_n\}$ for $\overline{\mathcal{T}}$ such that $P' \succeq P$. Since this is true for any maximal path through a given set of region sequences, and both $\mathcal{T}$ and $\overline{\mathcal{T}}$ have the same set of (maximal) region sequences, we have shown that $\overline{\mathcal{T}}$ can generate a DAG G such that for every maximal path $P \in G$, there is a maximal path $P' \in G'$ with $P' \succeq P$. This implies $G' \succeq G$, and since by Lemma 9 $\mathcal{T}$ preserves optimality, it thus follows that the DAG set generated from $\overline{\mathcal{T}}$ also preserves optimality according to Definition 4. ◀

▶ **Theorem 11.** *If Properties 5, 6 hold, Algorithm 1 preserves optimality based on the footprint, length and compilation constraints.*

Proof. By construction, the algorithm creates a segmented tree $\mathcal{T}$ of maximal region sequences. Let $\mathcal{P}'$ to denote the set of all valid paths for each region R. The actual path set $\mathcal{P}$ used for R is generated at line 12, 16 or 21. At line 12, region r_c is not sequential. Hence, $R = \{r_c\}$ is a maximal region. The algorithm generates a path comprising a single segment for r_c, which is the only valid path for R; thus we have $\mathcal{P} = \mathcal{P}'$. At line 16 and 21, the path set $\mathcal{P}$ is generated by calling either SEGMENTSEQUENCE(R) or TILE(r_c); by Properties 5, 6 and Lemma 10, in both cases $\mathcal{P} \subseteq \mathcal{P}'$ and $\mathcal{P}' \succeq \mathcal{P}$ hold. In summary, Lemma 10 applies to all maximal regions, hence the algorithm preserves optimality. ◀

4.3.1 Tiling Algorithm

We now discuss how to optimize the tile size for a 1-level tiled loop r, similarly to the example in Section 4.1. Note that while we present the case of 1-level tiling for simplicity, in practice 2-level tileable loops are common in embedded programs. Hence, our framework also implements a more general algorithm that can find tile sizes for 2-level tiling; due to space limitations, we detail it in the provided technical report [26].

Given an execution time for one iteration of t_1, a number of iterations N_r and a tile size k_t with $M = \lceil N_r/k_t \rceil - 1$ and $k_t^{last} = N_r - M * k_t$, tiling results in a path $p(k_t)$ comprising M segments of length $\max(\Delta, k_t * t_1 + t_{tile} + t_{seg})$, and one segment of length $\max(\Delta, k_t^{last} * t_1 + t_{tile} + t_{seg})$. Algorithm 2 simply iterates over k_t starting with $k_t^{\max}$, the maximum value of k_t such that the length of any segments in $p(k_t)$ is less than or equal to

$l_{\max}$ and its footprint is less than or equal to the SPM size. It then generates each path $p(k_t)$ and adds it to $\mathcal{P}$ if it is valid. Finally, based on Definition 5, it removes any path $p(k'_t)$ from $\mathcal{P}$ if there exists another path $p(k_t) \in \mathcal{P}$ with $p(k'_t) \succeq p(k_t)$. The following lemma then easily follows.

▶ **Lemma 12.** *Algorithm 2 satisfies Property 6.*

Proof. First note that r cannot be r_{end}, since the last region in a program must be a basic block and not a loop. By the compilation constraints, every generated tile must be assigned to a segment that comprises the tile only. Then by the footprint and length constraints, any path $p(k_t)$ with $k_t > k_t^{max}$ cannot be valid. Also since the algorithm adds all valid paths $p(k_t)$ with $k_t \leq k_t^{max}$ to $\mathcal{P}$, before executing line 5, $\mathcal{P}$ contains all valid paths for r. Since furthermore the filtering on line 5 respects Definition 7, Property 6 holds. ◀

4.3.2 Region Sequence Segmentation

Next, we consider Algorithm 3 that generates a path set $\mathcal{P}$ from a region sequence R. The algorithm iterates over each region r in R, incrementally constructing a set of paths $\bar{\mathcal{P}}$ for the sub-sequence that includes all regions from the beginning of R up to r. For simplicity of notation, for a path $\bar{p} \in \bar{\mathcal{P}}$, we use $\bar{p}.t_{end}$ to denote the WCET of the regions included in the last segment of $\bar{p}$, such that $\bar{p}.end = \max(\Delta, \bar{p}.t_{end} + t_{seg})$. At each step of the algorithm, for a region r with computation time t_r, a new set of paths is constructed by taking each path $\bar{p}$ in $\bar{\mathcal{P}}$ and adding r to it. Note that when doing so, two new paths might be generated in the following way:

1. Add a new segment comprising r to $\bar{p}$ to construct a new path $\bar{p}_n$ such that $\bar{p}_n.I = \bar{p}.I + 1$, $\bar{p}_n.t_{end} = t_r$, and $\bar{p}_n.L = \bar{p}.L + \bar{p}_n.end$. Note that $\bar{p}_n$ is always valid, since r is mergeable.
2. Add r to the last segment [3] of $\bar{p}$ to construct a new path $\bar{p}_m$ such that $\bar{p}_m.I = \bar{p}.I$, $\bar{p}_m.t_{end} = \bar{p}.t_{end} + t_r$, and $\bar{p}_m.L = \bar{p}.L - \bar{p}.end + \bar{p}_m.end$. Note that $\bar{p}_m$ might not be valid according to the constraints; so, it is only added to the new set of paths if valid.

The process continues until after we reach the last region in R; at that point, we return the final path set $\bar{\mathcal{P}}$.

Note that if we do not apply loop splitting, then there are 2^{m-1} possible paths for m mergeable regions in sequence. An enumeration of these ways is possible as m is usually small. However, adding loop splitting and tiling greatly increases the number of paths. We tackle this complexity in the extended technical report [26] by introducing a set of conditions that allows us to prune the generated paths from $\bar{\mathcal{P}}$ at each step while preserving optimality and hence improve the segmentation time.

In details, Algorithm 3 traverses the regions in R in topological order. If the current region r is not a splittable loop, then new paths $\bar{p}_m$ and $\bar{p}_n$ are generated by adding r to each previous path in function CreatePaths. The new paths are placed in $\bar{\mathcal{P}}_{next}$, before becoming the set of paths $\bar{\mathcal{P}}$ at the next iteration. If r is a splittable loop, then before generating a new path, the loop must be split to pre-loop region r_p, mid-loop region r_t and post-loop region r_s. Note that all combinations of pre-loop k_p and post-loop k_s splits are visited. For each (k_p, k_s), paths $\bar{\mathcal{P}}_{loop}$ for r_p are generated using CreatePaths, then r_t is tiled and each tile path is sequenced with the paths in $\bar{\mathcal{P}}_{loop}$. Then, paths are created using k_s for all paths in $\bar{\mathcal{P}}_{loop}$. All paths $\bar{\mathcal{P}}_{loop}$ are finally accumulated in $\bar{\mathcal{P}}_{next}$. After traversing all regions in R, the paths in $\bar{\mathcal{P}}$ are filtered using Definition 5 if $r_{end} \notin R$. Otherwise, all the generated paths are kept. Finally, the path set $\bar{\mathcal{P}}$ for R is returned.

[3] Note that adding a region r to a segment s implies that the footprint of the resulting segment is the union of the footprints of r and of the regions in s.

Algorithm 3 Segment a Region Sequence.

```
Require: A region sequence $R$ and the last basic block region $r_{end}$
 1: function SegmentSequence($R$)
 2:     $\bar{\mathcal{P}} = \{\bar{p} = \emptyset\}$, $\bar{\mathcal{P}}_{next} = \emptyset$
 3:     for all $r \in R$ do                    ▷ Traverse the sequence in topological order.
 4:         if $r$ is a splittable loop then
 5:             for all $k_p$, $k_s$ do:
 6:                 Split $r$ to $r_p, r_t$ and $r_s$
 7:                 $\bar{\mathcal{P}}_{loop}$ = CreatePaths($r_p, \bar{\mathcal{P}}$)
 8:                 $\bar{\mathcal{P}}_{loop}$ = generate all paths by joining $\bar{\mathcal{P}}_{loop}$ with Tile($r_t$)
 9:                 $\bar{\mathcal{P}}_{loop}$ = CreatePaths($r_s, \bar{\mathcal{P}}_{loop}$)
10:                 $\bar{\mathcal{P}}_{next} = \bar{\mathcal{P}}_{next} \bigcup \bar{\mathcal{P}}_{loop}$
11:         else                    ▷ $r$ is a mergeable region that is not a splittable loop
12:             $\bar{\mathcal{P}}_{next}$ = CreatePaths($r, \bar{\mathcal{P}}$)
13:         $\bar{\mathcal{P}} = \bar{\mathcal{P}}_{next}$, $\bar{\mathcal{P}}_{next} = \emptyset$
14:     If $r_{end} \notin R$, Filter $\bar{\mathcal{P}}$ by removing dominating paths based on Definition 5
15:     return $\bar{\mathcal{P}}$
16: function CreatePaths($r, \bar{\mathcal{P}}$)
17:     $\bar{\mathcal{P}}_{tmp} = \emptyset$
18:     for all $\bar{p}$ in $\bar{\mathcal{P}}$ do
19:         Create $\bar{p}_m$ by adding $r$ to the last segment in $\bar{p}$, add $\bar{p}_m$ to $\bar{\mathcal{P}}_{tmp}$ if valid
20:         Create $\bar{p}_n$ by adding new segment using $r$ to $\bar{p}$, add $\bar{p}_n$ to $\bar{\mathcal{P}}_{tmp}$
21:     return $\bar{\mathcal{P}}_{tmp}$
```

▶ **Lemma 13.** *Algorithm 3 satisfies Property 5.*

Proof. By construction, the algorithm explores all possible combinations for the parameters of a splittable loop, all possible valid assignments of sequential regions in R to segments, and tiling decisions based on Algorithm 2. Therefore, it must hold $\mathcal{P} \subseteq \mathcal{P}'$. It remains to show that if a path p' for R is discarded (i.e., the path is in $\mathcal{P}'$ but not in $\mathcal{P}$), then there exists a path p such that $p' \succeq p$, and if $r_{end} \in R$, then $p'.end = p.end$. A path in $\mathcal{P}'$ can be discarded if: (1) Algorithm 2 removes a tiling solution; (2) the path is filtered based on Definition 5.

Case (1): Assume that Algorithm 2 removes a path p'_t from the returned path set for a tiled loop; by Property 6, it must return another path p_t such that $p'_t \succeq p_t$. Then if we consider any path p' for R of the form $p' = \{p_1, ..., p'_t, ..., p_n\}$, there must exist another path $p = \{p_1, ..., p_t, ..., p_n\}$, and by Lemma 6, it must hold $p' \succeq p$. Next consider the case $r_{end} \in R$: by the compilation constraints, a tiled loop cannot generate the last segment in the program (the last region is a basic block, and tiles cannot be merged with another region). Therefore p_n is not empty and it must hold $p'.end = p.end = p_n.end$.

Case (2): Note this applies only if r_{end} is not contained in R. It thus suffices to notice that by Definition 5 $p' \succeq p$ must hold. ◀

5 Optimal Task Set Segmentation

Based on the analysis Properties 2, 3 introduced in Section 3.1 and segmentation Algorithm 1, we now show that we can obtain an optimal task set segmentation using Algorithm 4. The algorithm recursively calls function SegmentTaskSet for task index i from 1 to N

Algorithm 4 Task Set Segmentation.

```
Require: Task set $\Gamma$, source code for each task in $\Gamma$
1: SegmentTaskSet($\Gamma$, $i, +\infty, \emptyset$)
2: Terminate with FAILURE
3: function SegmentTaskSet($i, l^{\max}, \{G_1, \ldots, G_{i-1}\}$)
4:     Generate $\mathcal{G}_i$ = SegmentTask($\tau_i$) using Algorithm 1 based on length constraint $l^{\max}$
5:     if $i < N$ then
6:         for all $G_i \in \mathcal{G}_i$ do
7:             Compute the maximum value $\overline{l_i^{l\max}}$ of $l_i^{l\max}$ based on analysis
8:             SegmentTaskSet($\Gamma, i+1, \min\left(l^{\max}, \overline{l_i^{l\max}}\right), \{G_1, \ldots, G_i\}$)
9:     else
10:        for all $G_N \in \mathcal{G}_i$ do
11:            If analysis returns schedulable on $\{G_1, \ldots, G_N\}$, terminate with SUCCESS
```

by keeping track of the DAGs $G_1, \ldots G_{i-1}$ selected for the previous tasks. The function maintains a maximum segment length $l^{\max}$, which is provided as a constraint to Algorithm 1 to generate a DAG set $\mathcal{G}_i$ for τ_i. If $i < N$, the function iterates over all possible $G_i \in \mathcal{G}_i$; the schedulability analysis is used to determine $\overline{l_i^{l\max}}$, the maximum schedulable value of $l_i^{l\max}$, and the function is then invoked recursively for task $i+1$ after updating $l^{\max}$ based on the computed value. Note that if G_i is not schedulable, then we obtain $l^{\max} < 0$; hence, there will be no valid DAG for τ_{i+1} ($\mathcal{G}_i$ is empty), and the recursive call will immediately return. Once we reach task τ_N, the function checks if τ_N is schedulable for any DAG $G_N \in \mathcal{G}_N$, in which case we terminate by finding a solution $\{G_1, \ldots, G_N\}$. If no solution can be found, the algorithm eventually terminates on Line 2.

We now prove the optimality of Algorithm 4 for a program segmentation obeying the footprint and compilation constraints in Section 4.2. We start with a corollary.

▶ **Corollary 14.** *Consider two DAGs G_j, G'_j for task τ_j where $1 \le j \le i$ and $G'_j \succeq G_j$. Let $\overline{l_i^{l\max}}, \overline{l_i^{l\max}}'$ be the maximum value of $l_i^{l\max}$ under which τ_i is schedulable for G_j and G'_j, respectively, according to an analysis satisfying Properties 2, 3. Then $\overline{l_i^{l\max}} \ge \overline{l_i^{l\max}}'$.*

Proof. By Property 2, $\overline{l_i^{l\max}}$ and $\overline{l_i^{l\max}}'$ are well defined (i.e., there must exist such maximum values). Since τ_i is schedulable with $l_i^{l\max} \le \overline{l_i^{l\max}}'$ for G'_j, based on Property 3 it is also schedulable with $l_i^{l\max} \le \overline{l_i^{l\max}}'$ for G_j; this implies $\overline{l_i^{l\max}} \ge \overline{l_i^{l\max}}'$. ◀

▶ **Theorem 15.** *Algorithm 4 is an optimal segmentation algorithm for a conditional PREM task set Γ according to any (sufficient) schedulability analysis satisfying Properties 2, 3 and based on the footprint and compilation constraints.*

Proof. We have to show that if there exists a set of segment DAGs $G'_1, \ldots, G'_N$ for Γ that is valid according to the footprint and compilation constraints and is schedulable according to the analysis, then Algorithm 4 finds a (same or different) DAG set $G_1, \ldots, G_N$ that is also valid and schedulable.

By induction on the index i. We show that for every i, there exists a recursive call sequence of function SegmentTaskSet that results in a DAG set $G_1, \ldots G_i$ such that $G'_j \succeq G_j$ for every $j = 1 \ldots i$; by Property 3 with $i = N$, this proves the theorem (note that τ_N is schedulable by Property 3, while all other tasks are schedulable because the recursion reaches G_N). We also show that for every $j = 1 \dots i$ it holds $\overline{l_j^{l\max}}' \leq \overline{l_j^{l\max}}$, where $\overline{l_j^{l\max}}'$ is the maximum schedulable value of $l_j^{l\max}$ computed by the analysis with DAGs $G'_1, \dots, G'_j$, and $\overline{l_j^{l\max}}$ is the same value for DAGs $G_1, \dots, G_j$.

Base Case ($i = 1$): note $l^{\max} = +\infty$, meaning that only the footprint and compilation constraints apply when invoking Algorithm 1. Hence, by Definition 4 the algorithm must find a DAG $G_1 \in \mathcal{T}_1$ such that $G'_1 \succeq G_1$. By Corollary 14, this also implies $\overline{l_1^{l\max}}' \leq \overline{l_1^{l\max}}$.

Induction Step ($i = 2 \dots N$): consider the recursive call sequence that results in $G'_j \succeq G_j$ and $\overline{l_j^{l\max}}' \leq \overline{l_j^{l\max}}$ for each $j = 1 \dots i-1$ (such sequence exists by induction hypothesis); we have to show that we can find a DAG $G_i \in \mathcal{G}_i$ such that $G'_i \succeq G_i$ and $\overline{l_i^{l\max}}' \leq \overline{l_i^{l\max}}$.

Based on the recursive call at line 7 of the algorithm, it must hold: $l^{\max} = \min_{j=1}^{i-1} \overline{l_j^{l\max}}$. Define $l^{\max\prime} = \min_{j=1}^{i-1} \overline{l_j^{l\max}}'$; since the task set is schedulable for $G'_1, \dots, G'_N$, the maximum length of any segment in G'_i is at most $l^{\max\prime}$. By induction hypothesis, it must be $l^{\max\prime} \leq l^{\max}$, which means that the maximum segment length in G'_i is also no larger than $l^{\max}$. Hence, if we define $\mathcal{G}_i$ to be the set of all valid DAGs for a program according to the constraints with maximum segment length $l^{\max}$, we have $G'_i \in \mathcal{G}_i$. By Definition 4, this implies that Algorithm 1 finds a valid DAG G_i with maximum segment length $l^{\max}$ such that $G'_i \succeq G_i$. $\overline{l_i^{l\max}}' \leq \overline{l_i^{l\max}}$ then again follows by Corollary 14. ◀

Complexity. Since it iterates over all $G_i \in \mathcal{G}_i$, Algorithm 4 is exponential. Intuitively, it might seem sufficient to only use the DAG in $\mathcal{G}_i$ that results in the highest value of $\overline{l_i^{l\max}}$; however, given two DAGs G_i and G'_i with $\overline{l_i^{l\max}} \geq \overline{l_i^{l\max}}'$, it might be that $L_i^{\max} \geq L_i^{'\max}$, that is, G_i results in larger slack for τ_i, but it increases the interference caused by τ_i on lower priority tasks based on Equations 4. In this case, we have to test both G_i and G'_i. However, if $L_i^{\max} \leq L_i^{'\max}$, then we can safely ignore G'_i. As we show in Section 6, in practice this results in an acceptable runtime considering the algorithm is an offline optimization.

Composability and Generality. As we (re-)compile all tasks, our approach requires the source code of all applications in the system. Since Algorithm 4 segments tasks in priority order, any code change in a program will not affect higher priority tasks; however, it might force a recompilation of all lower priority tasks. This might be undesirable, especially if the priority ordering does not match criticality levels. Therefore, in Section 6 we also explore a simpler and faster (but non-optimal) heuristic that uses the same value of $l^{\max}$ for all tasks, thus ensuring that each program can be compiled independently. In this sense, we would like to stress that even if the optimality of Algorithm 4 depends on analysis Properties 2, 3, our compiler framework in conjunction with Algorithm 1 can still be used to produce a set of valid program segmentations for any PREM-based system.

6 Evaluation

We implemented our segmentation framework using LLVM to analyze and generate the region trees for the program as in [24], and estimate the data footprint for each part of the program. Poly [14] is used to handle loop transformations. For code generation, we target a simple MIPS processor model with 5-stages pipeline and no branch prediction. We assume that there are data SPM, and code SPM and that the task code fits in the code SPM. Note that the WCET of each region in a program is statically estimated using the simple MIPS processor model similar to [25]. For the data SPM, we vary its size from 4 kB to 512 kB. For

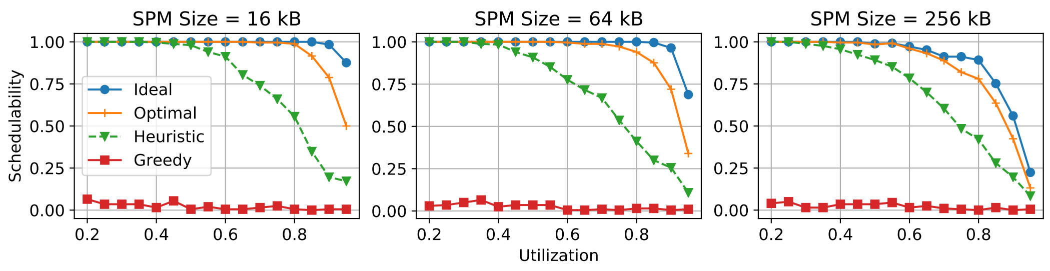
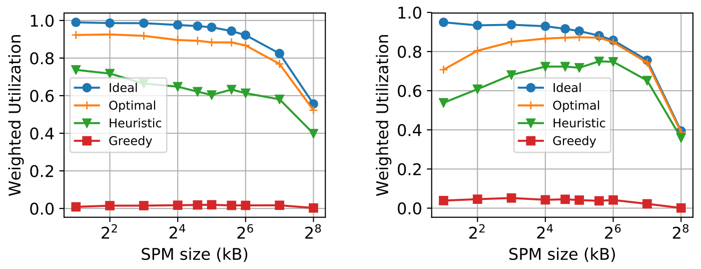


Figure 6 Schedulability vs Utilization.



(a) $t_{seg} = 100$. **(b)** $t_{seg} = 1000$, footprint>24 kB.

Figure 7 Weighted Utilization VS SPM Size.

Table 1 Benchmarks (LOC: lines of code).

Benchmark	Suite	LOC	Data(B)
adpcm_dec	TACLeBench	476	404
cjpeg_transupp	TACLeBench	474	3459
fft	TACLeBench	173	24572
compress	UTDSP	131	136448
lpc	UTDSP	249	8744
spectral	UTDSP	340	4584
disparity	CortexSuite	87	2704641

the memory transfer, we assume that the DMA speed is 1 cycle per word (4 bytes) [4]. The segmentation overhead t_{seg} includes the DMA intialization and the context switching, and it is assumed to be 100 cycles.

We evaluate the segmentation and scheduling algorithms using a set of synthetic and real benchmarks. We used applications from UTDSP [29], TACLeBench [11] and CortexSuite [28] benchmark suites. The application are chosen to represent a variety of sizes, complexities and data footprints (see Table 1). The applications are used to generate sets of random tasks. Each task set is composed of a random number of tasks between 4 and 12 tasks. Given a system utilization and the number of tasks, the utilization of each task is generated with uniform distribution [6], and then a period is assigned to each task. The period of τ_i is computed as $u_i * c_i$ where u_i is the generated utilization and c_i is the WCET of the application if executed without premption from the SPM. We assume deadlines equal to periods. Schedulability tests are conducted for 250 task sets.

We report the results in terms of the system schedulability and the weighted utilization. The *system scheduability* is the proportion of the schedulable task sets out of the total tested task sets. We define the *weighted utilization* μ of a system as: $\mu = \frac{\sum_u sched(u)*u}{\sum_u u}$ where $sched(u)$ is the system schedulability for system utilization u. We compare our optimal algorithm with ideal, greedy and heuristic algorithms. The *ideal* algorithm assumes no restriction on SPM size and that the program code can be segmented at any arbitrary point without any increased overhead. Hence, the only constraint is l_{max} which is produced from Algorithm 4 [5]. The *greedy* and *heuristic* algorithms do not depend on Algorithm 4 to drive

[4] In the extended technical report [26], we discuss the effect of the DMA speed on the system schedulability

[5] Note that the ideal algorithm is still compliant with PREM, i.e. the next segment has to be decided and loaded while the current segment is executing. Hence, to our understanding we cannot employ existing scheduling analyses for limited-preemptive task sets [8].

the segmentation of each task based on the schedulability analysis. The greedy algorithm resembles the algorithm used in [20] and assumes $l_{max} = \infty$ for all tasks. The heuristic algorithm uses the same l_{max} for all tasks by varying l_{max} between Δ and $10 * \Delta$ with step $0.5 * \Delta$, and picking the value of l_{max} that achieves the highest weighted utilization.

Figure 6 shows the system schedulability for the four algorithms for SPM sizes of 16, 64 and 256 kB. The graphs show that the optimal algorithm performs much better than the greedy and the heuristic algorithms and close to the ideal algorithm for different SPM sizes. This is confirmed in Figure 7a that shows the weighted utilization for the compared algorithms for SPM sizes between 4 kB and 512 kB. Note that the ideal algorithm may suffer from segmentation overhead, the interference and blocking overhead from other tasks in the system, and also segment under-utilization. This leads to lower schedulability at high system utilization.

We can notice in Figure 7a that the weighted utilization does not increase as SPM size increases. This might be counter-intuitive as increasing the SPM size allows more data to be loaded for each segment which leads to decreased segmentation overhead. However, the tasks suffer from a higher under-utilization penalty as Δ increases. The second effect is dominant since the segmentation overhead is relatively small and 4 benchmarks have data footprints of less than 8kB. For this reason, we show in Figure 7b the weighted utilization using only applications with data footprint greater than 24 kB and $t_{seg} = 1000$. The figure shows that the system schedulability ascents at first and then declines around SPM size of 48 kB.

The segmentation algorithm takes a few seconds to finish with a maximum of a minute compared to few hours for the naive segmentation algorithm with exhaustive search. Running the scheduling algorithm for one of the tested task sets takes an average of a minute to segment the tasks and apply the schedulability test with a maximum of few minutes. In the extended technical report [26], we discuss how the algorithm time scales with the number of tasks in a task set in more details.

7 Conclusions and Future Work

PREM-based scheduling schemes have recently attracted significant attention in the literature, but to make the approach applicable to industrial practice, there is a stringent need for automated tools. To this end, we have proposed a compiler-level framework that automatize the process of analyzing a program and transforming it into a conditional sequence of PREM segments. Furthermore, for the case of fixed-priority partitioned scheduling with fixed-length memory phases, which has been fully implemented and tested in [27], we have shown that it is possible to find optimal segmentation decisions within reasonable time for realistic programs.

This work could be extended in two main directions: first, by applying it to other PREM-based scheduling schemes. Note that since searching for an optimal segmentation solution might become too expensive, we might have to resort to a heuristic instead. Second, by extending it to other task and platform models. In particular, we are highly interested in looking at parallel tasks executed on heterogeneous multicore devices.

References

1 Ahmed Alhammad and Rodolfo Pellizzoni. Schedulability analysis of global memory-predictable scheduling. In *Proceedings of the 14th International Conference on Embedded Software - EMSOFT '14*, New York, New York, USA, 2014. ACM Press.

2 Ahmed Alhammad and Rodolfo Pellizzoni. Time-predictable execution of multithreaded applications on multicore systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*, New Jersey, 2014. IEEE Conference Publications.

3 Ahmed Alhammad, Saud Wasly, and Rodolfo Pellizzoni. Memory efficient global scheduling of real-time tasks. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2015.

4 Matthias Becker, Dakshina Dasari, Borislav Nicolic, Benny Akesson, Vincent Nelis, and Thomas Nolte. Contention-Free Execution of Automotive Applications on a Clustered Many-Core Platform. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2016.

5 E. Bini and G.C. Buttazzo. Schedulability analysis of periodic fixed priority systems. *IEEE Transactions on Computers*, 53(11), 2004.

6 Enrico Bini and Giorgio C. Buttazzo. Measuring the Performance of Schedulability Tests. *Real-Time Systems*, 30(1-2), 2005.

7 Paolo Burgio, Andrea Marongiu, Paolo Valente, and Marko Bertogna. A memory-centric approach to enable timing-predictability within embedded many-core accelerators. In *2015 CSI Symposium on Real-Time and Embedded Systems and Technologies (RTEST)*. IEEE, 2015.

8 Giorgio C. Buttazzo, Marko Bertogna, and Gang Yao. Limited Preemptive Scheduling for Real-Time Systems. A Survey. *IEEE Transactions on Industrial Informatics*, 2013.

9 Nicola Capodieci, Roberto Cavicchioli, Paolo Valente, and Marko Bertogna. SiGAMMA: Server based integrated GPU Arbitration Mechanism for Memory Accesses. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems - RTNS '17*, New York, New York, USA, 2017. ACM Press.

10 Guy Durrieu, Madeleine Faugère, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and W. Puffitsch. Predictable Flight Management System Implementation on a Multicore Processor. *{Embedded Real Time Software (ERTS'14)}*, February 2014.

11 Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. *DROPS-IDN/6895*, 55, 2016.

12 Bjorn Forsberg, Luca Benini, and Andrea Marongiu. HePREM: Enabling predictable GPU execution on heterogeneous SoC. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018.

13 Bjorn Forsberg, Andrea Marongiu, and Luca Benini. GPUguard: Towards supporting a predictable execution model for heterogeneous SoC. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017.

14 TOBIAS GROSSER, ARMIN GROESSLINGER, and CHRISTIAN LENGAUER. Polly — Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters*, 22(04), 2012.

15 Emna Hammami and Yosr Slama. An overview on loop tiling techniques for code generation. In *Proceedings of IEEE/ACS International Conference on Computer Systems and Applications, AICCSA*, volume 2017-October, 2018.

16 Mohamed Hassan and Rodolfo Pellizzoni. Bounding DRAM Interference in COTS Heterogeneous MPSoCs for Mixed Criticality Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11), 2018.

17 Hyoseung Kim, Dionisio De Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *Real-Time Technology and Applications - Proceedings*, 2014.

18 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, CGO*, 2004.

19 Renato Mancuso, Roman Dudko, and Marco Caccamo. Light-PREM: Automated software refactoring for predictable execution on COTS embedded systems. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, 2014.

20 Joel Matějka, Björn Forsberg, Michal Sojka, Zdeněk Hanzálek, Luca Benini, and Andrea Marongiu. Combining PREM compilation and ILP scheduling for high-performance and predictable MPSoC execution. In *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores - PMAM'18*, New York, New York, USA, 2018. ACM Press.

21 Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio Buttazzo. Memory-processor co-scheduling in fixed priority systems. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems - RTNS '15*, New York, New York, USA, 2015. ACM Press.

22 Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for COTS-based embedded systems. In *Real-Time Technology and Applications - Proceedings*, 2011.

23 Benjamin Rouxel, Steven Derrien, and Isabelle Puaut. Tightening Contention Delays While Scheduling Parallel Applications on Multi-core Architectures. *ACM Transactions on Embedded Computing Systems*, 16(5s), 2017.

24 M.R. Soliman and R. Pellizzoni. WCET-driven dynamic data scratchpad management with compiler-directed prefetching. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, 2017.

25 Muhammad R. Soliman and Rodolfo Pellizzoni. Data Scratchpad Prefetching for Real-time Systems. Technical report, UWSpace, 2017.

26 Muhammad R Soliman and Rodolfo Pellizzoni. Optimal Task Segmentation for PREM-based Systems Under Fixed Priority Scheduling. Technical report, University of Waterloo, Canada, 2019. URL: `http://ece.uwaterloo.ca/~rpellizz/techreps/optimal_seg_tech_report.pdf`.

27 Rohan Tabish, Renato Mancuso, Saud Wasly, Ahmed Alhammad, Sujit S. Phatak, Rodolfo Pellizzoni, and Marco Caccamo. A Real-Time Scratchpad-Centric OS for Multi-Core Embedded Systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2016.

28 Shelby Thomas, Chetan Gohkale, Enrico Tanuwidjaja, Tony Chong, David Lau, Saturnino Garcia, and Michael Bedford Taylor. CortexSuite: A synthetic brain benchmark suite. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2014.

29 UTDSP Benchmark Suite. URL: `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html`.

30 Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming Non-Blocking Caches to Improve Isolation in Multicore Real-Time Systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2016.

31 Saud Wasly and Rodolfo Pellizzoni. A Dynamic Scratchpad Memory Unit for Predictable Real-Time Embedded Systems. In *2013 25th Euromicro Conference on Real-Time Systems*. IEEE, 2013.

32 Saud Wasly and Rodolfo Pellizzoni. Hiding memory latency using fixed priority scheduling. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2014.

33 Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Emiliano Betti, and Marco Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems*, 48(6), 2012.

34 Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Heechul Yun, and Marco Caccamo. Global Real-Time Memory-Centric Scheduling for Multicore Systems. *IEEE Transactions on Computers*, 65(9), 2016.

RT-CASEs: Container-Based Virtualization for Temporally Separated Mixed-Criticality Task Sets

Marcello Cinque
Federico II University of Naples, Italy
`wpage.unina.it/macinque`
macinque@unina.it

Raffaele Della Corte
Federico II University of Naples, Italy
raffaele.dellacorte2@unina.it

Antonio Eliso
Federico II University of Naples, Italy
antonio.eliso@studenti.unina.it

Antonio Pecchia
Federico II University of Naples, Italy
antonio.pecchia@unina.it

Abstract

This paper presents the notion of real-time containers, or rt-cases, conceived as the convergence of container-based virtualization technologies, such as Docker, and hard real-time operating systems. The idea is to allow critical containers, characterized by stringent timeliness and reliability requirements, to cohabit with traditional non real-time containers on the same hardware. The approach allows to keep the advantages of real-time virtualization, largely adopted in the industry, while reducing its inherent scalability limitation when to be applied to large-scale mixed-criticality systems or severely constrained hardware environments. The paper provides a reference architecture scheme for implementing the real-time container concept on top of a Linux kernel patched with a hard real-time co-kernel, and it discusses a possible solution, based on execution time monitoring, to achieve temporal separation of fixed-priority hard real-time periodic tasks running within containers with different criticality levels. The solution has been implemented using Docker over a Linux kernel patched with RTAI. Experimental results on real machinery show how the implemented solution is able to achieve temporal separation on a variety of random task sets, despite the presence of faulty tasks within a container that systematically exceed their worst case execution time.

2012 ACM Subject Classification Software and its engineering → Real-time systems software

Keywords and phrases Containers, mixed-criticality, temporal separation, monitoring

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.5

Funding This work is partially supported by the RT-CASE project, funded by the Dept. of Electrical Engineering and Information Technology of the University of Naples Federico II, Italy.

Acknowledgements We are thankful to the anonymous reviewers of the ECRTS program committee for the valuable comments, which allowed us to improve the paper and provided useful guidance to better target our future research efforts on rt-cases.

1 Introduction

A **mixed-criticality system** (MCS) can be defined as a real-time and embedded system integrating software components with different levels of criticality onto a common hardware platform [4]. The trend in the development of MCSs was initially intertwined with the migration from single-core to many-core architectures, which paved the way to the opportunity of having components with different degrees of criticality (e.g., in terms of timeliness and fault tolerance) running on the same hardware. An increasing number of vendors, such as automotive and avionics industries, are considering MCSs to meet stringent non-functional requirements. A typical example is to run hard-real time tasks to control the breaking system of a car on the same board that runs non-critical monitoring tasks for diagnostics, in order to reduce costs, space, and power consumption. Noteworthy, the concept of mixed-criticality is now recognized and supported by the main software standards in automotive (e.g., AUTOSAR – `www.autosar.org`) and avionics (e.g., ARINC – `www.arinc.com`).

A fundamental research challenge for MCSs is to assure the correct execution of high critical tasks, with a disciplined (and possibly user-transparent) use of the underlying shared resources. At least **temporal separation**, and fault isolation of tasks must be guaranteed, in order to avoid that a problem or a delay in a low criticality task can affect a high criticality one. A number of theoretical approaches have been defined, especially for task allocation [15] and schedulability analysis [24] in multi-processor systems. However, when it comes to the actual implementation of the software components, the assurance of separation and isolation properties can easily become a burden for developers. For this reason, several frameworks have been proposed, many of them tailored for a particular domain (e.g., [11] in the automotive) or bound to particular platforms (such as [22] for FPGAs).

Many solutions capitalize on **virtualization technologies** to separate real-time kernels from non real-time ones, by means of different virtual machines (VMs) hosted by specific hypervisors. For instance, Wind River has recently introduced a Virtualization Profile for VxWorks [25], integrating a real-time embedded, type 1 hypervisor into the VxWorks real-time operating system, making it possible to consolidate multiple stand-alone hardware platforms onto a single multi-core platform. Other examples are PikeOS [14], a separation microkernel providing a paravirtualization real-time operating system for partitioned multi-core platforms, and ARINC 653, one of the first industrially used hypervisors, insuring temporal and spatial isolation of different RTOSes. The use of virtualization allows developers to work with their preferred environment, and to deal with separation and isolation issues at the hypervisor level; however, running VMs on a single host has a significant overhead. More importantly, creating VMs for every hardware platform to consolidate may lead to both software and OS stretch. In practice, real-time virtualization solutions – such as the ones above – are designed to deal with the (deterministic) performance penalty introduced by a limited number of VMs. We observe that the overhead caused by replicating entire OS environments makes it unfeasible to scale-up to a large number of applications of different criticalities, especially when they need to be consolidated on a limited number of machines.

A concrete example of such a need is represented by the so-called real-time infrastructure currently under development as part of the control and data acquisition system of the ITER[1] tokamak [26]. The ITER international project aims at the construction of the world's largest nuclear fusion experimental facility in Saint-Paul-lès-Durance, south of France. The construction of ITER is a challenge itself, as well as its future operation, which aims at proving the feasibility of energy production by means of nuclear fusion on Earth. The inherent complexity of ITER (such as any other large-scale critical system) requires to consolidate tens of thousands of applications of different size and criticality – spanning from distributed control to monitoring, sensor reading and network communication – on a limited number of interconnected machines. Different real-time frameworks are currently under development within the ITER project, in order to deal with the complexity of the overall project and to

[1] ITER is a Nuclear Facility INB-174. The views and opinions expressed herein do not necessarily reflect those of the ITER Organization.

facilitate integrated testing and commissioning [26]. Up to now, the ITER organization has not focused yet on a solution to manage the integration and orchestration steps in a simple way, considering the different criticality levels, temporal separation, and fault isolation needs of the components to be run on the available hardware.

1.1 Contributions of the paper

To face these issues, typical of any large-scale mixed-criticality system, in this paper we propose the notion and a possible implementation of **real-time containers** here referred to as: **rt-cases**. Containers are today considered a key technology to achieve high consolidation scalability with minimal space and performance overhead. For instance, Docker relies on an operating system-level virtualization method for running multiple isolated Linux systems (containers) on a single host. Technically a container is a set of processes and libraries isolated from the rest of the machine. Unlike a VM, a container does not need to replicate the whole OS environment for every system, hence reducing the overhead and increasing the number of applications with different criticalities that can be run on a single node. The possible advantages of rt-cases are evident: different developers can still work with their preferred environment, which is desirable in large and heterogeneous teams. The resulting rt-cases can be flexibly deployed on the available hardware, as done today with non real-time containers.

With rt-cases we aim to explore the benefits of combining containers and hard real-time co-kernels. We propose a reference architecture for the rt-cases concept, firstly introduced in [7], [6]. In this paper we explore the design space of rt-cases and discuss possible alternatives for their realization on top of a Linux kernel patched with a real-time co-kernel. We present a specific solution to rt-cases, based on fixed-priority scheduling, runtime execution monitoring, and on-line mitigation. This solution allows to achieve temporal separation without requiring modifications to the underlying real-time kernel. The paper presents the implementation details and technical challenges of such solution, based on Docker and RTAI, and reports the results of an experimentation with randomly generated feasible task sets. Results show that, thanks to our runtime monitoring and mitigation strategy, the tasks running within a container are not affected by *faulty* tasks (i.e., tasks that exceeds their declared worst case execution time) running in a different container, regardless of the criticality of the container. We also present the limitations of our current implementation, and discuss prospected improvements and related trade-offs.

The rest of the paper is organized as follows. Section 2 presents related work in the area and positions our work in context of existing contributions. Section 3 provides the reference architecture and a discussion on potential alternatives to support the implementation of rt-cases. Section 4 discusses the system model, while Section 5 describes a specific implementation, technical challenges and prospected developments. Section 6 illustrates the functioning of the approach with a case study and a measurement campaign. Section 7 concludes the work.

2 Related work

A consolidated trend in the literature and in the industry for time and space partitioning in mixed-criticality systems is to make use of virtualization technologies. These solutions range from type 1 hypervisors to full-featured operating systems, with the aim to completely separate real-time kernels from non real-time ones.

RT-Xen [29] is a real-time hypervisor scheduling framework for Xen [3], a widely used open-source type 1 hypervisor. The RT-Xen project extends Xen to support virtual machines with real-time performance requirements. RT-Xen features a compositional real-time scheduling framework that bridges the gap between virtualization technology and real-time scheduling theory for predictable computing on virtualized platforms. In [28] the second version of RT-Xen is presented, which implements both global and partitioned VM schedulers, with each scheduler being configurable to support dynamic or static priorities and to run VMs as periodic or deferrable servers. RT-OpenStack [27] is a cloud CPU resource management system for co-hosting real-time and regular VMs. RT-OpenStack integrates the real-time hypervisor RT-Xen in the OpenStack cloud management system through a real-time resource interface. PikeOS [14] is a separation microkernel targeted to real-time embedded systems, which aims to provide a partitioned environment for multiple operating systems with different design goals to coexist in a single multi-core platform. It provides full separation in both time and space for multiple software applications running on different criticality levels. XTRATUM [20] is a type 1 hypervisor specially designed for real-time embedded system, which employs para-virtualization techniques. XTRATUM can be used to build partitioned systems and provides both strong temporal separation, through a fixed cyclic scheduler, and strong spatial isolation, since all partitions are executed in processor user mode and do not share memory. The Wind River VxWorks RTOS [25] features a Virtualization Profile that integrates a real-time embedded, type 1 hypervisor into the core of VxWorks, which is able to slow down general purpose operating systems to ensure that real-time operating systems can execute without performance impact.

An alternative to virtualized solutions, which require running a number of virtual machines, is represented by the use of container-based environments. Differently from virtual machines, where the operating system stack is entirely replicated for each virtual machine, containers allow running multiple isolated Linux systems on a single host with minimal space and performance overhead. We believe that containers can be a useful alternative or complement to virtualized partitioned systems with a lightweight solution to sandbox multiple real-time applications into isolated software environments on the same hardware or VM.

Recently, few studies have started to explore the possibility to use containers to run real-time tasks. For example, [18] presents an empirical study on the problem of minimizing computational and networking latencies for Radio Access Networks (RAN) through lightweight containers. The study analyzes the performance of Docker containers running on the top of a Linux kernel patched with Ingo Molnar's preemption patch (PREEMPT-RT). The obtained results highlight that the use of PREEMPT-RT improves latencies on Docker containers when compared to a generic kernel. The paper in [19] proposes a sand-boxed environment, based on Docker containers on a Linux kernel patched with PREEMPT-RT, to deploy the software in automotive industry. Experimental results highlight that the use of containers does not affect the performance of the software when compared with the native environment.

It should be noted that both the solutions make use of PREEMPT-RT in order to meet the real-time requirements of the considered applications. Despite the good results obtained, it is recognized that real-time co-kernels, such as RTAI (www.rtai.org) or Xenomai (www.xenomai.org), outperform PREEMT-RT in terms of latencies and task switch times [8][12], since they make Linux fully preemptable in favor of real-time tasks. Co-kernels also add core real-time support to user level real-time tasks, such as fixed-priority or dynamic scheduling, resource management with priority inheritance, and inter-task communication. An alternative would be to guarantee a fixed CPU bandwidth to containers using server-based scheduling, such as the Sporadic Server (SS) [23], periodic or deferrable servers as done in RT-XEN [28], or the Constant Bandwidth Server (CBS) [1]. For instance, the SCHED_DEADLINE scheduling policy [16], available in the Linux kernel since version 3.14, is an implementation of the Earliest Deadline First (EDF) scheduling algorithm, augmented with a CBS, that makes it possible to isolate the behavior of task groups. However, the use of containers on top of co-kernels with server-based schedulers has not been explored yet.

Starting from these considerations, and from the opportunity, already emerged in the literature, to use containers in real-time environments, in this paper we aim to propose a reference architecture scheme where hard real-time tasks within containers are scheduled by a co-kernel, such as RTAI or Xenomai. Our aim is to fully inherit the advantages of the co-kernel in terms of real-time performance and functionalities, while letting tasks within containers to keep using the same application programming interface, as if they were running on the native system patched with the co-kernel.

3 High-level architecture

Figure 1 depicts the reference architecture underlying the proposed rt-case approach. The idea is to host real-time (rt-) tasks within containers marked with different Criticality Levels (CL, e.g. CL:0 and CL:1 in figure) on top of a patched real-time Linux kernel. The CL is used to establish the relative importance of rt-cases.

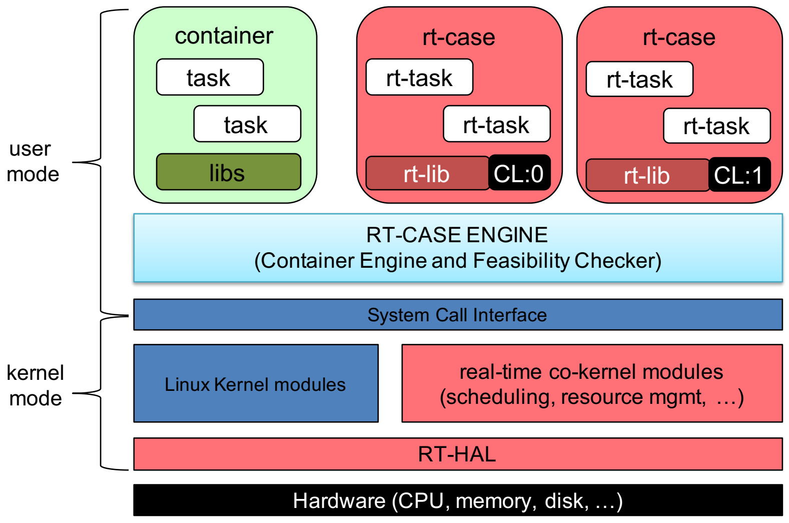


Figure 1 High level scheme of the rt-case architecture.

The RT-CASE engine includes a container engine, e.g., Docker (www.docker.com) or Linux Containers (www.linuxcontainers.org), and a *feasibility checker*, to verify if a new rt-case can be admitted on a running computing node, without affecting the rt-cases already hosted on it. At kernel level, we imagine to have a vanilla Linux kernel patched with a real-time co-kernel, such as RTAI or Xenomai, in order to make the Linux kernel and all the non real-time environment (including traditional containers) fully preemtable by rt-tasks run within rt-cases.

The rt-lib is a key component of the architecture: its objective is to provide a transparent mapping of rt-tasks on the underlying real-time core, depending on the CL of the container, possibly exposing standard primitives to rt-tasks. With this approach, the code of rt-tasks does not need to be modified to run in a container. Hence, the same rt-case can be moved over time on the different machines of a large-scale computing environment, and with a different CL, depending on temporal needs, hardware constraints, and presence of other rt-cases, as regularly done in container-based environments.

3.1 Design alternatives

A simple way to implement the rt-cases architecture is to map whole containers on real-time tasks (see Figure 2a). In this case, a cooperative, non-preemptive user-level scheduler can be implemented in the rt-lib to schedule real-time tasks running within an rt-case. Even if conceptually simple, this solution does not allow to define precise individual deadlines for tasks within containers, which are hence forced to share a container deadline. This also requires to introduce specific primitives for cooperative scheduling in the rt-lib (such as yield primitives) that contradicts our idea to adopt standard primitives for real-time tasks.

Preemptive scheduling for individual tasks within containers overcomes these limitations, but, to achieve temporal separation, tasks belonging to different containers have not to interfere each other. To this aim, a possible solution is to adopt hierarchical scheduling [9][21], e.g., by using Earliest Deadline First (EDF) on task groups (one group for rt-case), each with a fixed bandwidth U_i guaranteed by a server-based mechanism, such as, the Constant Bandwidth Server (CBS) [1] (see Figure 2b), or by using the Hierarchical CBS scheme [17]. This is similar to what done in RT-Xen to assign virtual machines on different virtual CPUs in a multicore environment, with different bandwidths [28]. In this case, to admit a $(n+1)th$ rt-case to an existing computing node with n rt-cases already running, it is sufficient to check that: $\sum_{i=1}^{n} U_i \ + \ U_{n+1} \leq 1$. Even if simple in principle, this solution implies a significant implementation effort in our architectural proposal, since it requires to modify the real-time co-kernel (e.g., RTAI or Xenomai core modules) to implement the hierarchical scheduling solution.

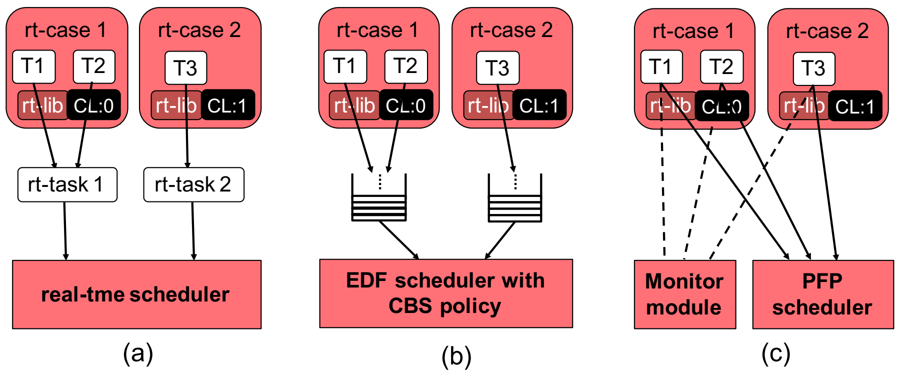


Figure 2 Design alternatives to implement the rt-case model: (a) mapping of whole containers on single rt-tasks; (b) use of task groups and hierarchical scheduling (for instance, EDF with CBS); (c) use of preemptive fixed-priority scheduling, static priority assignment, and temporal protection with on-line execution time monitoring.

A possible compromise to achieve temporal separation without requiring modification to the real-time kernel support is to perform a one-to-one mapping of tasks within containers with real-time tasks at kernel level, using a preemptive fixed-priority (PFP) scheduler and a static priority mapping (see Figure 2c). Using a proper priority assignment, we can assure that rt-tasks running within a container with a high CL (corresponding to a low CL value, e.g., 0 in the figure) cannot be preempted neither by any rt-task running in a container with a lower CL (e.g., 1 in the figure) nor by any non-real time task (running either in other containers or on the host OS). A priority assignment algorithm is needed in this case, in order to assure both the feasibility of individual tasks and the feasibility of whole rt-cases to be admitted on a computing node. The advantage of this solution is simplicity: in principle, it only requires a PFP scheduler, such as rate monotonic (RM), largely available in existing real-time operating systems. In practice, the solution requires a temporal protection mechanism (with on-line task execution time monitoring), in order to assure that faulty tasks, exceeding their worst case execution time (WCET), do not interfere with tasks belonging to different containers.

In this paper we focus on the implementation of the rt-cases model following the last alternative, i.e., fixed priority scheduling with monitoring, as in Figure 2c. Future work will be devoted to the implementation of the hierarchical scheduling alternative (Figure 2b) and to the comparison between the two solutions.

4 System model

We assume a system composed of M rt-cases, each of them with a criticality level $CL_j \;:\; j = 0...M-1$. A criticality level CL_j can assume an integer value in the interval $[0, CL^{max}]$, with 0 being the highest CL, and CL^{max} the lowest CL. Each rt-case hosts one or more periodic hard real-time tasks τ_i, characterized by a worst case execution time C_i and a period T_i. We assume T_i to be coincident with the relative deadline of the task. Overall, the system is composed by a set Γ of N tasks, each of them assigned to an rt-case with a given CL. With $\Gamma(CL)$ we indicate the subset of tasks with criticality level CL. By construction: $\Gamma = \Gamma(CL_0) \cup ... \cup \Gamma(CL_{M-1})$ and $\Gamma(CL_k) \cap \Gamma(CL_h) = \emptyset$, where $CL_k \neq CL_j$.

For example, considering the system depicted in Figure 3, with 4 rt-cases, 8 tasks, and 3 criticality levels, we have: $M = 4$; $N = 8$ $CL^{max} = 2$; $\Gamma = \{\tau_1 \;...\; \tau_8\}$, $\Gamma(0) = \{\tau_1, \tau_2\}$, $\Gamma(1) = \{\tau_3, \tau_4, \tau_5, \tau_6\}$, and $\Gamma(2) = \{\tau_7, \tau_8\}$.

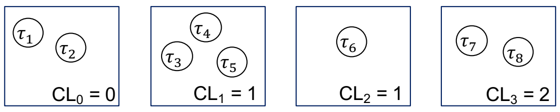


Figure 3 An example of decomposition of tasks within containers: circles represent tasks and squares represent rt-cases with different CLs; rt-cases 1 and 2 have the same CL.

Each task τ_i is characterized by a static priority p_i. Priority values range in the integer interval $[0, N-1]$, where N is the total number of tasks, assuming 0 to be the highest and $N-1$ the lowest priority values. Priorities have to be assigned to tasks according to the CL of the container they belong to. In particular, to avoid that tasks in a high criticality rt-case are preempted by tasks in a lower criticality rt-case, we assume that:

$$CL_k < CL_H \;\wedge\; \tau_i \in \Gamma(CL_k) \;\wedge\; \tau_j \in \Gamma(CL_h) \iff p_i < p_j \tag{1}$$

In other terms, tasks belonging to high criticality containers must receive high priorities, and viceversa.

Once priorities are assigned, tasks can be scheduled by a PFP scheduler. Having assumed a PFP scheduler, a simple method to assign priorities to tasks, while checking the feasibility of the system, is to adopt the Audsley's priority assignment algorithm [2], with the classical Joseph-Pandya's response time analysis (RTA) schedulability test [13]. RTA consists in computing the response time R_i of a task τ_i as: $R_i = C_i + \sum_{p_k<p_i} \lceil R_i/T_k \rceil \cdot C_k$. R_i takes into account the interference that a task can suffer due to preemptions by higher priority tasks. The task set is schedulable if and only if $R_i \leq T_i \; \forall i$.

The Audsley's algorithm is based on the following lemmas:

- *Lemma 1*: the worst case response time of a task τ_i can be determined with a test (such as RTA), by knowing the tasks that have priorities greater than τ_i, but without knowing the specific assignments of such priorities;
- *Lemma 2*: if a task is schedulable considering a given priority assignment, then it remains schedulable if it receives a higher priority.

We adapted the algorithm to our case, as shown in Alg. 1. In particular, to be compliant with our assumption in (1), we force the assignment of priorities to proceed from tasks belonging to the lowest criticality rt-case (with $CL = CL^{max}$) to the highest one (with $CL = 0$, see line 2 in the algorithm), starting from the lowest priority ($N - 1$) to the highest one (0). A priority is assigned to a task τ_i if $R_i < T_i$ (see lines 5-6). If we are not able to assign the priorities to all the tasks of a given CL, then the task set is unschedulable (see lines 10-11). If all priorities are assigned, the task set is schedulable.

Algorithm 1 Priority Assignment Algorithm (Audsley's algorithm adaptation).

```
1:  $p \leftarrow N - 1$
2:  for $CL = CL^{max}$ down to 0 do
3:      while $p \geq 0$ do
4:          for each unassigned task $\tau_i$ in $\Gamma(CL)$ do
5:              if $R_i \leq T_i$, with all unassigned tasks assumed to have priorities $< p$ then
6:                  assign $p$ to $\tau_i$
7:                  $p = p - 1$
8:              end if
9:          end for
10:         if not all tasks in $\Gamma(CL)$ can be assigned then
11:             return UNSCHEDULABLE;
12:         end if
13:         break
14:     end while
15: end for
16: return SCHEDULABLE;
```

Such priority assignment solution assures isolation of high criticality rt-cases from lower ones, since, by construction, a task running in an rt-case with CL_k can never be preempted by a task running in an rt-case with $CL_h > CL_k$, according to (1). However, we must also ensure that faulty tasks in high criticality rt-cases (e.g., a task instance, or job, exceeding its C_i) do not affect tasks in low criticality rt-cases. Hence, we assume the system to be equipped with a temporal protection mechanism implemented by a *monitor*, running on a different CPU than the one(s) running rt-cases. The monitor has to measure the execution time of tasks at runtime, in order to interrupt a job of a periodic task whenever it exceeds its declared C_i.

With reference to the rt-case architecture in Figure 1, the proposed priority assignment algorithm has to be implemented by the feasibility checker within the RT-CASE engine. The algorithm can be run whenever a new rt-case becomes ready, to check if it can be admitted to a CPU hosting other rt-cases without affecting their execution.

It has to be noted that the assignment obtained with Alg. 1 may not reflect the priority assignment originally planned by the application developer for the tasks to be run within his/her container. This might still be fine in the cases where the developer does not care about individual task priorities, as long as the assignment guarantees that task deadlines are met. However, if application dependent constraints on priorities must be met, Alg. 1 cannot be used as it is. In this case, our implementation leaves to the developer the chance to manually assign the priorities to tasks, at his/her own risk. A viable alternative is to force the algorithm (i) to allot disjoint priority intervals to different containers (even for rt-cases with the same CL), and (ii) to assign priorities to tasks within a rt-case by respecting their original relative order, in terms of the priorities assigned by the developer. A different solution is to adopt the rt-case design alternative based on hierarchical scheduling and server-based approaches (see Figure 2b), since it makes priority assignments within rt-cases independent from the feasibility checker. We leave the implementation and evaluation of this solution, along with the comparison with the currently developed scheme, as future work.

5 Implementation details

We present the details on the implementation of our proposal on top of RTAI (Real Time Application Interface), a real-time co-kernel extension for Linux. RTAI is an open-source project born to add real-time capabilities to standard Linux kernels. It is conceived as a patch of the Linux Hardware Abstraction Layer (the RT-HAL) that makes the Linux kernel fully preemptable in favor of real-time tasks, by masking the interrupts handling mechanism and by redirecting interrupts to the Linux kernel only when there is no real-time activity to be performed. The RT-HAL is complemented by a number of RTAI modules, providing a rich set of services for real-time tasks running at the user level, among which real-time task management, real-time inter-process communication with priority inheritance, etc.

We implement our proposal on a vanilla Linux kernel 4.9.80 patched with RTAI 5.1. The proposal encompasses three main components, as depicted in Figure 4: the *RT-CASE engine*, the *RT-lib* and the *RT-CASE monitoring*. The implementation of these components is presented in the following[2].

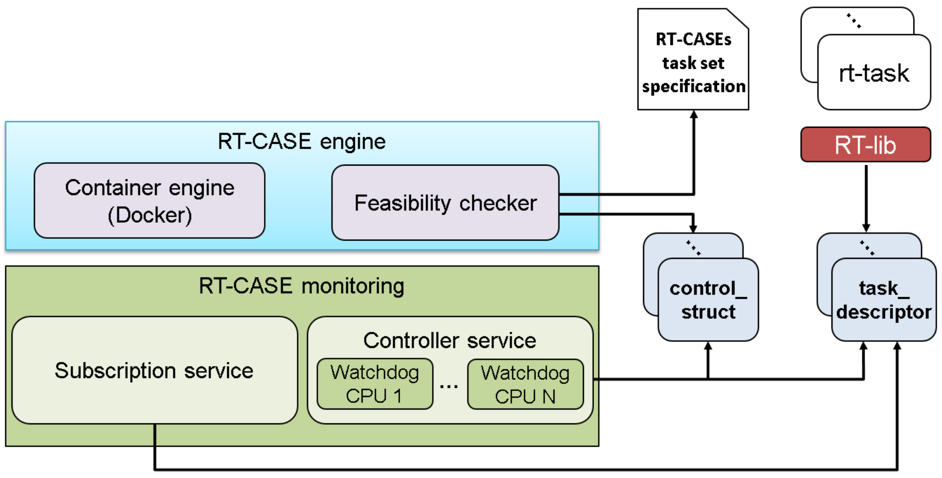


Figure 4 System components.

[2] The source code of the components has been made publicly available at `http://www.dessert.unina.it/RT-CASE.zip`.

5.1 RT-CASE engine

The *RT-CASE engine* represents a macro-component dedicated to the orchestration of the rt-cases and, more important, to the feasibility evaluation of the rt-task set to be run inside the rt-cases. The RT-CASE engine is composed by two components: the *feasibility checker* and the *container engine*.

The **feasibility checker** aims to verify the schedulability of a given task set Γ. It is a Python script that accepts the *specification* of a task set as input. For each task τ_i, the specification contains: (i) id, (ii) name, (iii) the period T_i, (iv) the worst case execution time C_i, (v) the CL and the name of the container hosting the task. Given the task set, the feasibility checker runs the priority assignment algorithm (Alg. 1); if the task set is schedulable, the component returns the priorities to be assigned to tasks. In particular, it fills a data structure, named *control_struct*, containing for each task: its id, its name, the rt-case where it runs, its priority and its C_i. The structure is then used at run-time by the *RT-lib*, as a *contract* for the tasks, as described in Section 5.2.

The **container engine** is used to launch feasible rt-cases. In our implementation, we use Docker as container engine. Each rt-case is a Docker container, including in its image the *RT-lib* and the executables of the rt-tasks to run. Each rt-case is run providing the *PIDC* env variable, which contains the ID of the container; the variable is used by the *RT-lib* as described in Section 5.2. It is important to note that only user-space rt-tasks are allowed to run inside the rt-cases. Kernel-space rt-tasks can be run at host level; however, they are not managed by our proposal. More important, a number of capabilities as well as the access to some host devices have to be granted to rt-cases in order to allow the execution of rt-tasks. For example, if the tasks of a given rt-case needs to access to both the `/dev/rtai_shm` and `/dev/rtc0` device files (representing the RTAI shared memory and real-time clock device files, respectively), the rt-case has to be lunched with the following Docker flags: `-device=/dev/rtai_shm:/dev/rtai_shm -device=/dev/rtc0:/dev/rtc0`. Similarly, since rt-tasks usually need to set the real-time clock, the rt-cases are lunched with the Docker flag `-cap-add=SYS_TIME`.

5.2 RT-lib

The RT-lib exports the user-space APIs provided by RTAI, which are made available for each rt-task running inside an rt-case. More in details, the RT-lib encapsulates a modified version of the RTAI LXRT library, i.e., the library allowing the access to all the services made available by RTAI and its schedulers in user-space; a number of primitives are modified in order to grant *naming isolation* and to assign priorities to rt-tasks running inside rt-cases.

Naming isolation allows avoiding clashes due to the use of the same name for a rt-task or a resource from applications running in different rt-cases. We modify the LXRT *rt_task_init_schmod* primitive, which allows to create and initialize rt-tasks in user-space, to combine the name of the rt-task, generally provided as input parameter, with the ID of the related rt-case, which is defined through the *PIDC* env variable of the rt-case. Similar modifications have to be applied to the other LXRT initialization primitives, such as *rt_typed_sem_init* and *rt_typed_mbx_init*, which allow the initialization of semaphores and mailboxes, respectively.

To perform the priority assignment, we modify the *rt_task_init_schmod* primitive. The modified version of this primitive sets the task priority to the one defined in the *control_struct* (the contract) generated by the *feasibility checker*. If the priority value is not defined in the *control_struct*, i.e., the feasibility check has not been executed for the current task set or the task set is not feasible, the primitive does not change the priority.

Finally, in order to assure temporal separation through the protection mechanism implemented by the monitoring component (see next section), both the *rt_task _init_schmod* and *rt_task_wait_period* are modified. The *rt_task_init_schmod* is modified to create and initialize a data structure, named *task_ descriptor*, used by the monitoring component and containing the information about the current task. The descriptor, created if the monitoring is active, logically extends the Linux *task_struct* (without actually modifying it) with a number of fields required by the monitoring component. The main fields of the the *task_descriptor* are detailed in Table 1. The *rt_task_wait_period*, i.e., the RTAI primitive that suspends the execution of a periodic hard real-time task until the next period, is modified to count the number of cycles of the task. This parameter is reported in the *cycle* field of the *task_descriptor*. Before invoking the syscall allowing the task to wait the next period, the primitive first verifies if the monitoring is active and, in this case, increments the *cycle* field. When the task is resumed at the next cycle, the primitive also saves the current time in the *last_switch_time* field of the *task_descriptor*, which is used by the monitoring component to measure the execution time of the task (see next section).

It should be noted that all the modifications explained above are totally transparent to the developer. In fact, the modifications affect only the body of the primitives, while their signatures are left unchanged. More important, the modifications only affect the RTAI LXRT user-space library, without any change to the RTAI kernel level source code. This avoids the need to re-compile the kernel; only the library source code needs to be compiled and installed in the containers, fostering an easy and quick adoption of the solution. Finally, it is important to note that if the monitoring component or the feasibility checker is not activated, all the modified primitives act as the original ones. This leaves the programmer the freedom to manually assign the priorities at his/her own risk.

Table 1 Main fileds of *task_descriptor*.

Field	**Description**
wcet	Worst Case Execution Time (WCET, or C_i) of the task
exec_time	Execution time of the task in the current cycle
last_switch_time	Time of the last context switch involving the task
last_overrun	Last time the task has been found in overrun
task_alarm	Pointer to the task in charge to manage the task, when it becomes faulty
cycle	Current cycle of the task
last_cycle	Last cycle of the task
overtime	Number of overtimes of the task
overrun	Number of overruns of the task

5.3 RT-CASE monitoring

The *RT-CASE monitoring* component aims to provide temporal protection to rt-cases. The component prevents that a faulty task (i.e., a task exceeding its WCET or having an activation frequency higher than the one declared during the feasibility checking) running within an rt-case may affect the tasks running in rt-cases with lower or equal CLs. To this aim, the component periodically checks the execution time and the activation frequency of the rt-tasks running inside the rt-cases; in addition, it measures the number of *overruns* and *overtimes* of each task, i.e., the number of times a task exceeds its deadline and WCET, respectively. When a faulty task is detected, the RT-CASE monitoring implements one of the following *policies* to guarantee temporal protection:

- `KILL`: the task is killed in a forced way;
- `SUSPEND`: the task is suspended indefinitely;
- `FORCE_PERIOD`: the task is suspended and it will be resumed at the next period activation;
- `SIGNAL`: a notification is sent to a given task, which will be in charge to take action as a consequence of the fault.

The policy to adopt represents one of the parameters of the monitoring component, which also includes the CPU it has to be run on and the monitoring period.

The RT-CASE monitoring component has been developed as a kernel module. The module launches a number of tasks on a dedicated processor in order to avoid that the monitoring operation affects the rt-tasks running inside rt-cases. Therefore, the proposed monitoring approach requires a multiprocessor hardware configuration with at least 2 CPUs, one for monitoring and the other one for the rt-tasks running in rt-cases. The usage of a single processor system is also possible (although we leave it as future work); however, there are two constraints to be considered in this case: (i) the monitoring task has to be taken into account in the feasibility checking and (ii) it has to be the task with the highest priority.

The RT-CASE monitoring component provides two main services: *subscription service* and *controller service.*

The **subscription service** is an aperiodic task that is activated when the `SIGNAL` policy is used. The service allows developers to subscribe a *control task* for a rt-task to monitor. The control task will be notified each time the monitored rt-task is detected as faulty, as defined by the `SIGNAL` policy. When a control task requests the subscription for a given rt-task, the subscription service retrieves the *task_descriptor* of the rt-task and stores the pointer to the control task into the *task_alarm* field. This policy is useful to handle the recovery of tasks in overtime at the user-space, within rt-cases; however developers must be aware of it. The other policies do not require modifications to the source code of tasks.

The **controller service** is the main service of the monitoring component since it aims to detect faulty tasks. The service is conceived as a set of periodic real-time tasks - *watchdogs* hereinafter. A watchdog is created for each CPU running the rt-cases. Each watchdog periodically executes the following steps: (i) gets the task currently running on the CPU it monitors; (ii) if the current task is a periodic, user-space, hard real-time task, it retrieves the pointer to its Linux *task_struct*; (iii) it retrieves the rt-case *task_descriptor* of the task and measures its execution time and activation frequency; (iv) if a deviation is detected, i.e., the execution time exceeds the WCET of the task and/or the activation frequency is higher than the one declared during the feasibility checking, the configured policy will take place for the task (i.e., `SIGNAL`, `KILL`, `SUSPEND` or `FORCE_PERIOD`).

It should be noted that in order to measure the execution time and the number of overruns and overtimes, the watchdogs leverage information contained in the *task_descriptor*, e.g., *cycle*, *last_cycle* and *last_switch_time* fields, as well as RTAI primitives (e.g., *rt_get_time_cpuid* to obtain the current time in internal count units on a given cpu). In addition, the *switch_time[cpu]* provided by RTAI is used, which indicates the time of the last context switch on the indicated *cpu.* Alg. 2 describes the algorithm used by the watchdogs to measure the execution time of rt-tasks.

The algorithm evaluates the execution time in three different cases, depicted in Figure 5, which shows two rt-tasks running in rt-cases with different CLs: (a) the current task is in a cycle different from the one of the last monitoring check (at M_0 and M_4 for Task 2, at M_2 for Task 1); (b) the current task is in the same cycle of the last monitoring check and it has not been preempted (at M_1, M_5 and M_6 for Task 2); (c) the current task is in the same cycle of the last monitoring check and it has been preempted in favor of a task with a higher

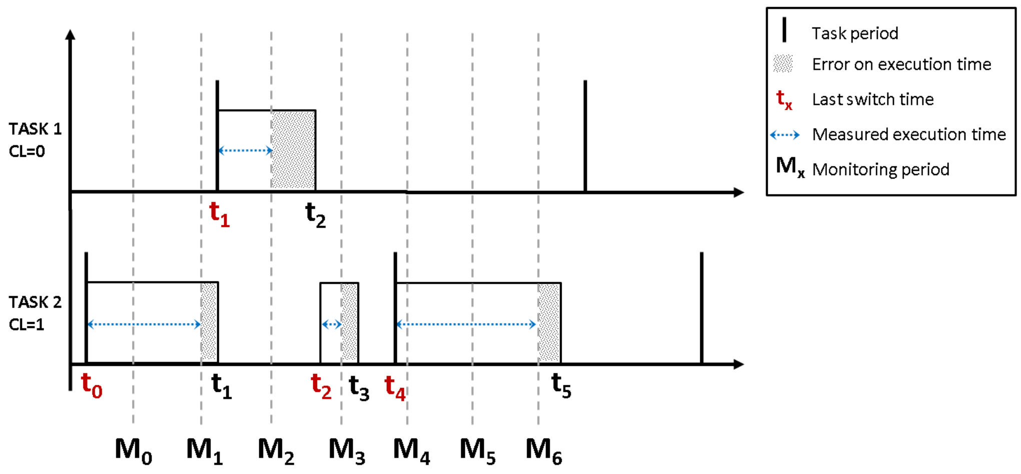


Figure 5 Execution time measurement.

Algorithm 2 Execution time evaluation algorithm.

```
 1: if tsk_desc–>cycle == tsk_desc–>last_cycle then
 2:     if tsk_desc–>last_switch_time == switch_time[cpu] then
 3:         tsk_desc–>exec_time = rt_get_time_cpuid (cpu) - tsk_desc–>last_switch_time;
 4:     else
 5:         tsk_desc–>last_switch_time = switch_time[cpu];
 6:         tsk_desc–>exec_time += rt_get_time_cpuid (cpu) - tsk_desc–>last_switch_time;
 7:     end if
 8: else
 9:     tsk_desc–>last_cycle = tsk_desc–>cycle;
10:     switch_time[cpu] = tsk_desc–>last_switch_time;
11:     tsk_desc–>exec_time = rt_get_time_cpuid (cpu) - tsk_desc–>last_switch_time;
12: end if
```

priority (at M_3 for Task 2). In both cases (a) and (b) the current execution time of the task is evaluated as the difference between the current time, provided by *rt_get_time_cpuid*, and the *last_switch_time* (Alg. 2 - lines 3 and 11), i.e., M_1 - t_0, M_6 - t_4 (Task 1) and M_2 - t_1 (Task 2) in Figure 5; however, in case (a) the algorithm also updates the *last_cycle* value and updates the *switch_time[cpu]* with the *last_switch_time* (lines 9 and 10) since a new cycle is started. It should be noted that the update of the *switch_time[cpu]* is required to avoid an erroneous evaluation of the execution time when a task enters in a new cycle without any context switch; in fact, in this case the *switch_time[cpu]* is not updated by RTAI. In case (c) the algorithm measures the execution time of the task as the sum between the last measured execution time for the task and the difference between the current time and the *last_switch_time* (Alg. 2 - lines 6), i.e., (M_1 - t_0) + (M_3 - t_2) in Figure 5 for Task 2; noteworthy, the *last_switch_time* here is updated to the *switch_time[cpu]* value (Alg. 2 - line 5), i.e., t_2, in order to consider the begin of the new activation of the task.

Alg. 3 describes the algorithm used by watchdogs to detect tasks in overrun and overtime, and rt-tasks activating before the expected time. An overrun is detected by evaluating the difference between the current time, the *periodic_resume_time*, i.e., the time the task has been or will be resumed after a cycle, and the period of the task (Alg. 3 - line 3). Both the *periodic_resume_time* and the task period are provided through the *task struct* of the task (*task* in Alg. 3); in addition, the function *count2nano* used in the algorithm converts a value

Algorithm 3 Detection algorithm.

```
 1: //CHECK OVERRUN
 2: overrun = count2nano(rt_get_time_cpuid (cpu) - task−>periodic_resume_time - task−>period);
 3: if overrun > 0  AND tsk_desc−>last_overrun != tsk_desc−>cycle AND tsk_desc−>cycle > 1  then
 4:     tsk_desc−>overrun++;
 5:     tsk_desc−>last_overrun = tsk_desc−>cycle;
 6: end if
 7: //CHECK OVERTIME
 8: if count2nano(tsk_desc−>exec_time) > tsk_desc−>wcet AND tsk_desc−>cycle > 0 then
 9:     tsk_desc−>overtime ++;
10:     start_policy(tsk_desc);
11: end if
12: //CHECK ACTIVATION
13: if rt_get_time_cpuid (cpu) < task−>periodic_resume_time) then
14:     start_policy(tsk_desc);
15: end if
```

in tick count to nanoseconds. If the obtained value is positive, the rt-task is considered in overrun, since its execution time exceeds its deadline (that we assume to be equal to the period of the task). For example, in Figure 6 it is depicted a task exceeding its deadline. It can be noted that the sum of the *periodic_resume_time* and the task period is lower than the current time; therefore, the difference evaluated by the detection algorithm is a positive value, which allows detecting the overrun at M_0. When a task is found in overrun, the algorithm increments the number of overruns (line 9), i.e., the *overrun* field of the *task_descriptor*, and saves the cycle of the overrun (line 10), i.e., the *last_overrun* field.

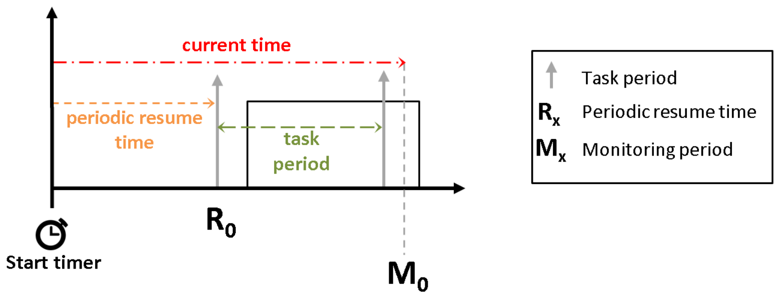


Figure 6 Overrun detection.

Overtimes are evaluated by comparing the evaluated execution time with the WCET of the task (Alg. 3 - line 8); if the execution time is greater than the WCET, the task is considered in overtime and the overtime field of the *task_descriptor* is updated (line 9). Finally, in order to detect task activating before the expected time, the algorithm verifies if the current time is lower than the *periodic_resume_time* (Alg. 3 - line 13); in this case, the task has been activated before expected. If an overtime or a premature task activation occurs, the algorithm calls the *start_policy* function, which applies the policy configured for the temporal protection (lines 10 and 14).

5.4 Discussion

The algorithm currently implemented for RT-CASE monitoring has the benefit of a constant computational complexity, which does not depend on the number of tasks running on the monitored CPUs. At each period, the monitor performs checks on the currently running rt-task, with a relatively small number of lines of code. We measured the typical execution time of an instance of the implemented monitor, e.g., the time it takes to perform a check within a period. The measurements have been conducted in our deployment (composed by a 2.4 GHz dual-core Intel Core i7-5500U machine, equipped with 8 GB DDR3 RAM and a Crucial MX500 SSD) and they show that a watchdog instance is able to run within about 3,500ns in the worst case, regardless of the number of tasks to monitor. This relatively short monitoring time allows very fine grain monitoring periods, in the order of few microseconds, if the monitor is run on a dedicated CPU.

As a drawback, it is important to note that the current implementation of the monitor provides an under-estimation of the execution time of rt-tasks. In fact, each time a task is preempted in favor of a task with a higher priority or terminates its execution in the current activation, the watchdog is not able to estimate the time between the last monitoring activation and the termination of the task execution. For example, in the case depicted in Figure 5 a watchdog is not able to measure the execution time in $[M_2, t_2]$ for Task 1, and in $[M_1, t_1]$, $[M_3, t_3]$ and $[M_6, t_5]$ for Task 2. However, the extent of the estimation error can be easily evaluated. Given a task with a priority p_i and period T_i, the error of the execution time measured by a watchdog in the worst case is given by $T_{mon} \cdot (P + 1)$. T_{mon} is the monitoring period, while P is the number of times the task can be preempted in favor of tasks with a higher priority. Thanks to the adoption of a PFP scheduling policy, the value of P can be easily obtained as: $\sum_{p_k<p_i} \lceil T_k/T_i \rceil$. It should be noted that T_{mon} is multiplied by $P+1$, since we have always an underestimation of the execution time at the task termination in the current period, also if the task is not preempted, as in $[M_6, t_5]$ of Figure 5. Such estimation error can be taken into account in the feasibility analysis (see Section 6.3).

We defined a possible solution for reducing the underestimation obtained with the current implementation. The solution requires that the watchdog stores the current task in a *last monitored task* variable, before terminating the current monitoring activation. This allows the watchdog to update the execution time of the previous monitored task during the next activation; the new value is obtained as the difference between the current *switch_time[cpu]* and *last_switch_time* of the previous monitored task. For example, in Figure 5 the execution time of the Task 2 will be updated at M_2 as t_1 - t_0. In this case, the underestimation error committed by the monitor is T_{mon} at most, when the task ends its execution between two monitoring executions. However, this solution requires the watchdog to analyze two tasks at each cycle. In particular, it has to evaluate the execution time of both the current and the previous task, and it has to execute the overtime and overrun detection for both of them. This may have an impact on the execution time of watchdog instances, consequently requiring to reduce the maximum frequency of monitoring. We leave the implementation of this new version of the monitoring approach, and its evaluation, as future work.

6 Case study

We present a case study of the proposed approach, which consists in a typical setup of *high*, *medium*, and *low* criticality containers, with CL values equal to 0, 1, and 2, respectively. Figure 7 depicts the case study: we hypothesize that the tasks hosted by the *medium* critical rt-case exhibit a **faulty behavior** by exceeding their WCET. We propose a mixture of experiments to gain insights into the following points: (i) thanks to our monitor, the faulty tasks in an rt-case do not impact tasks hosted by the remaining containers, (ii) trade-offs of the monitor and viable workarounds. We deploy the containers on a 2.4 GHz dual-core Intel Core i7-5500U machine, equipped with 8 GB DDR3 RAM and a Crucial MX500 SSD. The installation consists of the following key components: Docker 17.031-ce, Ubuntu 16.04, Linux kernel 4.9.80 and RTAI 5.1. We devote special care to achieve a representative setup although

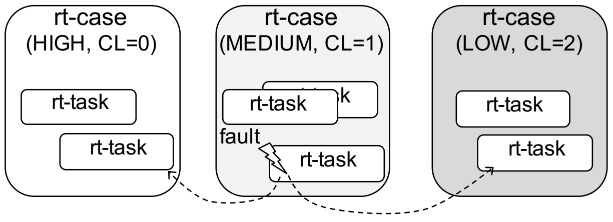


Figure 7 Representation of the case study.

by means of general purpose hardware. As such, the Linux kernel has been configured in order to prevent common sources of non-determinism, such as energy savings, frequency scaling and hyper trading. Monitor and containers are pinned to distinct cores; the *monitoring period* is set to 8,000ns. We would like to point out that this case study is not meant to be exhaustive, but it has been arranged with the aim of eliciting reasonable operating conditions, which illustrate the basic functioning of the approach and its concrete implementation.

6.1 Experiments

We conduct a campaign of experiments[3] to gain insights into the temporal separation across the containers. Each experiment of the campaign consists in generating a **task set** with a total **utilization** U. We run the task set for one minute, and record the following outcomes for each task at the end of the run:

- **overtime**: number of times the task exceeds the WCET;
- **overrun**: number of times the task misses the deadline.

We assess 5 levels of utilization within U=$\{0.45, 0.55, 0.65, 0.75, 0.85\}$; since the task set is generated randomly – as explained in the following – experiments are replicated 30 times for each level U, thus leading to 150 experiments. Moreover, experiments are done both *without* and *with* our monitor, i.e., total 2×5×30 experiments.

For each experiment, the task set with utilization U is synthesized according to the approach in [10]. As such, if we denote by $U_i = \frac{C_i}{T_i}$ the utilization of a task (with C_i and T_i denoting the WCET and the period, respectively), we obtain $U = \sum_{i=1}^{n} U_i$ where n is the number of tasks. In our case study, n is set to 14 because we note that with a higher number of tasks and at levels of utilization higher than 0.85 (i.e., the maximum level of U assessed) it is becomes hard to find feasible task sets with fixed priority scheduling. Once the task set is generated, we (i) assign the tasks to the containers, i.e., 3, 7, and 4 tasks, to the *high*, *medium*, and *low* criticality container, respectively, and (ii) assign the priorities with the feasibility checker based on Alg. 1 in Section 4. The feasibility checker assigns the priorities only if the task set is schedulable, so to be sure that potential overruns are not merely caused by unfeasible task sets.

The tasks execute a *CPU-bound* workload made of arithmetic operations for a time consistent with the declared WCET; however, in order to emulate *faulty* behaviors, the tasks allotted to the *medium* criticality container deliberately keep the CPU busy up to eight times the WCET. For the experiments with monitoring *on*, we use the `SIGNAL` policy: accordingly, each task is accompanied by the corresponding *control task*, which terminates the task under-monitoring when it receives a notification from the monitoring system.

[3] The code used for experiments has been made public available within the source code of the proposal.

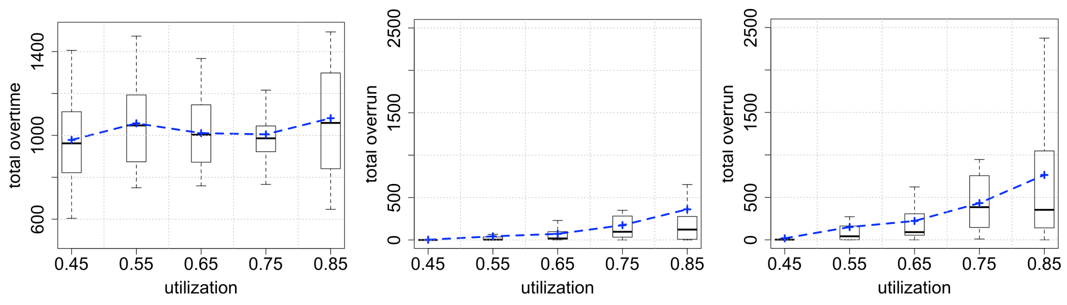
Table 2 shows the outcome of one experiment replication with no monitor and a task set with U=0.85. For each task, we record **overtime** and **overrun** at the end of the experiment by accessing the corresponding `task_descriptor`. In this specific instance, the tasks belonging to the *medium* criticality container cause 1,067 overtimes in total, which reflects into total 92 overruns by the tasks within the *low* criticality container. The *high* criticality container is not affected by the faulty tasks, as expected.

Table 2 Outcome of one experiment replication with no monitor and U=0.85.

C_i (ns)	T_i (ns)	U_i	criticality	priority	overtime	overrun
978854	7071458	0.138	*high*	1	0	0
621582	7566834	0.082	*high*	2	0	0
1380333	8008509	0.172	*high*	3	0	0
160056	4869494	0.033	*medium*	4	339	0
34866	6432178	0.005	*medium*	5	220	0
103005	6606403	0.016	*medium*	6	169	0
624445	7667583	0.081	*medium*	7	133	0
123770	8385032	0.015	*medium*	8	101	0
95034	8792447	0.011	*medium*	9	70	0
167473	9991428	0.017	*medium*	10	35	1
77299	5288777	0.015	*low*	11	0	21
250561	6660143	0.038	*low*	12	0	4
1438550	7360892	0.195	*low*	13	0	43
283605	8931703	0.032	*low*	14	1	24

6.2 Results with no monitoring

We discuss the results obtained at the termination of the experimental campaign. Figure 8 summarizes the key outcomes obtained *without* the monitoring approach, i.e., the tasks allotted to the *medium* criticality container are free to exceed their WCET with no temporal separation.



(a) Total overtime *medium* criticality container. **(b)** Total overrun *medium* criticality container. **(c)** Total overrun *low* criticality container.

Figure 8 Total overtime and overrun without our monitoring approach.

Boxplots in Figure 8a summarize the variability of the **total overtime** at increasing utilization U. One observation of the *total overtime* is the sum of the overtime of the tasks allotted to the *medium* criticality container at the end of one experiment: each boxplot is obtained with the data from 30 experiment replications for a given U as stated above.

On average, the tasks produce around 1,027 total overtime *per* experiment across the different levels of utilization U, such as shown by the dotted line in Figure 8a. By means of ANOVA, we observe that there is no statistically significant effect of U on the total overtime with respect to the inherent variability of the overtime across the experiments. This is an expected result that is given by the way we generate the task sets and emulate the faulty tasks.

Similarly to Figure 8a, boxplots in Figure 8b show the **total overrun** – i.e., the sum of the overrun – of the tasks allotted to the *medium* criticality container. Again, each boxplot summarizes the outcomes from 30 experiment replications. Figure 8c shows the total overrun for the *low* criticality container. In both Figure 8b and 8c the dotted line represents the mean of the observations. It can be noted that the total overrun increases as the utilization increases. Noteworthy, this is not *merely* caused by an increase of the total overtime of the tasks in the *medium* criticality container, because we have excluded the existence of a statistically significant trend with respect to U, as stated above. Rather, it can be reasonably stated that at low levels of utilization U (e.g., 0.45-0.55) the remaining amount of free CPU bandwidth provides higher chance to tolerate faults that reflect into exceeding the WCET. This chance is progressively smaller as U increases, which causes higher total overrun.

Regarding the tasks allotted to the *high* criticality container, we observe no overrun. Since this container hosts high-priority tasks – and given that RTAI has a PFP scheduler – even if the tasks in the *medium* criticality exceeds the WCET, they do not affect the *high* criticality container. The effect of priorities can be also noted by comparing Figure 8b and 8c, where the total overrun is smaller for the *medium* criticality container.

6.3 Results and considerations with the proposed monitor

We analyze the outcome of the remaining 150 experiments obtained by monitoring the tasks with the proposed approach, which means that any task exceeding the declared WCET is terminated by its corresponding *control task* upon the receipt of a `SIGNAL`. Differently from the results discussed in Section 6.2, we observe *no* overrun across all the levels of utilization assessed in this case study. Our monitor prevents the overrun affecting the tasks allotted both to the *medium* and *low* criticality container.

Beyond this favorable finding, which pertains the basic functioning of the proposed architecture, we would like to discuss *cons* of our monitor and potential workarounds. At this stage of development, we have pursued a lightweight implementation of the monitor, with a *reactive* protection approach, as it can be noted by the mitigation policies described in Section 5.3. In consequence, it is reasonable to assume the existence of a **latency** of the detection mechanism, i.e., the time between (i) a faulty task exceeding the WCET, and (ii) the effect of the mitigation, e.g., a `KILL`, triggered by our monitor.

Latency may be an issue in some very strict scheduling scenarios. For example, let us hypothesize a scenario – generated with *SimSo* [5] – which consists of six tasks with the parameters shown in Table 3 and total utilization U=1. Figure 9 shows a schedule of the tasks (*CPU* row) and a detailed view for the tasks. It can be noted that there is no timeslot left for tolerating any latency of the detection: for this worst case scenario any minimum delay at terminating the tasks that exceed the WCET will likely cause an overrun.

In principle, it is possible to *tolerate* the latency of the monitor with the following workaround. Let C_{mon} denote a WCET estimation for the *latency* of the detection. Given a task set, for the *sole* purpose of the schedulability test we use the declared WCETs of the tasks *augmented* by C_{mon}, i.e., C_i+C_{mon}. If the task set is admitted, any task can safely exceed its WCET up to C_{mon} since we assure – by construction – enough time for the actions of the monitor to take place. The proposed adjustment of the WCETs has a trade-off, which

Table 3 An example of worst case scenario with U=1.0.

ID	C_i (ms)	T_i (ms)	criticality	priority
Task T1	1.66	10	*high*	1
Task T2	8.33	50	*high*	2
Task T3	8.33	50	*medium*	3
Task T4	16.66	100	*medium*	4
Task T5	25.00	150	*low*	5
Task T6	50.00	300	*low*	6

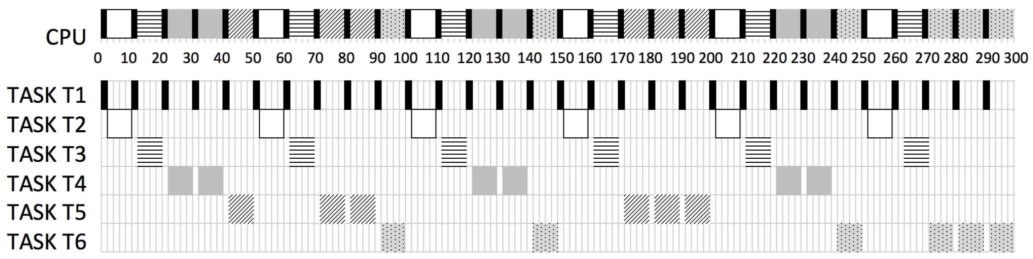


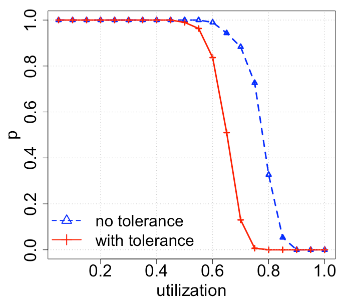
Figure 9 Representation of the worst case scenario (adapted from *SimSo*).

reflects into an inherent increase of the declared utilization of a task set. In the following, we explore this proposition by analyzing the sensitivity of the probability (p) to find schedulable task sets at increasing utilization *with/without* tolerance of the latency.

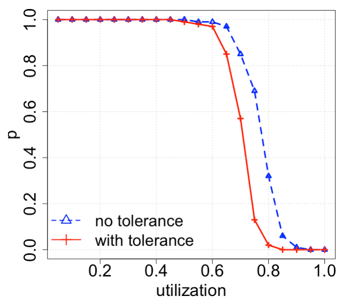
For this analysis we assume the `KILL` policy, i.e., a faulty task is terminated outright by the monitor without the mediation of the corresponding *control task*. In consequence, the duration of the mitigation action – *per se* – is negligible, and the main contribution to C_{mon} consists of the time taken by the monitor to detect a faulty task. As discussed in Section 5.4, at this stage of development C_{mon} depends on both the monitoring period T_{mon} and the number of potential task preemptions, which is a function of the relative priorities and periods of the tasks. In our case study, for simplicity, a reasonable estimate of C_{mon} is around 60,000ns. For example, this value can be obtained by applying the computations described in Section 5.4 to the tasks in Table 2 and taking the mean value across the tasks.

Figure 10a shows the probability (p) to find a schedulable task set with respect to U for 14 tasks and C_{mon}=60,000ns. For each value of utilization U within 0.05 and 1.0 (by step 0.05) we generate 100 task sets according to [10], beforehand. We then run the schedulability test (Alg. 1) with our feasibility checker: p is the ratio between the number of feasible task sets divided by 100. For the dotted time series (i.e., *no tolerance*) WCETs are not modified, while for the solid time series (i.e., *with tolerance*) WCETs are augmented by C_{mon} before assessing the schedulability. As shown in Figure 10a, p decreases as U increases both *with/without* tolerance. We observe that the tolerance does not affect p up to U=0.5, while its impact is significant between 0.6 and 0.9. For example, it can be noted that with *no tolerance*, p approaches 0 at $U = 0.9$, while *with tolerance* is around 0 at $U = 0.75$. Noteworthy, the distance between *no/with tolerance* series depends on n: for example, the series are closer with n=11 – i.e., Figure 10d – at a similar distribution of the tasks across the containers, i.e., 2, 6, and 3 tasks, to the *high*, *medium*, and *low* criticality container, respectively.

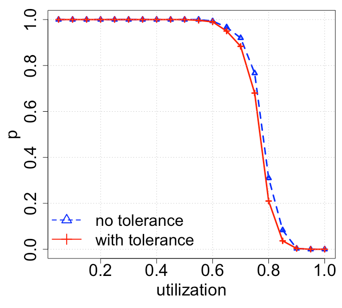
We discuss two workarounds that allow mitigating the loss of utilization. The former, reducing T_{mon}. Although we used 8,000ns in the case study, T_{mon} can be safely set to 4,000ns, which is larger than the WCET of the current algorithm implementation. In



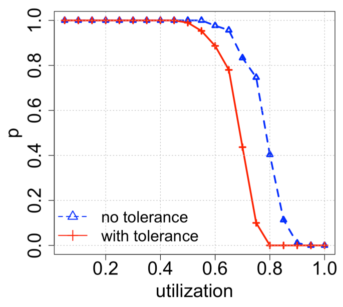
(a) n=14; $C_{mon} = 60,000ns$



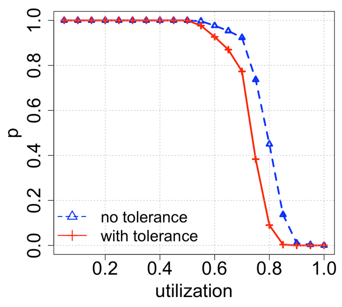
(b) n=14; $C_{mon} = 30,000ns$



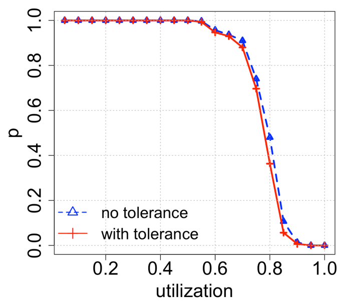
(c) n=14; $C_{mon} = 8,000ns$



(d) n=11; $C_{mon} = 60,000ns$



(e) n=11; $C_{mon} = 30,000ns$



(f) n=11; $C_{mon} = 8,000ns$

Figure 10 Sensitivity of p with respect to the utilization.

this case we obtain C_{mon}=30,000ns. Figure 10b and 10e show how p varies *with/without* tolerance at this lower C_{mon}, and denote a significant improvement with respect to the figures discussed before. The latter, improving the implementation of the monitoring algorithm. According to the discussion in Section 5.4, we are pursuing an implementation that reduces the measurement error on the execution time; at each monitoring activation we plan to leverage the *last_switch_time* of the current rt-task to properly measure the execution time of the previous monitored rt-task. Hence, in the new implementation, C_{mon} would be equal to T_{mon}, which is strongly desirable in practice. In fact, according to the sensitivity analysis presented in Figure 10c and 10f with C_{mon}=8,000ns, *no/with tolerance* series almost overlap. Based on this finding, we are confident that the prospected improvement of the implementation should address the current drawbacks in tolerating the latency.

7 Conclusions

This paper presented the notion and an implementation of real-time containers, or rt-cases, as a possible lightweight solution to let mixed-criticality hard real-time task sets cohabit on the same hardware. Containers are largely adopted in the software industry to modularize application components, to streamline the management of dependencies, and to simplify their deployment and migration in heterogeneous server environments. With rt-cases, we aim to bring these advantages and software management attitudes to real-time mixed criticality systems, allowing for the first time to run hard real-time tasks, from within containers, on low latency Linux systems patched with real-time co-kernels, such as RTAI.

The paper demonstrated the practical feasibility of the rt-case concept in a real context. The main enabling components of the proposed architecture have been implemented in a Linux environment, made publicly available and tested under realistic and feasible task sets. Preemptive fixed priority scheduling has been chosen as underlying task scheduling solution, to avoid modifications to the co-kernel support, thus fostering the early adoption of the solution. As drawbacks, we have observed that the measurement error committed by the monitor can affect significantly the feasibility of the task set when increasing the CPU utilization. In addition, we noted that the priority assignment algorithm currently implemented in the feasibility checker may not respect the relative ordering of task priorities, originally planned by the developer. To mitigate these problems, future work will be devoted to the implementation of the improved version of the monitoring algorithm that reduces the measurement error, as described in section 5.4, and to the development of the design alternative based on the use of task groups and hierarchical scheduling with server-based approaches, which makes priority assignments within rt-cases independent from the feasibility checker. We also plan to test the solution with realistic workloads in the context of the ITER project [26], thanks to an on-going research collaboration.

References

1 L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, pages 4–13, December 1998. doi:10.1109/REAL.1998.739726.

2 N.C. Audsley. Optimal Priority Assignment And Feasibility Of Static Priority Tasks With Arbitrary Start Times. *Technical report YCS 164*, 1991.

3 Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM. doi:10.1145/945445.945462.

4 Alan Burns and Robert I. Davis. Mixed Criticality Systems – A review. *Tech Rep of the University of York*, 2018. URL: https://www-users.cs.york.ac.uk/burns/review.pdf.

5 Maxime Chéramy, Pierre-Emmanuel Hladik, and Anne-Marie Déplanche. SimSo: A Simulation Tool to Evaluate Real-Time Multiprocessor Scheduling Algorithms. In *Proc. of the 5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, WATERS, 2014.

6 M. Cinque and D. Cotroneo. Towards Lightweight Temporal and Fault Isolation in Mixed-Criticality Systems with Real-Time Containers. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 59–60, June 2018. doi:10.1109/DSN-W.2018.00029.

7 M. Cinque and G. De Tommasi. Work-in-Progress: Real-Time Containers for Large-Scale Mixed-Criticality Systems. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 369–371, December 2017. doi:10.1109/RTSS.2017.00046.

8 N. T. Dantam, D. M. Lofaro, A. Hereid, P. Y. Oh, A. D. Ames, and M. Stilman. The Ach Library: A New Framework for Real-Time Communication. *IEEE Robotics Automation Magazine*, 22(1):76–85, March 2015. doi:10.1109/MRA.2014.2356937.

9 Z. Deng and J. W. . Liu. Scheduling real-time applications in an open environment. In *Proceedings Real-Time Systems Symposium*, pages 308–319, December 1997. doi:10.1109/REAL.1997.641292.

10 P. Emberson, R. Stafford, and R.I. Davis. Techniques For The Synthesis Of Multiprocessor Tasksets. In *WATERS workshop at the Euromicro Conference on Real-Time Systems*, pages 6–11, July 2010.

11 G. Farrall, C. Stellwag, J. Diemer, and R. Ernst. Hardware and software support for mixed-criticality multicore systems. In *Proc. of the Conference on Design, Automation and Test in Europe, WICERT, DATE*, 2013.

12 G. Garre, D. Mundo, M. Gubitosa, and A. Toso. Real-Time and Real-Fast Performance of General-Purpose and Real-Time Operating Systems in Multithreaded Physical Simulation

of Complex Mechanical Systems. *Mathematical Problems in Engineering, Article ID 945850*, 2014. doi:10.1155/2014/945850.

13 M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, January 1986. doi:10.1093/comjnl/29.5.390.

14 R. Kaiser. The PikeOS concept history and design. *Technical Report, SYSGO*, 2007.

15 K. Lakshmanan, D. d. Niz, R. Rajkumar, and G. Moreno. Resource Allocation in Distributed Mixed-Criticality Cyber-Physical Systems. In *2010 IEEE 30th International Conference on Distributed Computing Systems*, pages 169–178, June 2010. doi:10.1109/ICDCS.2010.91.

16 Juri Lelli, Claudio Scordino, Luca Abeni, and Dario Faggioli. Deadline scheduling in the Linux kernel. *Software: Practice and Experience*, 46(6):821–839, 2016. doi:10.1002/spe.2335.

17 G. Lipari and S. Baruah. A hierarchical extension to the constant bandwidth server framework. In *Proceedings Seventh IEEE Real-Time Technology and Applications Symposium*, pages 26–35, May 2001. doi:10.1109/RTTAS.2001.929863.

18 C. Mao, M. Huang, S. Padhy, S. Wang, W. Chung, Y. Chung, and C. Hsu. Minimizing Latency of Real-Time Container Cloud for Software Radio Access Networks. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 611–616, November 2015. doi:10.1109/CloudCom.2015.67.

19 Philip Masek, Magnus Thulin, Hugo Sica de Andrade, Christian Berger, and Ola Benderius. Systematic Evaluation of Sandboxed Software Deployment for Real-time Software on the Example of a Self-Driving Heavy Vehicle. *CoRR*, abs/1608.06759, 2016. arXiv:1608.06759.

20 Miguel Masmano, Ismael Ripoll, Alfons Crespo, and J Metge. Xtratum: a hypervisor for safety critical embedded systems. In *11th Real-Time Linux Workshop*, pages 263–272. Citeseer, 2009.

21 M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos. Mixed-Criticality Real-Time Scheduling for Multicore Systems. *10th IEEE International Conference on Computer and Information Technology, Bradford, pp. 1864-1871*, 2010.

22 R. Santos, S. Venkataraman, A. Das, and A. Kumar. Criticality-aware scrubbing mechanism for SRAM-based FPGAs. *Technical report, Nanyang Technological University, Singapore*, 2014.

23 Brinkley Sprunt, Lui Sha, and John Lehoczky. Aperiodic task scheduling for Hard-Real-Time systems. *Real-Time Systems*, 1(1):27–60, June 1989. doi:10.1007/BF02341920.

24 X. Wang, Z. Li, and W. M. Wonham. Optimal Priority-Free Conditionally-Preemptive Real-Time Scheduling of Periodic Tasks Based on DES Supervisory Control. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 47(7):1082–1098, July 2017. doi:10.1109/TSMC.2016.2531681.

25 WindRiver. VxWorks Virtualization Profile. http://www.windriver.com/products/vxworks/technology-profiles/#virtualization. [Online; accessed 15-Jan-2019].

26 A. Winter, P. Makijarvi, S. Simrock, J.A. Snipes, A. Wallander, and L. Zabeo. Towards the conceptual design of the ITER real-time plasma control system. *Fusion Engineering and Design*, 89(3):267–272, 2014. doi:10.1016/j.fusengdes.2014.02.064.

27 S. Xi, C. Li, C. Lu, C. D. Gill, M. Xu, L. T. X. Phan, I. Lee, and O. Sokolsky. RT-Open Stack: CPU Resource Management for Real-Time Cloud Computing. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 179–186, June 2015. doi:10.1109/CLOUD.2015.33.

28 S. Xi, M. Xu, C. Lu, L. T. X. Phan, C. Gill, O. Sokolsky, and I. Lee. Real-time multi-core virtual machine scheduling in Xen. In *2014 International Conference on Embedded Software (EMSOFT)*, pages 1–10, October 2014. doi:10.1145/2656045.2656061.

29 Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. RT-Xen: Towards Real-time Hypervisor Scheduling in Xen. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 39–48, New York, NY, USA, 2011. ACM. doi:10.1145/2038642.2038651.

Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling

Daniel Casini
Scuola Superiore Sant'Anna, Pisa, Italy
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany
daniel.casini@sssup.it

Tobias Blaß
Corporate Research, Robert Bosch GmbH, Renningen, Germany
Saarland Informatics Campus, Saarland University, Saarbrücken, Germany
tobias.blass@de.bosch.com

Ingo Lütkebohle
Corporate Research, Robert Bosch GmbH, Renningen, Germany
ingo.luetkebohle@de.bosch.com

Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany
bbb@mpi-sws.org

Abstract

Bounding the end-to-end latency of processing chains in distributed real-time systems is a well-studied problem, relevant in multiple industrial fields, such as automotive systems and robotics. Nonetheless, to date, only little attention has been given to the study of the impact that specific frameworks and implementation choices have on real-time performance. This paper proposes a scheduling model and a response-time analysis for ROS 2 (specifically, version "Crystal Clemmys" released in December 2018), a popular framework for the rapid prototyping, development, and deployment of robotics applications with thousands of professional users around the world. The purpose of this paper is threefold. Firstly, it is aimed at providing to robotic engineers a practical analysis to bound the worst-case response times of their applications. Secondly, it shines a light on current ROS 2 implementation choices from a real-time perspective. Finally, it presents a realistic real-time scheduling model, which provides an opportunity for future impact on the robotics industry.

2012 ACM Subject Classification Software and its engineering → Real-time schedulability

Keywords and phrases ROS, real-time systems, response-time analysis, robotics, resource reservation

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.6

Supplement Material ECRTS 2019 Artifact Evaluation approved artifact available at `https://dx.doi.org/10.4230/DARTS.5.1.5`
The associated source code is available at `https://github.com/boschresearch/ros2_response_time_analysis`.

1 Introduction

ROS, the *Robot Operating System* [43], is one of the most popular frameworks for designing and developing Linux-based robots. Powering over 100 different robot designs, it is used by tens of thousands of developers and researchers in both industry and academia [4, 22]. However, after over a decade of development and in the face of increasingly demanding applications, it became clear to the ROS community that the framework is held back by several long-standing shortcomings and architectural limitations that cannot be rectified in a backwards-compatible manner. This motivated the development of ROS 2, a complete refactoring of ROS that puts the successful concept onto a modernized and improved foundation. Of particular interest to us, a major design goal of ROS 2 is to improve the real-time capabilities of the framework, enabling the implementation of time-critical control paths inside ROS [24].

Safely implementing such control paths requires predicting the end-to-end latency (or *response time*) of time-critical *processing chains*. For instance, such a chain might cover all steps from a sensor, via a controller, to the final actuator and span multiple software components, multiple cores, and even multiple hosts. Although the end-to-end latency problem is well-studied in the literature, existing work often assumes idealized scheduling models that are not always directly applicable in real systems. Case in point, the ROS[1] scheduling approach (as described in detail in Section 3) does not match any of the classic results on bounding end-to-end latencies. ROS developers therefore have to resort to fully prototyping and deploying a design to measure its timing properties, which severely limits the degree to which the design space can be explored in practice.

While the scheduling model of operating systems like Linux has been studied extensively, this is not the case for ROS. Even though it is a middleware layer and not a proper operating system, its effects on an application's runtime behavior are substantial, rivaling or even exceeding that of the underlying OS. For example, ROS multiplexes independent message handlers onto shared threads using custom scheduling policies. Consequently, applications running on top of ROS are subject to the scheduling decisions of the underlying operating system *and* the middleware layer, with complex and interdependent effects on timing.

An additional complication stems from one of the key strengths of ROS: its modular structure. ROS emphasizes composing existing, battle-tested components instead of reimplementing common subsystems for each robot from scratch. While this greatly simplifies and speeds up robot development, it also obfuscates the overall timing behavior. This problem is further aggravated by ROS' event-driven design style, which gives rise to data dependencies and potentially long processing chains. As a result, it is extremely difficult for developers to anticipate, or even just understand, the timing of processing chains that cross multiple, loosely coupled components, most of which are developed by independent teams all around the world. Realistically, automated end-to-end response-time analysis is thus required to safely employ ROS in time-critical situations.

In this paper, we seek to lay the theoretical foundations for such an automated analysis tool by exploring the temporal behavior of ROS 2 "Crystal Clemmys" (released in December 2018) [5]. We present and validate a model of ROS applications running on top of a resource reservation scheduler such as `SCHED_DEADLINE` in Linux. Based on this model, we develop an end-to-end response-time analysis for ROS processing chains that takes the peculiarities and engineering constraints of the ROS environment into account. Finally, to demonstrate the applicability of our analysis to practical ROS components, we evaluate our approach on the popular `move_base` package [3], the core of the ROS navigation stack.

2 Background

This section introduces necessary background on the three pillars on which this paper rests. First, the structure of ROS and its execution model are presented. Then, we review resource reservations, an OS-level mechanism to isolate the resource consumption of processes, and last we review the Compositional Performance Analysis approach for response-time analysis.

[1] For brevity, we omit the version number and refer to ROS 2 as ROS in the remainder of the paper.

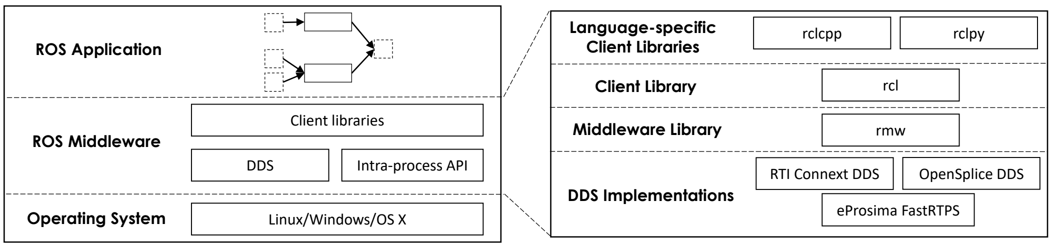


Figure 1 Layered structure of ROS.

2.1 ROS

ROS places great emphasis on modularity and composability. It therefore encourages strict separation between the logical structure of the application and the mapping of this structure onto hosts, processors, and threads. While the former is defined by the package developer, the latter is entirely up to the system integrator. This way, software modules can be developed independently of the target platform without losing the ability to taylor them to the deployment characteristics of a particular robot.

From a logical perspective, ROS applications are composed of *nodes*, the smallest self-contained units of behavior. These nodes communicate using the publish-subscribe paradigm: nodes publish messages on *topics*, which broadcast the message to all nodes that are subscribed to the topic. Nodes react to incoming messages by activating *callbacks* to process each message. Since these callbacks may publish messages themselves, complex behavior can be implemented as a network of topics and callbacks. ROS also allows callbacks to invoke remote procedure calls by means of the *service mechanism* using a continuation-passing style. Specifically, a callback can initiate a non-blocking service request to a *service* callback and specify a third *client* callback to be invoked once the response is available.

ROS seamlessly allows composing nodes written in different programming languages and using different communication backends. The ROS implementation is therefore split into multiple layers of abstraction, which are visualized in Figure 1. Each supported programming language requires a client library that provides a language-specific API to the ROS application model. The ROS project officially supports C++ and Python, with community-provided support for numerous other languages. Below the surface, these libraries use a common system model provided by the *rcl* library. This ensures consistent behavior between the languages and reduces code duplication.

Despite this unified implementation, some parts of the ROS system are allowed to differ between languages. In particular, client libraries have a lot of freedom in implementing the execution model to allow the callback graph to be expressed in the most natural way in each language. A language supporting coroutines, for example, might allow coroutines as event handlers instead of callbacks. We therefore limit the focus of this paper to the C++ interface, which we believe to be the most likely choice for time-critical components.

For inter-node communication, ROS uses the Data Distribution Service [39] (DDS), an industry standard for data distribution in real-time systems. DDS specifies a network-transparent publish-subscribe mechanism that can be adapted to the needs at hand using a rich set of *Quality-of-Service* (QoS) policies. ROS works with different, independent implementations of the DDS standard (currently, FastRTPS by eProsima [2], Connext by RTI [6], and Vortex OpenSplice by Adlink [7]), each with a different API. The *rcl* client library therefore accesses the DDS subsystem over the common *rmw* (ROS MiddleWare) interface, which provides a DDS-agnostic API to the *rcl* layer. Each supported DDS implementation requires a dedicated *rmw* implementation, which translates between the common *rmw* interface and the vendor-specific DDS API.

To deploy a ROS application, the individual nodes have to be distributed to hosts and then mapped onto operating system processes. ROS does not impose any restrictions on this mapping. Processes implement the ROS execution model by running *executors*, which receive messages from *rcl* and invoke the corresponding callbacks. ROS provides two built-in executors: a sequential one that executes callbacks in a single thread, and a parallel one that distributes the processing of pending callbacks across multiple threads. Moreover, ROS supports arbitrarily complex setups of multiple, user-defined executors.

From a real-time perspective, it is important to note that executors implement custom scheduling policies; we revisit this issue in Section 3 in detail. Furthermore, how each ROS executor's threads are scheduled by the OS has a major impact on the overall timing behavior of the application. To ensure predictable scheduling, executor threads can be bound to a reservation server, which is a pragmatic configuration approach to increasing predictability that we advocate in this paper. Next, we briefly review key aspects of resource reservations.

2.2 Resource Reservations

An ideal mechanism to ensure predictable service for ROS threads is a *resource reservation*, which is a classic OS-level abstraction that limits interference between processes by bounding their resource consumption. Resource reservations are typically implemented by reservation servers. In general, a reservation server r_i is characterized by a budget Q_i and a period P_i, and guarantees that its client threads receive Q_i units of execution time in each period.

Many different reservation algorithms have been designed and developed over the last 30 years [12], with various different features and support for different scheduling algorithms (e.g., fixed-priority, deadline-based scheduling, etc.). This paper does not focus on any specific algorithm, but we assume the reservation server to comply with the periodic resource model [58], namely, we require that: **(i)** each reservation has an implicit deadline, **(ii)** whenever r_i has workload, the reservation algorithm guarantees at least Q_i units of service every P_i time units, and **(iii)** there exists a bounded maximum service delay, i.e., a bounded maximum release delay that a process running in a reservation can experience because of budget exhaustion and delays due to other reservations. Under these assumptions, the minimum amount of service provided by the reservation in an interval of length Δ can be expressed with a supply-bound function $sbf(\Delta)$ [33, 58]. In the following, we assume that such a supply bound function is known for each reservation and refer to [12, 15, 33, 58] for a discussion of how to obtain them.

On Linux systems, resource reservations are available through the `SCHED_DEADLINE` [32] scheduling class, which implements the *Constant Bandwidth Server* [8] reservation algorithm. Moreover, as a special case of practical relevance, a thread running on a dedicated core at the highest priority of the `SCHED_FIFO` scheduling class can be considered as running in a reservation with the supply bound function $sbf(\Delta) = \Delta$.

Next, to complete the overview of needed background, we review the Compositional Performance Analysis approach, upon which we base our response-time analysis.

2.3 Compositional Performance Analysis

Compositional Performance Analysis (CPA) is an approach to analytically evaluate the performance of heterogeneous and distributed systems [27]. CPA models systems as networks of resources, and workloads as tasks with dependencies. Resources provide processing time, which is consumed by tasks. Tasks with dependencies are organized as a direct acyclic graph, and paths in the graphs are denoted as *processing chains*. Tasks sharing the same resource are scheduled according to a resource-specific scheduling policy. The source task of a chain is triggered according to an *externally provided event arrival curve* [27, 30, 62] $\eta^e(\Delta)$, which defines an upper bound on the number of events that can arrive in any time window $[t, t+\Delta)$. Event arrival curves are general enough to model both periodic and event-driven (e.g., interrupt-driven) activations [47]. For example, when a task T_x is periodically triggered, its arrival curve can be expressed as $\eta^e_x(\Delta) = \left\lceil \frac{\Delta}{period(x)} \right\rceil$. Non-source tasks are triggered according to *derived activation curves*. Activation curves are obtained from arrival curves by accounting for release jitter, which reflects the activation delay due to predecessor tasks and depends on their response times. However, response times also depend on the release jitter, thus creating a cyclic dependency. To solve this problem, the analysis starts with an initial jitter of zero, and then iteratively applies the response-time analysis and updates all jitter bounds until convergence is achieved [42, 63]. Convergence is guaranteed (for non-overloaded systems) by the monotonic dependency between response time and jitter (the more jitter, the higher the response times, and vice versa). The basic CPA approach bounds the end-to-end latency of a chain with the sum of the individual response-time bounds of each task. Extensions have been subsequently designed to improve analysis precision, e.g., [54, 56].

Since ROS provides executors that dispatch callbacks in peculiar ways using custom scheduling policies, the existing CPA literature and tooling is not a perfect match for ROS. However, we liberally take inspiration from CPA to obtain a similarly flexible timing model that reflects the idiosyncrasies of ROS, which we introduce next.

3 ROS Scheduling

As described in Section 2.1, the ROS execution model multiplexes all callbacks associated with an executor onto one or more threads. The ROS C++ library provides its built-in executor in two variants: a single-threaded and a multi-threaded one. In this initial study of the ROS timing behavior, we focus exclusively on the simpler and more predictable single-threaded executor. The following description is based on a careful study of the ROS source code and documentation, and is to our knowledge the first comprehensive description of the scheduling behavior of ROS. To validate our observations, we conclude this section with an experiment that demonstrates and corroborates our findings on a concrete example.

The executor distinguishes four categories of callbacks: *timers*, which are triggered by system-level timers, *subscribers*, which are triggered by new messages on a subscribed topic, *services*, which are triggered by service requests, and *clients*, which are triggered by responses to service requests. The executor is responsible for taking messages from the input queues of the DDS layer (by interacting with the *rcl* layer) and executing the corresponding callback. Since it executes callbacks to completion, it is a non-preemptive scheduler. However, unlike most commonly studied schedulers, it does not always consider all ready tasks for execution. Instead, it bases its decisions on the *readySet*, a cached copy of the set of ready non-timer callbacks, which it updates in irregular, execution-dependent intervals. The algorithm is depicted in Figure 2, in which we assume $\mathcal{C}$ to be the set of all callbacks assigned to the executor, and $\mathcal{C}^{\text{tmr}}$, $\mathcal{C}^{\text{sub}}$, $\mathcal{C}^{\text{srv}}$, $\mathcal{C}^{\text{clt}}$ to be the subsets of $\mathcal{C}$ consisting only of timers, subscribers, services, and clients, respectively.

If the executor is idle, it updates its *readySet*. This is the only step in which the executor interacts with the underlying communication layer (i.e., *rmw*, via *rcl*). It then looks for a callback to execute by searching through the four callback categories (for efficiency, the

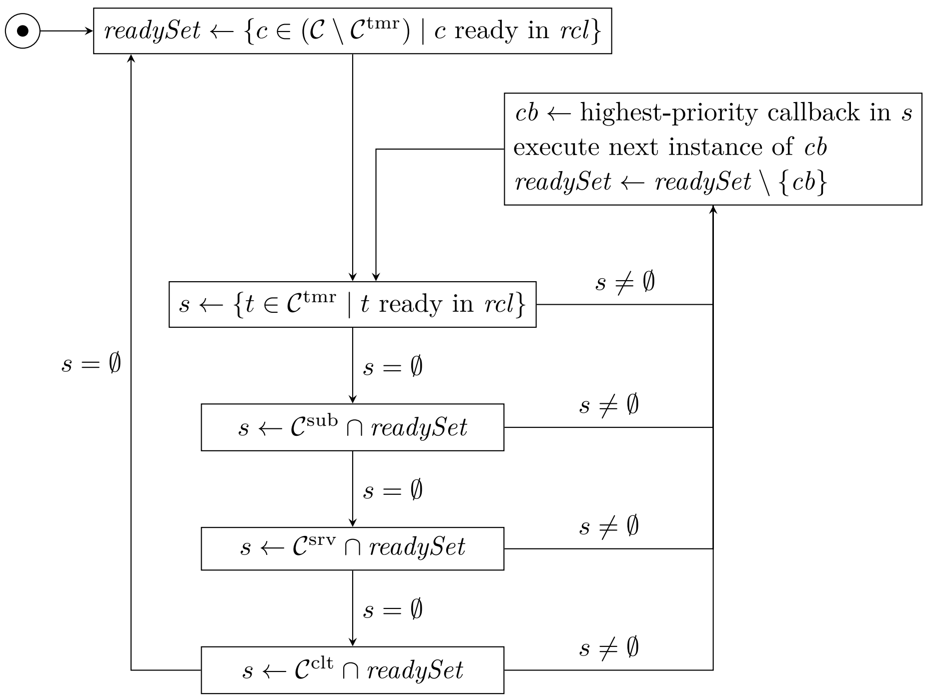

$readySet \leftarrow \{c \in (\mathcal{C} \setminus \mathcal{C}^{\text{tmr}}) \mid c \text{ ready in } rcl\}$
$cb \leftarrow$ highest-priority callback in s
execute next instance of cb
$readySet \leftarrow readySet \setminus \{cb\}$
$s \leftarrow \{t \in \mathcal{C}^{\text{tmr}} \mid t \text{ ready in } rcl\}$
$s \neq \emptyset$
$s = \emptyset$
$s \leftarrow \mathcal{C}^{\text{sub}} \cap readySet$
$s \leftarrow \mathcal{C}^{\text{srv}} \cap readySet$
$s \leftarrow \mathcal{C}^{\text{clt}} \cap readySet$

Figure 2 The executor scheduling algorithm.

executor blocks if there is nothing to do; this optimization has been omitted for clarity). It first checks whether any timers have expired. Since these are not managed by the DDS layer, this check is based on the current timer state and does not depend on the *readySet*. It then searches the *readySet* for subscriptions, services, and clients (in this order). Evaluating the queues in a fixed order has the intrinsic effect of assigning each queue a different priority (i.e., the timer queue is examined first and hence has the highest priority, and the client queue is examined last and has the lowest priority). When a queue is considered, callback instances are examined in callback registration order, i.e., the order in which the callbacks were registered with the executor. Consequently, the registration order represents a second level of priorities. Overall, the pair (callback type, registration time) is a unique priority for each callback.

Whenever a category has at least one ready callback, the highest-priority one is selected, executed, and then removed from the *readySet*. Finally, when the *readySet* is empty and no expired timers are left, the executor returns to the idle state and updates the *readySet* based on a current snapshot of the communication layer. We refer to the updating of the *readySet* as a *polling point* and the interval between two polling points as a *processing window*. The n-th polling point is referred to as PP_n, and the n-th processing window (ranging from PP_n to PP_{n+1}) as PW_n.

Compared to regular fixed-priority scheduling, this algorithm exhibits a few unusual properties. First, messages arriving during a processing window are not considered until the next polling point, which depends on *all* remaining callbacks. This leads to priority inversion, as lower-priority callbacks may implicitly block higher-priority callbacks by prolonging the current processing window.

Second, it relies on a ready *set* instead of the more usual ready *list*. This means that the algorithm cannot know how *many* instances of any non-timer callback are ready. It therefore processes at most one instance of any callback per processing window. This aggravates the

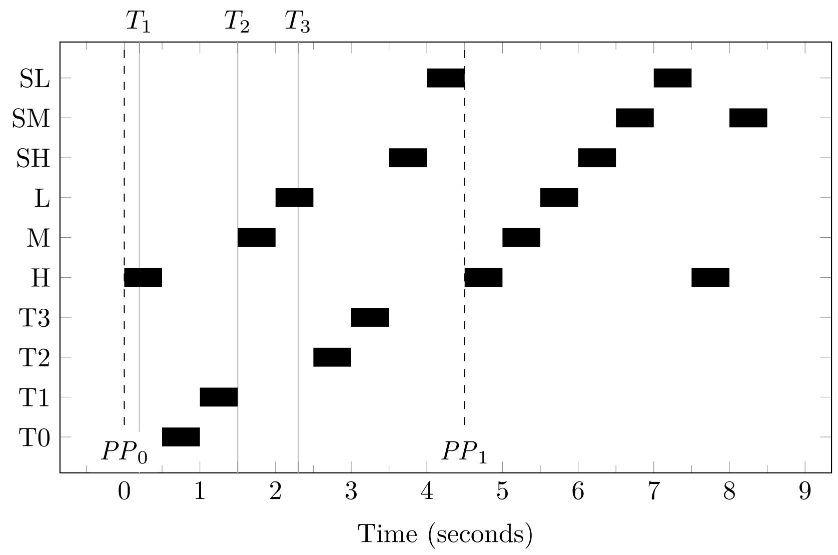


Figure 3 Gantt-Chart of the scheduler validation test. At times T_1 and T_3, the timers trigger. At time T_2, the second batch of service requests and messages is submitted.

priority inversion above, as a backlogged callback might have to wait for *multiple* processing windows until it is even considered for scheduling. Effectively, this means that a non-timer callback instance might be blocked by multiple instances of the same lower-priority callback.

The presented description of the ROS scheduler is based on manual code inspection. In a system as complex as ROS, however, this is potentially error-prone, as there might be subtle interactions that are easily overlooked yet change the behavior drastically. Thus, to validate our model, we implemented a special-purpose ROS node that executes arbitrary-length callbacks in a way that allows inferring the behavior of the ROS scheduler from the resulting trace. Specifically, the node is controlled using three topics (*H*, *M*, and *L*), three services (*SH*, *SM*, and *SL*), and a special-purpose topic to create timers. Note that the chosen names assume that topics and services are prioritized in registration order; checking that topic *H* actually has the highest priority is part of the model validation. In the following description, time zero refers to the point in time when the first batch of validation callbacks arrives at the node. The i-th timer is denoted as t_i. For ease of visualization, all callbacks run for 500ms.

Our test first sets up two timers at 200ms (T_0) and two timers at 2300ms (T_3). It then sends the message sequence <*L M H SH SL L M H SH SL*>, waits for 1.5 seconds (T_2), and then sends <*SM SM H*>. The result is visualized in Figure 3. Note that the polling points are not determined by the test; rather, they are inferred from the resulting timing behavior.

One can clearly observe the scheduler executing only a single callback per ready event, even if multiple messages have been queued up; this is especially apparent after the second polling point. Furthermore SM is visibly skipped at time 4, even though it arrives earlier at T_2 (i.e., during the execution of t_1). This proves the existence of polling points. The timers, however, are clearly not subject to these polling points, since t_2 and t_3 arrive later than *SM* but are still executed during the first processing window.

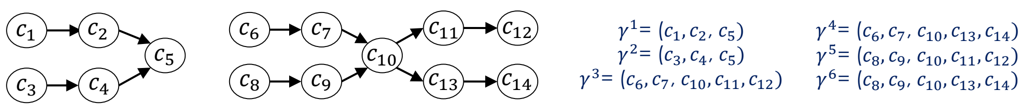

c_1
c_2
c_3
c_4
c_5
c_6
c_7
c_8
c_9
c_{10}
c_{11}
c_{12}
c_{13}
c_{14}
$\gamma^1 = (c_1, c_2, c_5)$
$\gamma^2 = (c_3, c_4, c_5)$
$\gamma^3 = (c_6, c_7, c_{10}, c_{11}, c_{12})$
$\gamma^4 = (c_6, c_7, c_{10}, c_{13}, c_{14})$
$\gamma^5 = (c_8, c_9, c_{10}, c_{11}, c_{12})$
$\gamma^6 = (c_8, c_9, c_{10}, c_{13}, c_{14})$

Figure 4 Example of a ROS graph. Circles represent callbacks, and edges represent communication relations among them. The corresponding processing chains are also shown.

4 System Model

In this section, we introduce a model of the timing-related aspects of a ROS system, its callbacks, and their activation relations. Table 1 summarizes our notation.

We model a ROS system as a direct acyclic graph (DAG) $\mathcal{D} = \{\mathcal{C}, \mathcal{E}\}$ composed of a set of callbacks $\mathcal{C} = \{c_1, \ldots, c_n\}$ and a set of directed edges $\mathcal{E} \subseteq \mathcal{C} \times \mathcal{C}$. We assume the graph $\mathcal{D}$ to be fixed, i.e., callbacks can neither join nor leave the system at runtime. Recall from Section 3 that $\mathcal{C}^{\mathrm{tmr}}$, $\mathcal{C}^{\mathrm{sub}}$, $\mathcal{C}^{\mathrm{clt}}$, and $\mathcal{C}^{\mathrm{srv}}$ denote the subsets of all timer, subscriber, client, and service callbacks, respectively.

Each callback $c_i \in \mathcal{C}$ has a worst-case execution time e_i, a unique priority π_i, and releases a potentially infinite sequence of instances. We assume a discrete-time model, that is, all time parameters are integer multiples of some basic time unit (e.g., a processor cycle).

Depending on its type, a callback instance is activated when the DDS layer receives a message or a timer expires. When an instance of a callback is activated, it is said to be *pending*, and it remains pending until it completes. A callback instance is said to be *ready* when it is pending but not executing. Each edge $(c_i, c_j) \in \mathcal{E}$ encodes an activation relation from callback c_i to callback c_j, meaning that during the execution of an instance of c_i it activates up to one instance of c_j (e.g., by publishing a message to the topic to which c_j is subscribed). Each callback is associated with a set of predecessors $\mathit{pred}(c_i) = \{c_j \in \mathcal{C} :\ \exists\ (c_j, c_i) \in \mathcal{E}\}$ and a set of successors $\mathit{succ}(c_i) = \{c_j \in \mathcal{C} :\ \exists\ (c_i, c_j) \in \mathcal{E}\}$. A callback without predecessors (respectively, successors) is said to be a *source callback* (respectively, *sink callback*).

Processing chains. The ROS graph $\mathcal{D}$ can have multiple source and sink callbacks. Each source originates one or more *callback chains* $\gamma^x = (c_s, \ldots, c_e)$, i.e., directed paths in the graph. The set of all chains of the graph from a source callback to any other callback is denoted by $\mathit{chains}(\mathcal{D}) = \{\gamma^1, \ldots, \gamma^s\}$. Callbacks can be shared by multiple chains. An example of a ROS graph with several chains is shown in Figure 4.

Activation model. As in CPA, each source callback c_s is associated with a given external event arrival curve $\eta_s^e(\Delta)$, denoting the maximum number of instances of c_s that can be released in any interval of length Δ, while non-source callbacks are associated with a (derived) activation curve. We assume w.l.o.g. that $\eta_s^e(\Delta) > 0$ for $\Delta > 0$.

As discussed in Section 2.1, non timer-callbacks are activated in a data-driven fashion. Our model assumes that callbacks belong to a single timer, topic, or service. (This is not a restriction, since two callbacks may execute the same code.) Consequently, a callback may have multiple incoming edges only if it subscribes to a topic with multiple publishers (similarly to what is referred to as `OR`-activation semantics in other work [27]). In this case, all subscribers are triggered once for each message published to the topic. The derivation of activation curves for callbacks with multiple incoming edges is discussed in Section 5.1.

Table 1 Summary of Notation.

Symbol	Description	Symbol	Description
c_i	the i-th callback	γ^x	the x-th processing chain
r_k	the k-th reservation	$\gamma^{x,y}$	the y-th subchain of γ^x
$\mathcal{C}_k$	set of callbacks in reservation r_k	η_i^e	external arrival curve of c_i
$\delta_{i,j}$	propagation delay from c_i to c_j	η_i^a	derived activation curve of c_i
A	an offset into a busy window	$sbf_k(\Delta)$	supply-bound function of r_k
$R_i^*(A)$	least positive solution of c_i's	$rbf_i(\Delta)$	request-bound function of c_i
	response-time equation for offset A	$RBF(C,\Delta)$	$\sum_{c_i \in C} rbf_i(\Delta)$

Scheduling of executors. As discussed in Section 3, this paper adopts the built-in single-threaded executor. To compel the OS to guarantee predictable service to ROS executors, we assume each executor (i.e., each thread) to be assigned to a single reservation server, and each reservation server to handle a single executor. Consequently, callbacks assigned to an executor can equivalently be considered as assigned to the corresponding reservation server.

The system comprises a set of w reservations $\mathcal{R} = \{r_1, \ldots, r_w\}$, and the set of all the callbacks assigned to reservation r_k is denoted $\mathcal{C}_k$. Analogously, the sets of all timers, subscriptions, clients and services allocated to reservation r_k are denoted as $\mathcal{C}_k^{\text{tmr}}$, $\mathcal{C}_k^{\text{sub}}$, $\mathcal{C}_k^{\text{clt}}$, and $\mathcal{C}_k^{\text{srv}}$, respectively. The symbols $lp_k(c_i)$ and $hp_k(c_i)$ denote the set of callbacks in $\mathcal{C}_k$ with lower and higher priority than π_i, respectively. The reservations are partitioned onto a set of m processors $\mathcal{P} = \{p_1, \ldots, p_m\}$, i.e., each reservation is statically assigned to a processor.

The results presented in this paper rely on the availability of a supply-bound function $sbf_k(\Delta)$ denoting the minimum service provided by a reservation r_k in any interval of length Δ (recall Section 2.2). Whenever multiple reservations are allocated to the same processor, the resource provisioning described by the supply-bound function is guaranteed only if all reservations are schedulable, i.e., they are always able to provide their complete budget during each period [33]. In this paper, reservations are assumed to be schedulable. The problem of guaranteeing reservations to meet their timing constraints is also referred to as *global schedulability* in prior works (e.g., in the context of hierarchical scheduling [9]). Many results are available to ensure global schedulability, e.g., [37].

Propagation delay. The propagation delay between the publication of a message by a sending callback and the activation of an associated receiving callback can be significant. Indeed, due to the inherently distributed topology of ROS systems, the message exchange can involve the network, introducing additional latencies. To model such a delay, each pair of reservations (r_x, r_y) is characterized by a (DDS-dependent) worst-case communication delay $\delta_{i,j}$, denoting the maximum time experienced by a message sent from the DDS layer of a sending callback c_i allocated to r_x until being received by the DDS layer of a receiving callback c_j allocated to r_y, i.e., the maximum additional delay experienced by c_j before being activated. When $r_x = r_y$, $\delta_{i,j}$ is assumed to be negligible. This delay can be either analytically upper-bounded for different types of networks (e.g., see [17, 19, 51]), or pragmatically measured, depending on the requirements of the target application domain.

Event sources. With the exception of timers, all callback types provided by ROS implement data-driven activation semantics. Consequently, all chains comprised solely of ROS callbacks are initially triggered by a timer. Nonetheless, applications often have to react to external events that are delivered asynchronously via interrupts (e.g., certain sensors, network packets delivering inputs from supervisory controllers or human operators, etc.). To integrate such events in our ROS model, we allow external threads to interact with ROS and model them as pseudo-callbacks. Specifically, we name these threads *event sources*. An event source is a regular OS-level thread that is sporadically activated, and interacts with ROS by publishing to one or more topics, thus acting as an interface or ingress point for external events. As we do in the case of executors, we assume each event source to be exclusively assigned to a dedicated reservation. For notational convenience, we let $\mathcal{C}^{\text{evt}}$ denote the set of all event sources and refer to event sources as callbacks $c_i \in \mathcal{C}^{\text{evt}}$.

5 Response-Time Analysis for Processing Chains

This section presents an analysis of the end-to-end delay (i.e., the maximum response time) of a generic ROS processing chain. Our analysis is inspired by the CPA approach (described in Section 2.3), whose event-propagation mechanism is a natural fit for the distributed and message-based nature of ROS. A discussion of possible alternatives is postponed to Section 7.

As in CPA, a complex ROS graph is analyzed by computing individual response-time upper bounds for each callback. End-to-end latencies can then be obtained by summing the individual response times of the callbacks of each chain [27]. Unfortunately, none of the existing instantiations of CPA can compute these per-callback response times, as they are unaware of the peculiarities of the ROS scheduling mechanism (e.g., polling points). We therefore present a ROS-specific response-time analysis for callbacks in Section 5.2.

Although this approach provides a safe and simple upper bound on end-to-end latencies, the resulting bounds may be overly pessimistic if arrival bursts of interfering callbacks are accounted for multiple times, once for each callback in the chain under analysis. This effect is known in the literature as the "pay-burst-only-once" problem [30, 56]. To improve the accuracy of our analysis, Section 5.4 presents a bound in which portions of chains, named *subchains*, are analyzed in a holistic way. We define the y-th subchain $\gamma^{x,y}$ of γ^x as a sequence of consecutive callbacks $c_i \in \gamma^x$ of the original chain that are allocated to a single reservation r_k, i.e., $c_i \in \gamma^{x,y} \Rightarrow c_i \in \mathcal{C}_k$. With this approach, arrival bursts are accounted for only once per subchain. The CPA approach can then be applied on a per-subchain basis, by propagating arrival curves and summing response-time bounds whenever a subchain crosses a reservation boundary, or joins with another chain in a callback with multiple predecessors.

5.1 High-Level Overview

Figure 5 shows an example that illustrates how the proposed analysis can be used to upper-bound the response time of a callback chain spanning multiple reservations. For clarity, interfering callbacks have been omitted in the figure. Response-time bounds for the various subchains of γ^x (i.e., $\gamma^{x,1} = (c_1, c_2)$, $\gamma^{x,2} = (c_3, c_4)$, $\gamma^{x,4} = (c_6, c_7)$, and $\gamma^{x,3} = (c_5)$) can be derived with the results that will be presented in Sections 5.2 and 5.4.

As discussed in Section 2.3, activation curves of non-source subchains must be derived from their predecessors and depend on both the response time of previous subchains and communication delays. In this example, $\eta^a_{x,2}(\Delta) = \eta^e_x(\Delta + R_{x,1} + \delta_{2,3})$, $\eta^a_{x,3}(\Delta) = \eta^a_{x,2}(\Delta + R_{x,2} + \delta_{4,5})$, $\eta^a_{x,4}(\Delta) = \eta^e_{x,3}(\Delta + R_{x,3} + \delta_{5,6})$, where $R_{x,y}$ is a response-time upper bound for $\gamma^{x,y}$. The response time of the chain shown in this example can then be computed as the sum of the response times of the subchains and communications delays, i.e., as $R_x = R_{x,1} + \delta_{2,3} + R_{x,2} + \delta_{4,5} + R_{x,3} + \delta_{5,6} + R_{x,4}$.

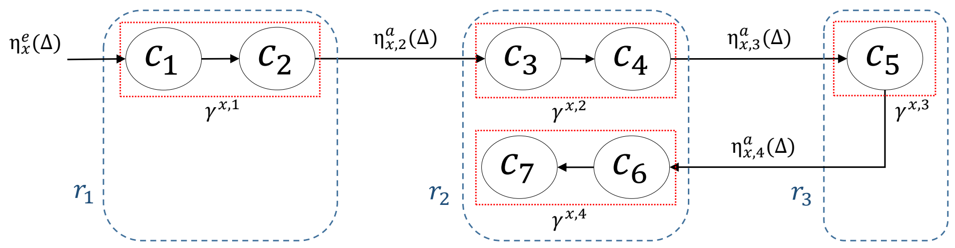


Figure 5 A callback chain crossing multiple reservations. Activations of the first subchain are bounded by an external event arrival curve, while subsequent subchains are characterized by an activation curve that depends on response times and communication delays in the previous subchains.

Processing chains sharing one or more callbacks are also supported by the analysis framework. To deal with this case, the jitter propagation approach is extended to callbacks with multiple incoming edges, i.e., multiple predecessors [29]. As discussed in Section 4, a callback with multiple incoming edges is triggered when it receives a message from *any* of its predecessors. Consequently, the activation curve of a callback (i.e., the source callback of a subchain, when the holistic approach of Section 5.4 is adopted) is derived from the activation curves of the predecessors as follows:

$$\eta^a_i(\Delta) = \sum_{c_j \in pred(c_i)} \eta^a_j(\Delta + R_j + \delta_{j,i}), \tag{1}$$

where R_j is the response time of c_j and $\delta_{j,i}$ is the propagation delay of messages from c_j to c_i. The sum in Equation (1) follows since *each* incoming message spawns a callback instance.

5.2 Analysis for Individual Callbacks

To start, we recall some classic definitions from uniprocessor schedulability analysis. A time t is a *quiet time* with respect to a callback c_i if there is no pending instance of c_i that arrived prior to t. An interval $[t_1, t_2)$ is a *busy period* [59] for c_i iff t_1 and t_2 are quiet times of c_i and there is no quiet time (w.r.t. c_i) in between t_1 and t_2. The *response time* R_i of a callback c_i is defined as the maximum difference, over all possible instances, between the finishing time and the release time of the specific instance. For each callback c_i, the *request-bound function* $rbf_i(\Delta)$ is defined as the maximum amount of (cumulative) processor service required by callback instances released in an interval of length Δ, i.e., $rbf_i(\Delta) = \eta^a_i(\Delta) \cdot e_i$ [31]. Finally, we define the total request-bound function of a given set of callbacks $\mathcal{C}^*$ as $RBF(\mathcal{C}^*, \Delta) = \sum_{c_i \in \mathcal{C}^*} rbf_i(\Delta)$.

From a scheduling perspective, callbacks can be divided into three categories: event sources, timers, and *polling-point-based* (*pp-based*) callbacks. For convenience, in addition to the sets $\mathcal{C}^{\text{evt}}_k$ and $\mathcal{C}^{\text{tmr}}_k$ containing respectively the event sources and timers allocated to r_k, we define also the set $\mathcal{C}^{\text{pp}}_k = \mathcal{C}_k \setminus (\mathcal{C}^{\text{evt}}_k \cup \mathcal{C}^{\text{tmr}}_k)$ of pp-based callbacks allocated to r_k. Event sources are the easiest to analyze since, as described in Section 4, each event source is exclusively allocated to a dedicated reservation. Building on the concept of the supply-bound function $sbf_k(\Delta)$, i.e., the minimum amount of service provided by reservation r_k in an interval of length Δ, Lemma 1 provides a response-time bound for event sources.

▶ **Lemma 1.** *If $A \geq 0$ is the time at which the instance of an event source callback $c_i \in \mathcal{C}^{\text{evt}}_k$ under analysis is released (relative to the beginning of the current busy period), and $R^*_i(A)$ is the least positive value that satisfies*

$$sbf_k(A + R^*_i(A)) = rbf_i(A + 1), \tag{2}$$

then $R_i = \max\{R^*_i(A) \mid A \geq 0\}$ is a response-time bound for c_i.

Proof. By assumption (cf. Section 4), if an event source c_i is allocated to a reservation r_k, no other callbacks are allocated to r_k. Consequently, each callback instance can suffer only self-interference from other instances of the same callback. The lemma follows since the amount of service provided by r_k in the interval $[0, A + R_i^*(A))$ is lower-bounded by $sbf_k(A + R_i^*(A))$ and the maximum amount of service required by instances of c_i released in the interval $[0, A]$ is bounded by $rbf_i(A + 1)$. ◀

Lemma 1 is not directly applicable, as it requires checking an unbounded number of possible release offsets A. To actually implement a response-time analysis, both an upper bound on the length of the analysis interval and a reduction of the number of release offsets that must be checked are needed; we revisit this issue in Section 5.3.

Next, we consider the response times of timers, which are proper callbacks and thus dispatched by ROS executors. As described in Section 3, timer scheduling is not subject to polling points. Nevertheless, since executors process callbacks non-preemptively, timers are subject to lower-priority blocking. Lemma 2 bounds the blocking experienced by a timer callback due to the lower-priority callbacks $c_j \in lp_k(c_i)$.

▶ **Lemma 2.** *A timer callback $c_i \in \mathcal{C}_k$ is blocked for at most $B_i = \max\{e_j \mid c_j \in lp_k(c_i)\}$ time units by lower-priority callbacks.*

Proof. First note that callbacks allocated to any reservation $r_o \neq r_k$ cannot block c_i since there is an independent executor in each reservation and, as explained in Section 3, timers are not subject to polling points. An instance of a timer callback $c_i \in \mathcal{C}_k$ can be released at time $t^* + 1$, where t^* is the time at which a lower-priority callback $c_j \in lp_k(c_i)$ started executing. Due to non-preemptive scheduling, c_i cannot start until c_j completes, i.e., after at most e_j time units. The lemma follows. ◀

With Lemma 2 in place, Lemma 3 upper-bounds the response time of timer callbacks.

▶ **Lemma 3.** *If $A \geq 0$ is the time at which the instance under analysis of a timer callback $c_i \in \mathcal{C}_k^{\text{tmr}}$ is released (relative to the beginning of the current busy period), and $R_i^*(A)$ is the least positive value that satisfies*

$$sbf_k(A + R_i^*(A)) = rbf_i(A + 1) + RBF(hp_k(c_i), A + R_i^*(A) - e_i + 1) + B_i, \tag{3}$$

then $R_i = \max\{R_i^*(A) \mid A \geq 0\}$ is a response-time bound for c_i.

Proof. By Lemma 2, the blocking due to lower-priority callbacks experienced by c_i is bounded by B_i. Due to the priority assignment presented in Section 3, every callback with a priority higher than a timer is itself a timer, i.e., $c_j \in hp_k(c_i) \Rightarrow c_j \in \mathcal{C}_k^{\text{tmr}}$. Due to non-preemptive scheduling, as soon a callback instance starts executing it cannot be interfered with by any other callback, i.e., higher-priority callbacks can interfere only in the interval $[0, A+R_i^*(A)-e_i]$. The lemma follows by noting that: **(i)** the interference from higher-priority callbacks is bounded by the total request-bound function, i.e., $RBF(hp_k(c_i), A+R_i^*(A)-e_i+1)$, **(ii)** the callback under analysis can suffer self-interference only from instances released in $[0, A]$, and **(iii)** the amount of service provided by r_k in the interval $[0, A + R_i^*(A))$ is lower-bounded by $sbf_k(A + R_i^*(A))$, i.e., the callback under analysis completes no later than when the guaranteed minimum service matches the maximum total demand. ◀

Again, we discuss how to use Lemma 3 in a practical response-time analysis in Section 5.3. Next, we consider pp-based callbacks. Due to the unpredictable nature of dynamic polling points, pp-based callbacks suffer additional blocking. Indeed, when an instance of a pp-based

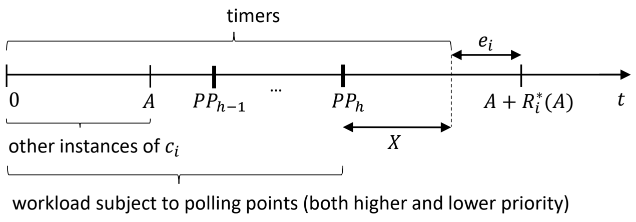


Figure 6 Time intervals in which other pp-based callbacks and timers can interfere with a pp-based callback c_i under analysis.

callback is released, it requires the completion of one or more processing windows before being executed. A response-time bound for pp-based callbacks is provided by Lemma 4, which is illustrated in Figure 6.

▶ **Lemma 4.** *If $A \geq 0$ is the time at which the instance of a pp-based callback $c_i \in \mathcal{C}_k^{\text{pp}}$ under analysis is released (relative to the beginning of the current busy period), $X \geq 0$ is the difference between time $A + R_i^*(A) - e_i$ and the last polling point before time $A + R_i^*(A) - e_i$ (see Figure 6), and $R_i^*(A)$ is the least positive value that satisfies*

$$\begin{aligned} \mathit{sbf}_k(A + R_i^*(A)) = \mathit{rbf}_i(A+1) &+ \mathit{RBF}(\mathcal{C}_k^{\text{oth}}, A + R_i^*(A) - e_i - X + 1) \\ &+ \mathit{RBF}(\mathcal{C}_k^{\text{tmr}}, A + R_i^*(A) - e_i + 1), \end{aligned} \tag{4}$$

where $\mathcal{C}_k^{\text{oth}} = \mathcal{C}_k \setminus (\mathcal{C}_k^{\text{tmr}} \cup \{c_i\})$ is the set of the other non-timer callbacks allocated to r_k, then $R_i = \max\{R_i^*(A) \mid A \geq 0\}$ is a response-time bound for c_i.

Proof. Due to polling points, non-timer callbacks (both of higher and lower priority) can delay the callback under analysis only with instances that have arrived by the last polling point. Note that a polling point cannot occur while a callback is executing. Consequently, the polling point at time $A + R_i^*(A) - e_i - X$ is the last polling point before $A + R_i^*(A)$, and pp-based callbacks can delay the callback instance under analysis only with instances released in $[0, A + R_i^*(A) - e_i - X)$ (note that a callback released exactly at a polling point PP_n is processed during PW_n). Due to the priority assignment discussed in Section 3, each timer callback has higher priority than any pp-based callback. It follows that, due to non-preemptive scheduling, all timer callbacks $\mathcal{C}_k^{\text{tmr}}$ can interfere with the pp-based callback under analysis up to the time at which it starts executing, i.e., at any time in $[0, A + R_i^*(A) - e_i]$. The lemma then follows analogously to Lemma 3 by noting that: **(i)** the callback under analysis can suffer self-interference only from instances released in $[0, A]$, and **(ii)** the amount of service provided by r_k in the interval $[0, A + R_i^*(A))$ is lower-bounded by $\mathit{sbf}_k(A + R_i^*(A))$. ◀

Lemma 4 upper-bounds the response time experienced by a pp-based callback. As for the previous lemmas, we will discuss how to bound the space of possible times A in Section 5.3. Moreover, Lemma 4 depends on the time distance X between $A + R_i^*(A) - e_i$ and the last polling point before $A + R_i^*(A)$, which is generally unknown during offline analysis. Consequently, we need to determine the scenario (i.e., the value of X) that maximizes the response time. Intuitively, this case occurs when the callback c_i under analysis starts executing just after the last polling point, i.e., lower-priority callbacks can interfere with c_i throughout the time from its release until it starts executing. In this case, $X = 0$. Lemma 5 proves that $X = 0$ indeed dominates all possible values of X.

▶ **Lemma 5.** *The delay experienced by a pp-based callback $c_i \in \mathcal{C}_k \setminus (\mathcal{C}_k^{\text{tmr}} \cup \mathcal{C}_k^{\text{evt}})$ due to other pp-based callbacks is maximized when c_i starts executing just after the last polling point:*

$$\max_{A\geq 0, X\geq 0} RBF(\mathcal{C}_k^{\text{oth}}, A + R_i^*(A) - e_i - X + 1) = \max_{A\geq 0} RBF(\mathcal{C}_k^{\text{oth}}, A + R_i^*(A) - e_i + 1), \quad (5)$$

where $R_i^*(A)$, A, and X are defined as in Lemma 5.

Proof. The lemma follows by noting that $X \geq 0$ and that $RBF(\mathcal{C}_k^{\text{oth}}, A+R_i^*(A)-e_i-X+1)$ is a sum of monotonic non-decreasing functions; hence it is monotonic non-decreasing, too. ◀

By Lemma 5, it follows that the amount of interference generated by timer callbacks $c_t \in \mathcal{C}_k^{\text{tmr}}$ and non-timer callbacks $c_n \in \mathcal{C}_k^{\text{oth}}$ is the same in the worst case. Consequently, we can merge the two sets, and rewrite Equation (4) in a simpler manner:

$$sbf_k(A + R_i^*(A)) = rbf_i(A + 1) + RBF(\{\mathcal{C}_k \setminus c_i\}, A + R_i^*(A) - e_i + 1). \quad (6)$$

Equation (6) highlights that the scheduling policy adopted by the built-in ROS executor allows every other callback, independent of priority, to interfere with pp-based callbacks. Consequently, polling points make the priority assignment ineffective for upper-bounding the response time of pp-based callbacks. This confirms what we empirically observed during the model validation (Section 3) from an analytical perspective. Note that timer callbacks are not affected by polling points and their response-time bound is equivalent to non-preemptive fixed-priority scheduling [17], in the context of a resource reservation (Lemma 3).

5.3 Bounding the Search Space

The lemmas presented in Section 5.2 require checking Equations (3), (4) and (5) for all possible $A \geq 0$, where A represents the relative release time (with respect to the beginning of the current busy period) of the callback instance under analysis. To use the previous lemmas in a practical response-time analysis, both a bound on the analysis interval and a reduction of the search space size are required. Note that the analysis interval can be bounded by the longest interval during which a reservation r_k is busy serving higher-or-equal-priority workload, i.e., the length of the longest busy period [59], which Lemma 6 bounds.

▶ **Lemma 6.** *Let $\mathcal{C}_k^{\text{evt}}$, $\mathcal{C}_k^{\text{tmr}}$, and $\mathcal{C}_k^{\text{pp}}$ be the sets of all event source, timer, and pp-based callbacks allocated to r_k, respectively. If $c_i \in \mathcal{C}_k$ is the callback under analysis, and L^* is the least positive value that satisfies*

$$sbf_k(L^*) = \begin{cases} rbf_i(L^*) & \text{if } c_i \in \mathcal{C}_k^{\text{evt}} \\ RBF(hp_k(c_i), L^*) + B_i + rbf_i(L^*) & \text{if } c_i \in \mathcal{C}_k^{\text{tmr}} \\ RBF(\mathcal{C}_k, L^*) & \text{if } c_i \in \mathcal{C}_k^{\text{pp}}, \end{cases} \quad (7)$$

then L^* is an upper bound on the length of the longest busy period.

Proof. By contradiction, assume that there exist a busy period of c_i with length $L' > L^*$. Under this assumption, in the busy period corresponding to L' either **(i)** there are more callbacks delaying c_i, or **(ii)** callbacks execute for more time or, **(iii)** callbacks arrive more frequently than in the busy period corresponding to L^*. By Lemmas 1, 3, and 4, Equation (7) accounts for all callbacks that can delay c_i. Moreover, by definition of the request-bound function, Equation (7) is composed of a sum of products of worst-case execution times and arrival curves. By definition, no callback can execute for more than its worst-case execution time. Further, the activation curve $\eta^a(\Delta)$ defines an upper bound on the number of events that can arrive in any time window $[t, t+\Delta)$, thus leading to a contradiction. ◀

With Lemma 6 restricting the search to a finite interval, Lemma 7 below reduces the number of points contained in the search space. To this end, consider the response-time bounds computed with Equations (3), (4) and (6): each can be expressed as an instance of a *general response-time equation* $sbf_k(A+x) = rbf_i(A+1) + I(A+x) + B$, where B is a constant and the function I depends only on its argument. Equation (6), for example, can be written in this form by substituting $B = 0$ and $I(\Delta) = RBF(\{\mathcal{C}_k \setminus c_i\}, \Delta - e_i + 1)$. For any A, we let $SOL(A)$ denote the set of all positive x that satisfy the general response-time equation.

▶ **Lemma 7.** *For a callback $c_i \in \mathcal{C}_k$ under analysis, let $\mathcal{A}_i^- = \{A > 0 \mid rbf_i(A+1) = rbf_i(A)\}$ denote the points where $rbf_i(A)$ stays constant. For any $a \in \mathcal{A}_i^-$, $R_i^*(a) \neq \max_{A \geq 0} R_i^*(A)$.*

Proof. We prove that $R_i^*(a)$ is strictly less than its "neighbor" $R_i^*(a-1) \in SOL(a-1)$, and hence necessarily also less than $\max_{A\geq 0} R_i^*(A)$. To this end, we establish that $R_i^*(a-1) = R_i^*(a) + 1$ (which is well-defined since $0 \notin \mathcal{A}_i^-$) by showing that **(i)** $R_i^*(a) + 1 \in SOL(a-1)$ and **(ii)** $R_i^*(a) + 1 \leq a'$ for any $a' \in SOL(a-1)$.

Step (i): By definition, $R_i^*(a) \in SOL(a)$, and thus $sbf_k(A + R_i^*(A)) = rbf_i(A+1) + I(A + R_i^*(A)) + B$. By adding $0 = 1-1$ and using the fact that $a \in \mathcal{A}_i^-$ and hence $rbf_i(a+1) = rbf_i(a)$, we equivalently obtain $sbf_k(a - 1 + R_i^*(a) + 1) = rbf_i(a-1+1) + I(a-1+R_i^*(a)+1) + B$. This is the definition of $SOL(a-1)$ and hence proves $R_i^*(a) + 1 \in SOL(a-1)$.

Step (ii): Consider any $a' \in SOL(a-1)$. Then $sbf_k((a-1)+a') = rbf_k(a) + I((a-1)+a') + B$. Using again that $rbf_i(a+1) = rbf_i(a)$, this is equivalent to $sbf_k(a+(a'-1)) = rbf_k(a+1) + I(a+(a'-1)) + B$, which matches the definition of $SOL(a)$ and hence $a'-1 \in SOL(a)$. By definition, $R_i^*(a) = \min\{x \mid x \in SOL(a)\}$, and thus we have $a' - 1 \geq R_i^*(a) \Leftrightarrow R_i^*(a) + 1 \leq a'$. ◀

Together, Lemmas 6 and 7 enable an efficient implementation of the response-time analysis by restricting the required search space $\mathcal{A}_i$ (w.r.t. a callback c_i) to

$$\mathcal{A}_i = \{A \mid 0 \leq A \leq L^*\} \setminus \mathcal{A}_i^- = \{0 \leq A \leq L^* \mid rbf_i(A+1) \neq rbf_i(A)\} \cup \{0\}.$$

To further reduce the effects of arrival bursts, we next provide a joint response-time bound for a sequence of callbacks in a single reservation.

5.4 Analysis for Processing Chains

This section is provides an end-to-end analysis for linear subchains composed of multiple callbacks, where each subchain does not cross reservation boundaries. To this end, we extend the notion of request bound functions to subchains as $rbf^{x,y}(\Delta) = \eta_s^a(\Delta) \cdot e^{x,y}$, where c_s is the first callback of the subchain $\gamma^{x,y}$, and $e^{x,y} = \sum_{c_i \in \gamma^{x,y}} e_i$ is the cumulative worst-case execution time of the subchain. Consequently, $RBF^\gamma(\Gamma_k, \Delta) = \sum_{\forall \gamma^{x,y} \in \Gamma_k} rbf^{x,y}(\Delta)$, where Γ_k is the set of subchains allocated to r_k. Lemma 8 allows us to compute a response-time bound for a subchain composed of multiple callbacks (if a subchain consists of only a single callback, its response time can be computed with the results of Section 5.2).

▶ **Lemma 8.** *If $\gamma^{x,y} = (c_s, \ldots, c_e)$ is a subchain composed of $|\gamma^{x,y}| \geq 2$ callbacks, Γ_k is the set of subchains allocated to r_k, and $R_{x,y}$ is the least positive value that satisfies*

$$sbf_k(R_{x,y}) = RBF^\gamma(\Gamma_k, R_{x,y} - e_e + 1), \tag{8}$$

then $R_{x,y}$ is a response-time bound for $\gamma^{x,y}$.

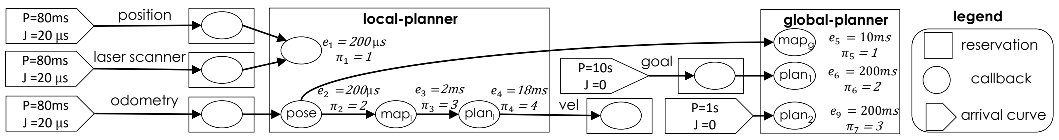


Figure 7 Callback graph of the event-driven `move_base` system.

Proof. Since $|\gamma^{x,y}| \geq 2$, the last callback in a subchain is a pp-based callback (timers and event source are necessarily source callbacks). By Lemmas 4, 5, and Equation (6), every other callback can interfere with a pp-based callback. Since each instance of a subchain completes when the last callback of the chain terminates, it follows that the chain under analysis can be interfered with by all other chains, regardless of callback priorities. The lemma follows by noting that, due to non-preemptive scheduling, the subchain cannot be interfered with during the execution of its last callback. ◀

Lemma 8 extends Lemma 4 for subchains. As observed in Section 5.2, also in this case the presence of pp-based callbacks make the priority assignment ineffective for the purpose of computing a tighter response-time bound. However, since arrival bursts of interfering callbacks are accounted only for once per subchain, analyzing a subchain holistically still overall improves the analysis accuracy for long subchains.

5.5 Analysis Summary

The results presented in this section allows analyzing ROS systems under reservation-based scheduling. Specifically, Section 5.2 proposed a response-time analysis for single callbacks, and Section 5.4 extended it to subchains allocated to a single reservation. As discussed in Section 5.1, both approaches allow to compute a safe end-to-end latency for generic processing chains by propagating arrival curves and summing individual response-time bounds. Specifically, the effects of predecessor callbacks are accounted for as release jitter in the activation curves of non-source callbacks. Such release jitter depends on the response times of predecessors, but also response times depend on jitter in a circular manner. As in the CPA approach, this problem can be solved by iteratively searching for a global fixed point at which all jitter terms and response times are consistent.

6 Case Study

Our analysis seeks to enable ROS developers to easily and quickly try different designs and explore various what-if scenarios. To evaluate the suitability of our approach for that purpose, we analyzed a safety-critical processing chain in the popular `move_base` package, the central part of the ROS navigation stack for wheeled robots, using sensor rates and (observed) maximum execution times from a Bosch-internal case study. Since `move_base` has not been ported to ROS 2 yet, we model the ROS 1 version as if it ran on a ROS 2 system.

The `move_base` package addresses the *path planning* problem: given a map of the environment, first find a path to the goal location (*global planning*), and then control the robot's velocity to follow that path while avoiding obstacles (*local planning*). Both planners base their decision on internal maps, which reflect the component's knowledge of obstacles and properties of the environment. As the robot moves through the environment, these maps are continously updated based on the most recent sensor data.

The `move_base` callback graph is illustrated in Figure 7. The incoming sensor and position data is normalized to absolute coordinates based on the robot's *pose* and then integrated into the respective maps. The local planner then updates its plan based on the new information.

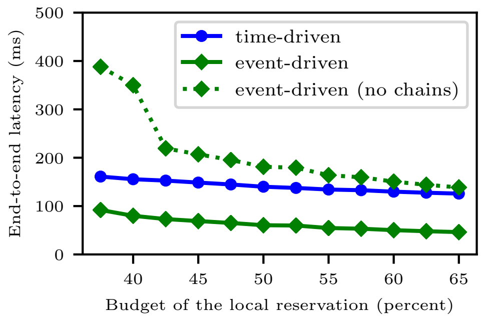
From a timing perspective, the execution time of the global planner stands out. Unlike the local planner, the global planner is difficult to predict and its execution time depends heavily on the path-finding difficulty. The global planner's map is also significantly larger and often updated only partially, further reducing predictability. In the Bosch case study, global planning times reached up to 200 ms, more than ten times the local planner's runtime. Fortunately, the global planning is not time-critical. At worst, computing the global plan too late may cause the robot to take a detour for a few moments. We therefore separate the global planning callbacks from the time-critical local planning callbacks using two reservations. This not only isolates the more unpredictable components from the critical path, but also helps to limit the effects of the ROS executor scheduling policy.

Internally, the `move_base` subsystem is completely time-driven, using ROS topics only to communicate with other components. We configured the local planner to run in sync with the fixed sensor rate of 12.5 Hz, and the global planner to run more rarely at 1 Hz. While this setup makes for a quite predictable system, it is also very inflexible and makes it difficult to ensure that components do not consume stale data. For comparison, we therefore modeled two variants: the original, time-driven version, and an event-driven alternative design that models the activation dependency explicitly using internal topics.

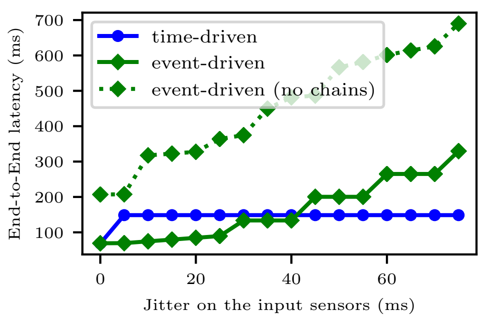
In our case study, we are interested in the end-to-end latency of the path from the odometry input to the wheel command output (denoted "vel" for velocity). This latency determines, for example, how long the robot takes to react to an obstacle suddenly appearing in front of it, and hence is safety-relevant. In the event-driven setup, this is defined as the worst-case response time from activation to completion of the chain. For the time-driven setup, we compute the response time of the local planner and, if there is jitter on the sensor inputs, add the activation period as worst-case sampling delay. Although generally speaking the response time of the last chain element is not necessarily identical to the chain's overall latency, it happens to coincide here: by prioritizing the chain in decreasing order, and triggering all tasks at the same time, no task can run before its predecessor completes. The completion of the local planner thus implies the completion of the entire processing chain.

One of the most difficult steps in using reservation-based scheduling is dimensioning the reservations correctly. When isolating the local from the global planner, one would like to give the local planner just enough budget to complete in time, leaving as much execution time as possible for the global planner. To this end, we prototyped our analysis in the pyCPA framework [18]. Figure 8a shows the results for both the time-driven setup and the event-driven setup. In case of the event-driven setup, we also included the analysis results when disabling the whole-chain analysis described in Section 5.4. The graph shows the entire range of local planner budgets in percent of the total core bandwidth. For simplicity, we allocate the rest of the core to the global planner reservation. Since we do not analyze any processing chains through the global planner's reservation, though, the exact amount of bandwidth dedicated to this reservation does not impact the reported response times.

The graph clearly shows a similar effect of budgeting on the time-driven and event-driven system. However, due to the worst-case sampling delay, the time-driven system remains one sampling period of 80 ms above the event-driven latencies. Clearly, the event-driven approach is advisable in this setup, allowing the system to wait until the sensor results arrive instead of commencing planning based on stale values. One can also observe the beneficial effects of the whole-chain analysis; when disabled, the chain's self-interference significantly inflates the predicted response-time bounds. Since the analysis conservatively assumes that every callback is blocked by every other callback, actual interference is overcounted four-fold.



(a) Effects of reservation dimensioning on the critical path latency.



(b) Effects of input jitter on the critical path latency, using a budget of 45%.

Figure 8 Experimental results.

Another important property of any control system is how it copes with input jitter. While the previous experiment modeled the sensors as strictly following the 12.5 Hz schedule with only 200 μs of jitter, Figure 8b shows the predicted end-to-end latency as input jitter increases, using a local planning budget of 45%. Here, one clearly observes the main benefits of purely time-driven systems; they are very robust to input jitter, mainly because they are not influenced by bursts. For the event-driven system, one can observe a significant rise roughly every 20 ms. These are the points at which one more event can arrive during the execution of the processing chain. The event-driven system remains superior below 40 ms of jitter (i.e., at most one event at the same time), but succumbs to self-interference at larger jitters. Without a systematic analysis, such tradeoffs are extremely difficult to anticipate.

In conclusion, this case study highlights the benefits of automated response-time analysis. Without implementing a single line of ROS code, we are able to reason about the worst-case latencies of two quite different `move_base` designs, noting the advantages and disadvantages in different scenarios. Having a fully-integrated and automatic version of this analysis would clearly be a major aid to ROS developers, allowing response times to be treated as a measurable design constraint instead of relying solely on intuition, trial-and-error, or post-implementation experimentation. Extrapolating a bit further, it might even allow one day to reason about the latency of external dependencies, enabling the safe and easy reuse of well-tested components for time-critical purposes.

7 Related Work

The literature concerning the real-time aspects of ROS 2 processing chains is quite limited. To the best of our knowledge, this is the first paper modeling a ROS system from a real-time perspective and proposing a response-time analysis for ROS processing chains.

Most of the existing work on ROS targets ROS 1 systems, mainly conducting empirical performance measurement and proposing possible improvements. For instance, Saito et al. [53] proposed a priority-based message transmission algorithm for ROS 1 that allows publishers to send data to multiple subscribers according to their priorities, and a mechanism for synchronizing communications among multiple nodes running at different frequencies. Suzuki et al. [61] presented a mechanism for coordinating CPU and GPU execution of ROS 1 nodes, and an offline scheduling algorithm that assigns priorities to nodes according to their laxities. Maruyama et al. [36] conducted an experimental study aimed at comparing the performance of ROS 1 and a preliminary version of ROS 2 under different DDS implementations. Gutiérrez et al. [25] performed a similar evaluation of ROS 2 "Ardent Apalone" under Linux with the PREEMPT-RT patch. Concerning more general robotic systems, Lotz et al. [34, 35] presented a meta-model for designing non-functional aspects of robotics systems such that the resulting models can be analyzed with the SymTA/S timing analysis tool [27], which is based on the CPA approach [29, 46, 48, 49, 65].

Concerning the analysis of processing chains in distributed systems, one of the first proposals to verify end-to-end timing constraints is due to Fohler and Ramamritham [20], who proposed an approach for obtaining a static schedule composed of tasks with precedence constraints. In the context of non-static scheduling, prior work can be divided into two main categories: those based on CPA [27] and those adopting an holistic approach [42, 63]. The first method adopts arbitrary arrival curves [47] and analyzes chains crossing different nodes of a distributed system individually (by means of a *local component analysis*), and propagates the event model (i.e., activation curves) until convergence is achieved [50]. Different local analyses have been designed over years. For instance, Schlatow and Ernst [54, 55] proposed a local analysis for chains entirely contained in a single resource (e.g., a processing node) where tasks along the chain can have arbitrary priorities, under preemptive scheduling. Other authors [26, 28, 52, 56, 64] improved the analysis precision by accounting for correlations among events in different components. Previously, Thiele et al. [62] proposed Real-Time Calculus, an approach similar to CPA in which the service demand of the workload is modeled with arrival curves, and service curves model the processing capacity of local components. As in Network Calculus [30], arrival curves and services curve are combined together by means of a max-plus algebra, thereby obtaining the timing behavior of the component. Concerning the holistic approach, the seminal work is due to Tindell and Clark [63], who proposed a schedulability analysis for transactions, i.e., sporadically triggered sequences of events, scheduled under fixed-priority preemptive scheduling. Their analysis has been refined by Palencia et al. considering offsets [42] and precedence relations [41]. More details about the transactional task model can be found in a survey by Rahni et al. [44].

Only little attention has been given to date on how specific frameworks affect worst-case response times. To the best of our knowledge, all of them target the OpenMP framework [40], which is usually used for globally scheduled parallel tasks. For example, Serrano et al. [57] distinguished between tied and untied sub-tasks in OpenMP, proposing a response-time analysis for a parallel task composed of untied sub-tasks. While untied nodes have no particular scheduling restrictions, tied sub-tasks are OpenMP-specific and consist of a subgraph whose nodes must all execute on a single thread. Subsequently, Sun et al. [60] proposed an improvement of the OpenMP scheduling policy. To the best of our knowledge, the present paper is the first to systematically study the temporal behavior of ROS.

8 Limitations, Extensions, and Conclusions

This initial work on the timing analysis of ROS 2 processing chains can already handle practical components (such as `move_base`), and provides a rich foundation for future developments. Nonetheless, given the inevitable complexities associated with a mature, flexible, and widely used framework, we had to elide certain infrequently used aspects of ROS. In the following, we discuss these limitations and highlight promising direction for future extensions.

This paper considers the built-in single-threaded ROS executor. ROS also provides a multi-threaded variant of that executor, and additionally allows the definition of arbitrary special-purpose executors. Being able to easily integrate special-purpose schedulers tailored to specific robot needs would allow for interesting domain-specific research in the future.

When using multiple executor threads in a shared process, concurrency problems arise. ROS introduces *mutually-exclusive callback groups* to address this problem, and guarantees that callbacks in the same group are never executed concurrently. Extending our analysis to handle blocking relationships among callbacks remains future work.

This paper assumed the graph of the callbacks to be fixed. However, ROS allows nodes to dynamically join and leave, as well as to subscribe to and unsubscribe from topics dynamically at runtime, which is particularly useful for implementing different operating modes. This problem is referred to as *mode changes* in the literature [38, 45]. Our analysis can be applied to each mode in stable operation, but not does account for transient effects during mode changes. The design of new analysis techniques accounting for mode changes (e.g., extending [10, 11, 13] to ROS systems) represents another relevant future direction.

We modeled the overhead of network delays and the underlying DDS implementation as a single variable $\delta_{i,j}$, which allows for a safe and simple accounting for network-related delays in the overall response time by summing the communication delay every time the network is crossed. An opportunity for future improvements would be to integrate network analysis to eliminate the pessimism induced by the pay-burst-only-once problem when the network is crossed multiple times. Furthermore, a detailed study of available DDS implementations would allow for a more precise modeling of message processing overheads.

In addition to topics and services, ROS also provides a *waitable* callback type. This type is intended to implement more complex communication primitives like the long-running and high-level *actions* known from ROS 1 [1]. Since this mechanism was only introduced in the latest release, there are no known users of this mechanism as of now. It will be necessary to extend our analysis to these additional methods as and when they are adopted in ROS 2.

We assumed each callback to trigger an activation of all its successors at most once per execution. As a future improvement, we would like to extend the proposed analysis to allow a callback to trigger its successors only after having executed a predefined number of instances, or to trigger multiple instances of each successors in a single execution [23].

Our analysis based on the CPA approach allows to simply and efficiently analyze a real-world system, limiting the complexity by considering reservations individually. The analysis accuracy can be further improved by considering correlations among activation events in the chain, thus reducing the "pay-burst-only-once" problem also for chains spanning multiple reservations. A possible research direction for future work consists in extending the approaches presented by Fonseca et al. [21] and Casini et al. [14] (in the context of preemptive and non-preemptive fixed-priority scheduling of parallel tasks, respectively), based on which chains crossing multiple reservations could be modeled by means of self-suspending tasks [16]. In this way, arrival bursts can be considered only once per reservation, thus improving the analysis precision for chains crossing the same reservation multiple times.

To conclude, we have presented the first comprehensive scheduling model of ROS 2 systems, based on a review of its source code and documentation. We derived a response-time analysis for processing chains that takes the specific properties of the ROS framework into account and applied to a realistic case study. While there remain ample opportunities for future extensions, our contributions represent the first steps towards an automated analysis tool that could allow ROS users without expert knowledge in real-time systems to quickly and conveniently determine temporal safety and latency properties of their applications.

References

1 Action Lib. URL: `http://wiki.ros.org/actionlib`.

2 eProsima FastRTPS. URL: `https://www.eprosima.com/index.php/products-all/eprosima-fast-rtps`.

3 The move_base package. URL: http://wiki.ros.org/move_base.

4 Robots using ROS. URL: http://robots.ros.org.

5 ROS 2 "Crystal Clemmys". URL: http://www.ros.org/news/2018/12/ros-2-crystal-clemmys-released.html.

6 RTI Connext DDS. URL: https://www.rti.com/products/connext-dds-professional.

7 Vortex OpenSplice. URL: http://www.prismtech.com/vortex/vortex-opensplice.

8 L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS 1998)*, Madrid, Spain, december 2-4, 1998.

9 A. Biondi, G. C. Buttazzo, and M. Bertogna. Schedulability Analysis of Hierarchical Real-Time Systems under Shared Resources. *IEEE Transactions on Computers*, 65, May 2016.

10 A. Block and J. H. Anderson. Accuracy versus migration overhead in real-time multiprocessor reweighting algorithms. In *12th International Conference on Parallel and Distributed Systems - (ICPADS'06)*, Minneapolis, USA, july, 12-15, 2006.

11 A. Block, J. H. Anderson, and G. Bishop. Fine-grained task reweighting on multiprocessors. In *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*, Hong Kong, China, july, 17-19, 2005.

12 G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Third Edition.* Springer, 2011.

13 D. Casini, A. Biondi, and G. C. Buttazzo. Handling Transients of Dynamic Real-Time Workload Under EDF Scheduling. *IEEE Transactions on Computers*, 2018.

14 D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo. Partitioned Fixed-Priority Scheduling of Parallel Tasks Without Preemptions. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, December 2018.

15 Daniel Casini, Luca Abeni, Alessandro Biondi, Tommaso Cucinotta, and Giorgio Buttazzo. Constant Bandwidth Servers with Constrained Deadlines. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, RTNS '17, 2017.

16 Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil Audsley, Raj Rajkumar, Dionisio de Niz, and Georg von der Brüggen. Many suspensions, many problems: a review of self-suspending tasks in real-time systems. *Real-Time Systems*, September 2018.

17 Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, April 2007.

18 Jonas Diemer, Philip Axer, and Rolf Ernst. Compositional performance analysis in python with pyCPA. In *In Proceedings of WATERS'12*, 2012.

19 Jonas Diemer, Jonas Rox, and Rolf Ernst. Modeling of Ethernet AVB Networks for Worst-Case Timing Analysis. *IFAC Proceedings Volumes*, 2012. 7th Vienna International Conference on Mathematical Modelling.

20 G. Fohler and K. Ramamritham. Static scheduling of pipelined periodic tasks in distributed real-time systems. In *Proceedings 9th Euromicro Workshop on Real Time Systems*, June 1997.

21 J. Fonseca, G. Nelissen, V. Nelis, and L. M. Pinho. Response time analysis of sporadic DAG tasks under partitioned scheduling. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, May 2016.

22 T. Foote. ROS community metrics report. URL: http://download.ros.org/downloads/metrics/metrics-report-2018-07.pdf.

23 J. J. G. Garcia, J. C. P. Gutierrez, and M. G. Harbour. Schedulability analysis of distributed hard real-time systems with multiple-event synchronization. In *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000*, 2000.

24 B. Gerkey. Why ROS 2.0? URL: http://design.ros2.org/articles/why_ros2.html.

25 C. Gutierrez, L. Juan, I. Uguarte, and V. Vilches. Towards a distributed and real-time framework for robotsç Evaluation of ROS 2.0 communications for real-time robotics applications. Techical report, Erle Robotics S.L., 2018.

26 R. Henia and R. Ernst. Context-aware scheduling analysis of distributed systems with tree-shaped task-dependencies. In *Design, Automation and Test in Europe*, March 2005.

27 R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the SymTA/S approach. *IEEE Proceedings - Computers and Digital Techniques*, March 2005.

28 K. Huang, L. Thiele, T. Stefanov, and E. Deprettere. Performance Analysis of Multimedia Applications using Correlated Streams. In *2007 Design, Automation Test in Europe Conference Exhibition*, 2007.

29 M. Jersak. *Compositional Performance Analysis for Complex Embedded Applications.*

30 Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet.* Springer-Verlag, Berlin, Heidelberg, 2001.

31 J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *[1989] Proceedings. Real-Time Systems Symposium*, 1989.

32 J. Lelli, C. Scordino, L. Abeni, and D. Faggioli. Deadline scheduling in the Linux kernel. *Software: Practice and Experience*, 46(6):821–839, 2016.

33 G. Lipari and E. Bini. Resource partitioning among real-time applications. In *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings.*, pages 151–158, July 2003.

34 A. Lotz, A. Hamann, R. Lange, C. Heinzemann, J. Staschulat, V. Kesel, D. Stampfer, M. Lutz, and C. Schlegel. Combining robotics component-based model-driven development with a model-based performance analysis. In *2016 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, December 2016.

35 Alex Lotz, Arne Hamann, Ingo Lütkebohle, Dennis Stampfer, Matthias Lutz, and Christian Schlegel. Modeling Non-Functional Application Domain Constraints for Component-Based Robotics Software Systems. In *6th International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob'15)*, 2015.

36 Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ROS2. In *Proceedings of the 13th International Conference on Embedded Software*, page 5. ACM, 2016.

37 L. Marzario, G. Lipari, P. Balbastre, and A. Crespo. IRIS: a new reclaiming algorithm for server-based real-time systems. In *Proceedings. RTAS 2004. 10th IEEE Real-Time and Embedded Technology and Applications Symposium, 2004.*, May 2004.

38 M. Negrean, M. Neukirchner, S. Stein, S. Schliecker, and R. Ernst. Bounding mode change transition latencies for multi-mode real-time distributed applications. In *ETFA2011*, 2011.

39 Object Management Group. *Data Distribution Service (DDS)*, 1.4 edition, 2015.

40 OpenMP. *OpenMP Application Program Interface, Version 4.0.*, 2013.

41 J. C. Palencia and M. G. Harbour. Exploiting precedence relations in the schedulability analysis of distributed real-time systems. In *Proceedings 20th IEEE Real-Time Systems Symposium*, 1999.

42 J. C. Palencia and M. Gonzalez Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings 19th IEEE Real-Time Systems Symposium*, 1998.

43 Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

44 Ahmed Rahni, Emmanuel Grolleau, Michaël Richard, and Pascal Richard. Feasibility Analysis of Real-time Transactions. *Real-Time Syst.*, 48(3), 2012.

45 J. Real and A. Crespo. Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal. *Real-Time Systems*, 26(2):161–197, March 2004.

46 K. Richter. *Compositional Scheduling Analysis Using Standard Event Models: The SymTA/S Approach.*

47 K. Richter and R. Ernst. Event model interfaces for heterogeneous system analysis. In *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, pages 506–513, 2002.

48 K. Richter, M. Jersak, and R. Ernst. A Formal Approach to MpSoC Performance Verification. *Computer*, 36(4), 2003.

49 K. Richter, R. Racu, and R. Ernst. Scheduling Analysis Integration for Heterogeneous Multiprocessor SoC. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS)*, 2003.

50 K. Richter, D. Ziegenbein, M. Jersak, and R. Ernst. Model composition for scheduling analysis in platform design. In *Proceedings 2002 Design Automation Conference*, June 2002.

51 Jonas Rox and Rolf Ernst. Formal Timing Analysis of Full Duplex Switched Based Ethernet Network Architectures. In *SAE Technical Paper*, 2010.

52 Jonas Rox and Rolf Ernst. Compositional Performance Analysis with Improved Analysis Techniques for Obtaining Viable End-to-end Latencies in Distributed Embedded Systems. *Int. J. Softw. Tools Technol. Transf.*, 15(3), 2013.

53 Yukihiro Saito, Takuya Azumi, Shinpei Kato, and Nobuhiko Nishio. Priority and synchronization support for ROS. In *2016 IEEE 4th International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pages 77–82. IEEE, 2016.

54 J. Schlatow and R. Ernst. Response-Time Analysis for Task Chains in Communicating Threads. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016.

55 Johannes Schlatow and Rolf Ernst. Response-Time Analysis for Task Chains with Complex Precedence and Blocking Relations. *ACM Trans. Embed. Comput. Syst.*, 16(5s), September 2017.

56 Simon Schliecker and Rolf Ernst. A Recursive Approach to End-to-end Path Latency Computation in Heterogeneous Multiprocessor Systems. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '09, 2009.

57 M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quiñones. Timing characterization of OpenMP4 tasking model. In *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2015.

58 Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *24th IEEE Real-Time Systems Symposium*, 2003.

59 Marco Spuri. Analysis of Deadline Scheduled Real-Time Systems. Technical report, RR-2772, INRIA, 1996.

60 J. Sun, N. Guan, Y. Wang, Q. He, and W. Yi. Real-Time Scheduling and Analysis of OpenMP Task Systems with Tied Tasks. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, December 2017.

61 Yuhei Suzuki, Takuya Azumi, Shinpei Kato, and Nobuhiko Nishio. Real-Time ROS extension on transparent CPU/GPU coordination mechanism. In *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, pages 184–192. IEEE, 2018.

62 L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *2000 IEEE International Symposium on Circuits and Systems. Emerging Technologies for the 21st Century. Proceedings*, May 2000.

63 Ken Tindell and John Clark. Holistic Schedulability Analysis for Distributed Hard Real-time Systems. *Microprocess. Microprogram.*, 40(2-3):117–134, 2012.

64 Ernesto Wandeler and Lothar Thiele. Workload Correlations in Multi-processor Hard Real-time Systems. *J. Comput. Syst. Sci.*, 73(2), 2007.

65 D. Ziegenbein, M. Jersak, K. Richter, and R. Ernst. Breaking Down Complexity for Reliable System-Level Timing Validation. In *Ninth IEEE/DATC Electronic Design Processes Workshop (EDP'02)*, 2002.

Implementation of Memory Centric Scheduling for COTS Multi-Core Real-Time Systems

Juan M. Rivas[1]
PARTS Research Centre, Université libre de Bruxelles, Brussels, Belgium
jrivasco@ulb.ac.be

Joël Goossens
PARTS Research Centre, Université libre de Bruxelles, Brussels, Belgium
joel.goossens@ulb.ac.be

Xavier Poczekajlo
PARTS Research Centre, Université libre de Bruxelles, Brussels, Belgium
xavier.poczekajlo@ulb.ac.be

Antonio Paolillo
HIPPEROS S.A., Louvain-la-Neuve, Belgium
antonio.paolillo@hipperos.com

Abstract

The demands for high performance computing with a low cost and low power consumption are driving a transition towards multi-core processors in many consumer and industrial applications. However, the adoption of multi-core processors in the domain of real-time systems faces a series of challenges that has been the focus of great research intensity during the last decade. These challenges arise in great part from the non real-time nature of the hardware arbiters that schedule the access to shared resources, such as the main memory. One solution proposed in the literature is called Memory Centric Scheduling, which defines a separate software scheduler for the sections of the tasks that will access the main memory, hence circumventing the low level unpredictable hardware arbiters. Several Memory Centric schedulers and associated theoretical analyses have been proposed, but as far as we know, no actual implementation of the required OS-level underpinnings to support dynamic event-driven Memory Centric Scheduling has been presented before. *In this paper* we aim to fill this gap, targeting cache based COTS multi-core systems. We will confirm via measurements the main theoretical benefits of Memory Centric Scheduling (e.g. task isolation). Furthermore, we will describe an effective schedulability analysis using concepts from distributed systems.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases real-time, multi-core, memory centric, predictability, implementation, rtos

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.7

Funding *Juan M. Rivas*: This research was funded by Wallonia Region (Belgium) BEWARE grant PARTITA (1610375).

1 Introduction

Advancements in the manufacturing process of integrated electronics, in addition to the sheer size of the general computing market, are increasingly widening the offer and lowering the costs of commercial off-the-shelf (COTS) multi-core processors. While these commercial processors are generally designed with *average* performance in mind, their wide availability and low cost are pushing their adoption into real-time applications where a different set of requirements such as predictability and worst-case guarantees are needed.

[1] Corresponding author

The main challenge of considering COTS multi-core processors for real-time applications can be found in the way some resources are shared between the cores. For instance, the access to the shared main memory is usually scheduled by hardware arbiters that are generally not designed with real-time or predictability considerations in mind. In essence, all the jobs concurrently accessing the shared memory can interfere with each other, thus adding delays to the execution time that are potentially large and very difficult to tightly bound. This interference can be compounded by other inter-dependent sources such as the shared cache: a task could evict a cache line of another task in a different core, which could produce further accesses to the shared memory and interferences down the line. In conclusion, in a multi-core processor the execution time of any task can be affected by any other concurrent task, regardless of priorities or criticalities. This leads to two main problems: (1) the task execution times are inflated due to interferences from other cores, and (2), the worst-case execution time (WCET) estimations tend to be greatly inflated due to the difficulty to predict these inter-core interferences.

This mismatch between the predictability requirements needed in real-time systems and the performance characteristics of available multi-core processors has triggered an intense research effort in the real-time community during the last decade [17]. One line of research proposes additional layers of software and associated mathematical analyses to add predictability to the commercial multi-core platforms. An example of such approach is based on the new task model called the PRedictable Execution Model (PREM) [31]. With PREM, each task code is explicitly divided into coarse grained phases that will access the shared memory (memory phases) and phases that will operate exclusively on cached data and instructions (execution phases). Typically, the memory phases are composed of prefetch or load instructions that copy into the cache the data and instructions needed by a subsequent execution phase, or by write-back instructions to copy updated data from the cache to the main memory.

Leveraging the PREM model, a new type of scheduling scheme called Memory Centric [38] defines high level schedulers for the memory phases with the aim to limit or avoid concurrent memory phases. This way, the contention in the shared memory subsystem can be solved by software, avoiding relying on the low level unpredictable arbiters. Several previous works have proposed different variations of Memory Centric schedulers, a selection of which will be briefly discussed in Section 2. Notwithstanding this body of work, and as far as we know, all these proposals are either theoretical works or implementations relying on static or time triggered scheduling.

The main objective of this paper is to complement those previous approaches, by implementing dynamic Memory Centric Scheduling in an actual Real-time Operating System (RTOS). While the basic ideas to sustain our implementation do not depend on any particular RTOS, we choose to target HIPPEROS [28]. HIPPEROS is built from the ground-up to support multi-core processors and employs an asymmetric master-slave architecture in which the schedulers run on a dedicated core, called the *master core*. By exploiting this architecture the overheads of executing the schedulers are concentrated in the *master core*, while the *slave cores* can execute tasks with a *baremetal level* of overheads.

To tease the outcome of this paper, Figure 1 shows the execution times of a periodic task during a span of 40 seconds. From 0 to 10 seconds the task runs alone in the system, and then a new task is added to a new core every 10 seconds (each core has at most one task). The blue line (contention) shows how the execution time of the task increases any time a new task is added, indicating that it is being affected by interferences from other cores. The orange line (labelled as "this paper") shows the execution times of the same task, but using the implementation we provide in this paper. We can see that now the task execution time remains constant, thus achieving a level of isolation from other tasks.

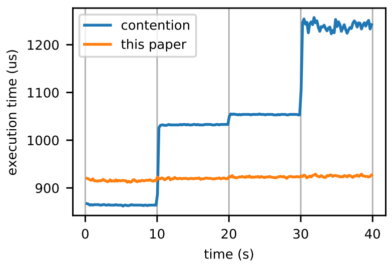


Figure 1 Execution times of a task with different number of competing tasks.

Contributions of this paper

- **Implementation of dynamic Memory Centric Scheduling in HIPPEROS**. This includes a kernel-level scheduler and a user-level API. We target fixed task priority and preemptive memory phases, that can be dynamically invoked either synchronously or asynchronously.
- **An evaluation study** to confirm with measurements the theoretical benefits of Memory Centric Scheduling, namely an execution free of interferences in the main memory.
- **A schedulability analysis** defined by exploiting the similarities between the task model used in Memory Centric Scheduling and the distributed task model [36]. We compare the analytical results with actual measurements.

The paper is organized as follows: Section 2 provides more background on works related to handling interferences in the main memory of multi-core systems, with a focus on Memory Centric Scheduling, and a brief introduction to the HIPPEROS RTOS main relevant features. Section 3 presents a description of the system model and general hardware characteristic assumptions adopted throughout the paper. Section 4 presents our implementation of Memory Centric Scheduling in HIPPEROS. Section 5 describes our schedulability analysis based on distributed systems. Section 6 shows the results of the extensive evaluation performed. Finally, Section 7 provides the main conclusions we have reached in this work, and hints at some general research paths that we could follow in the future.

2 Background and Related Work

In this section we will contextualize this paper in relation to other similar previous works, focusing on approaches that tackle the problem of handling the interferences in the main memory. In Section 2.1 we will briefly describe the basic elements of Memory Centric Scheduling, also providing a selection of papers that develop it. Section 2.2 is dedicated to discuss other approaches similar to Memory Centric Scheduling. Finally, Section 2.3 provides a description of the main relevant characteristics of the RTOS we target in this paper, HIPPEROS.

2.1 Memory Centric Scheduling

In the context of this paper we define Memory Centric Scheduling (MCS) as a scheduling framework of real-time tasks that complies with the following characteristics:

(a) The tasks follow a PREM'like model [31]: tasks have two possible coarse grained states or phases: memory phases (M-Phases) in which the task will access the shared main memory; and execution phases (E-Phases) in which the task operates on cached data and instructions with no access to the shared memory.

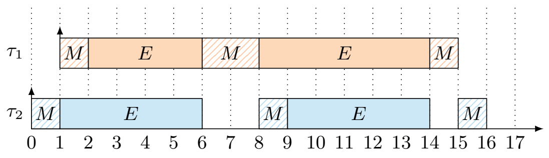


Figure 2 Simple example of Memory Centric Scheduling with two tasks.

(b) There exists a system wide scheduler for the memory phases (memory phases scheduler, MPS) that dynamically decides which memory phase can execute, hence effectively controlling the access to the shared memory.

If the MPS limits the number of concurrent memory phases to 1, the low level non real-time hardware arbiters that control the access to the shared memory are effectively bypassed, avoiding any unpredictable interference delays they could produce. Figure 2 shows an example of MCS with 2 tasks, in which each task executes in a different core, memory phases and execution phases are labeled with M and E respectively, and vertical arrows indicate the task activation. Note that at any time there is at most one memory phase executing.

Memory Centric Scheduling was first introduced by Yao et al [38], proposing a TDMA schedule for the memory phases. That paper also introduces the concept of memory promotion, by which the memory phases are given a higher priority than the execution phases to seek a better schedulability. The work in [9] studies with simulations different partitioned scheduling policies for MCS, and reaches the conclusion that non-preemptive least-laxity first memory phases is the best approach. The papers [3, 22] propose static schedules of the memory and execution phases. MCS with global scheduling has also been studied [2, 39].

The authors of [37] propose a mechanism to hide the latencies of the memory phases by executing them as background DMA transfers. This initial work targets single-core systems, but is later extended for multi-core systems in [4]. These two papers assume special hardware such as double ported scratchpad memories, or a cache based system that supports DMA transfers from the main memory to the cache. The paper [35] presents an integration of MCS in an RTOS, relying on scratchpad memory and implementing the memory phases with DMA transfers scheduled with TDMA.

In this paper we provide an actual implementation of Memory Centric Scheduling in an RTOS (HIPPEROS), where the memory phases can be dynamically invoked. We will also describe latency hiding techniques that do not rely on special hardware. Additionally we will adapt existing analysis techniques for real-time distributed systems, and compare it with our measurements.

2.2 Alternative approaches

Memory bandwidth regulators are usually mentioned as an alternative to MCS. This approach assigns per-core memory bandwidth reservation budgets that do not overload the memory bus. Therefore, each core can run with a guaranteed memory bandwidth, independently on the number of other active cores. A clear advantage of this approach is that it is transparent to the tasks, i.e., no task modifications are needed. On the other hand, with MCS, once a task is granted access to the memory, it can enjoy full access to its bandwidth.

Examples of this approach implemented in software are MEMGUARD [42] or the Multi-Resource Server [10]. A challenge of these techniques is to find an optimal budget assignment per core. A framework called Single Core Equivalence [20] proposes a static and even budget assignment, which is complemented by a mechanism to lock in cache the most frequently used pages [19], and a tool called PALLOC [41] that allocates pages to specific memory banks to allow a certain level of parallelism in the main memory. Subsequent works allow uneven memory bandwidth budget allocations [21, 40, 1, 8]. Recently ARM presented the Memory Partitioning and Monitoring (MPAM) feature in the Armv8.4-A specification [7], which allows to partition the memory bandwidth among different system components.

2.3 Multi-core real-time operating system: HIPPEROS RTOS

HIPPEROS [33] is a multi-core real-time operating system targeting high performance and safety-critical applications running on embedded systems.

The HIPPEROS kernel is a micro-kernel written to natively support multi-core and parallel systems. The kernel software architecture is distributed and asymmetric. This means that, when entering kernel mode, different cores are running different kernel code. In this case, only one core has a different code path than the others, and it is called the *master core*. The *master core* is responsible of the heavy kernel operations such as scheduling, memory management and task states. The other cores are the *slave cores* and have a lighter kernel mode of operations. The rationale behind that asymmetric design is that the *master core*, being dedicated to heavy management operations, is freeing the *slave cores* of that burden, letting them focus on user-mode task execution with few interferences. The *master core* can orchestrate task context switches remotely on the *slave cores* through inter-core interrupts. When a *slave core* requires to invoke the scheduler (for example when a task is completed and exits), it can do so by calling a *remote system call*, triggering an inter-core interrupt to the *master core* which is in turn effectively calling the scheduler module. The *slave core* waits the system call response from the *master core* by spinning the task in user-mode, making it easy to preempt the task when executing a context switch ordered by the *master core*.

This software architecture has the following advantages.

- For a kernel developer perspective, the design and the code base is *much easier to write and maintain*. It avoids having to lock every single structure which is shared among cores in a symmetric kernel design.
- As *slave cores* do not directly interact with each other but only with the *master core*, the asymmetric design has the effect of *reducing peak contention* on spinlocks when cores are trying to acquire them. In fact, the only spinlocks required are to implement the communication mechanisms between *slaves* and *master* to trigger the remote system call procedure and the context switches. Less contention allows for an *improved scalability* (in the worst case) when increasing the number of cores.
- The asymmetric design naturally partitions the data among cores, automatically making *a better use of private caches*. As the state of the system regarding tasks and scheduling is only maintained by the *master core*, this data must not be shared among cores and can stay in the private caches of the *master core*, allowing for both a faster execution of the *master core* kernel path and less cache misses on the *slave* side.

The concept of an asymmetric kernel for multi-core real-time systems was previously studied and validated in prior work [12]. Regarding HIPPEROS, it has been the target of previous contributions, such as its own parallel micro-kernel design [28], power-aware real-time scheduling [30], mixed-criticality scheduling [29] and hardware acceleration for embedded image processing applications [34, 16]. In this paper, the HIPPEROS RTOS is used to showcase an *in-kernel* implementation of the Memory Centric Scheduling approach.

3 System Model

3.1 Hardware Assumptions

We consider a typical commercial off-the-shelf (COTS) multi-core processor, composed of M identical cores with one or more levels of cache connected to the main memory via a shared memory bus. We assume that the last-level cache (LLC) is shared among the cores and employs a write-back policy: an LLC miss produces a share memory access to load a cache line, and possibly another memory bus access to write-back an evicted LLC line. Therefore we can establish that accesses to the shared bus only occur during LLC misses. We also assume that the shared LLC cache can serve concurrent hits from several cores with negligible interference delays. We will see in Section 6 that this assumption holds in our measurements. Additionally, we consider that only the cores can trigger a memory bus access (e.g. no peripheral DMA transfers are allowed). Figure 3 depicts a typical COTS multi-core processor, with 4 cores, private L1 caches and shared L2 cache.

3.2 Task Model

The cores execute a set of N preemptive sporadic tasks $\Gamma = \tau_1, ..., \tau_N$. In this paper we consider fixed task priorities (FTP) partitioned scheduling. We use a modified PREM task model that, in addition to the Execution Phases, defines two types of memory phases, called Prefetch Phases and Write-Back Phases. These phases operate in the following manner:

1. Prefetch Phase (P-Phase): tasks start with a memory phase called Prefetch Phase that prefetches (i.e. copies) the necessary data and instructions from the main memory to the cache.
2. Execution Phase (E-Phase): the task operates on data and instructions cached during the previous phase. As a result, no accesses to the main memory are triggered during this phase.
3. Write-back Phase (W-Phase): this is a memory phase that executes after an E-Phase to copy any updated data from the cache to the main memory. Additionally, the W-Phase also flushes all the previously prefetched cache lines, so any subsequent P-Phase could start with a known clean state.

For a predictable execution, cache lines prefetched during a P-Phase should only be evicted in a controlled manner during a W-Phase. Otherwise, any cache line that is inadvertently evicted could later trigger main memory accesses during the E-Phases, thus breaking the assumptions of the PREM model. Accidental cache line evictions can be produced by the task to itself (self-eviction), by other tasks in the same core (intra-core eviction) or from a different core (inter-core eviction). In this paper we propose to evade intra-core and

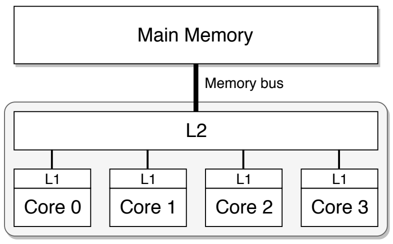


Figure 3 Simplified diagram of a typical commercial quad-core processor with 2 levels of cache.

inter-core evictions by assigning each task a partition of the shared cache. Cache partitioning is a commonly used solution that has been extensively studied [11, 23, 6]. Formally, we define A_i as the set of shared cache partitions assigned to τ_i, and BA_i the total size of the cache partitions assigned to τ_i. In this paper we assume that the cache partition mapping is performed offline. Section 4.4 provides more details about how we implemented cache partitioning for our particular platform.

Regarding the problem of self-evictions, in the context of this paper we can establish that they occur when the size of the data/instructions prefetched in a P-Phase exceeds the size of the cache partition assigned to the task. To handle this situation, we allow tasks to have several sequential batches of P-E-W phases, each one targeting a different portion of the memory with a size up to the size of the task cache partition. Formally, we define B_i as the memory requirement of τ_i, that is, the total size of data and instructions that τ_i needs to execute. If the memory requirement is larger than the task cache partition ($B_i > \mathrm{BA}_i$), the task must invoke several P-E-W phases to cover its memory requirement. We call each P-E-W phase triplet a `Section`.

We define $S_{i,j}$ as the j-th `Section` of task τ_i. Additionally, $P_{i,j}$, $E_{i,j}$ and $W_{i,j}$ are the P-Phase, E-Phase and W-Phase in section $S_{i,j}$, respectively. To avoid self-evictions, each `Section` operates on data/instructions with a size up to BA_i. Thus, the number of `Sections` needed by task τ_i is at least $\lceil \frac{B_i}{BA_i} \rceil$. Furthermore, we assume that the memory requirements of the tasks are fully available at task release, and that they can be partitioned. An example of a PREM task with two `Sections` is shown in Figure 4.

In this paper we target fixed priority scheduling of the memory phases. Accordingly, the tasks have two fixed priority values: its fixed task priority (with a core local scope) and its fixed memory phase priority (with a global scope). In HIPPEROS the task priority can be defined by an external configuration file, or by using the standard API's such as `OpenMP` or `pthreads`. To set the memory phase priority we have implemented a new system call (`memphase_priority_set`).

In single-core systems, the worst-case execution time (WCET) of a task is usually defined as an upper bound of its execution time when it runs *alone* in its core. This definition cannot be maintained in multi-core systems due to inter-core interference delays [17]. Accordingly, in this paper we define the WCET of task τ_i as an upper bound of its execution time when it is running alone in its core, and a known set of tasks are running in other cores. Similarly, we define the worst-case response time (WCRT) of τ_i as an upper bound of its execution time when it runs with a known set of tasks, in the same core and others. Thus, the WCRT includes the WCET of the task, and possible scheduling delays due to tasks in the same core.

This paper will define the system calls to start memory phases, but is not concerned about how to generate the code of the tasks, or how to determine the memory addresses to prefetch. We assume that the memory phases are either defined manually, or using some automatic tool [31, 18].

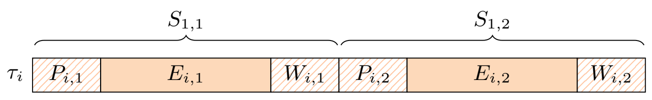


Figure 4 PREM task τ_i with 2 sections.

4 Implementation of Memory Centric Scheduling

In this section we describe the main contribution of this paper, which is a full implementation of Memory Centric Scheduling in an RTOS. In the next subsection we first lay-out the goals and intended behavior which will later shape the implementation.

4.1 Overview and Goals

We identify that an implementation of Memory Centric Scheduling is composed of two main interconnected components: (1) a scheduler for the memory phases and (2) an API for tasks to invoke the start and end of memory phases.

An initial approximation could understand a memory phase as a critical region protected by a mutex located in shared memory, in which the mutex lock and unlock functions would signal the start and end of the memory phases. While this approach can indeed assure that only one memory phase is executing at a time, it restricts their behavior to be non-preemptive. Consequently, any task could be temporarily blocked while requesting the start of a memory phase, independently of any priority assignment.

We propose an architecture similar to that of a mutex but with a *preemptive* nature to avoid those blocking times. Also, the memory phases are scheduled according to their fixed priorities and only one memory phase is allowed to execute at a time. By removing the interference in the shared memory and cache, and prioritizing the memory phases, the target of our implementation is that the response time of any task would only depend on the number of higher priority tasks executing in the same core, and the number of higher priority memory phases system-wide.

It is worth noting that the underlying hardware platform could force the use of non-preemptive memory phases. For example, the cache controller could not support performing a write-back cache operation (e.g. clean or invalidate) while a previous cache operation has yet not finished. This would not allow a W-Phase to preempt another W-Phase. For such situations we support non preemptive memory phases, which can however co-exist with preemptive ones.

In the next subsections we explain the implementation in more detail. Section 4.2 deals with the kernel-level memory phase scheduler and system calls to start and end a memory phase, while Section 4.3 presents the user level API that uses those system calls to request prefetch and write-back phases. In Section 4.4 we will describe how to use existing techniques to analyze our task model.

4.2 Kernel-level: memory phases scheduler (MPS)

As we described in the previous subsection, to implement the memory phases scheduler (MPS) we get inspiration from how mutexes operate, aiming to implement a mechanism that acts like a "*preemptive critical region*".

The two main components of the MPS are the system calls to invoke the start and end of a memory phase, called `memphase_start` and `memphase_end`, respectively. To illustrate how they operate we present a simple example with two tasks, τ_1 and τ_2, in which the memory phases of τ_1 have a higher priority to those of τ_2. Additionally, each task is mapped to a different core, and a third core handles the scheduler and MPS (HIPPEROS *master core*). Figure 5 shows the timeline of their execution, with a focus on the memory phase invoked by

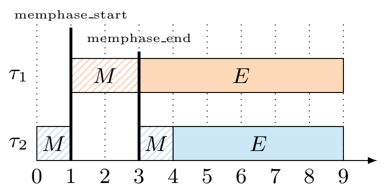


Figure 5 Simple example of scheduling of memory phases and needed system calls.

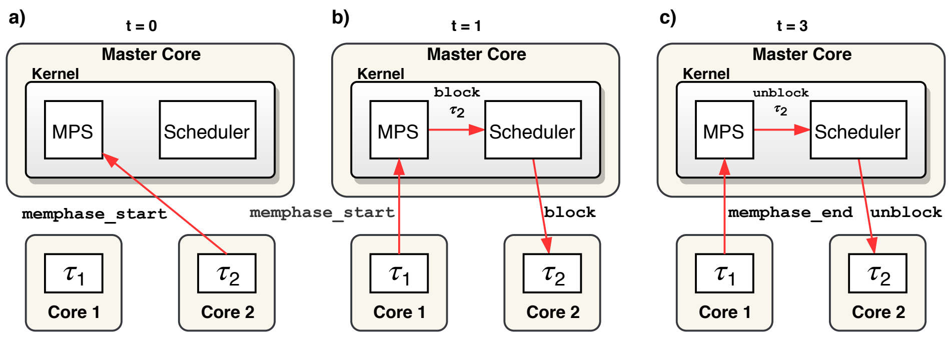


Figure 6 System calls to schedule the simple example of memory phases.

τ_1, while Figure 6 shows the different system calls involved and the interaction between the *master core* and the *slave cores*. It is worth noting that the *master core* is still available to execute user tasks.

First, at $t = 0$, τ_2 requests the start of a memory phase with `memphase_start`, which is immediately granted by the MPS as no other memory phase is executing at that time (Figure 6a). At this point the MPS stores τ_2 in its internal task queue that keeps track on the tasks that have a pending memory phase, ordered according to their priorities. Note that in this context, the MPS grants access to τ_2 by just letting it continue its execution. At $t = 1$, τ_1 requests the start of a memory phase (Figure 6b). At this time the MPS decides to grant access to τ_1 since its memory phase has a higher priority than the memory phase of τ_2. Accordingly, the MPS requests the system scheduler to block the execution of τ_2. At $t = 2$, τ_1 signals the end of its memory phase with the system call `memphase_end` (Figure 6c), which prompts the MPS to unblock τ_2 to let it finish its memory phase.

The pseudocode shown in Listing 1 describes the system call `memphase_start`. The variable “caller” is a pointer to the task requesting the memory phase, while “owner” is a pointer to the task currently executing a memory phase. Basically, the system call always add to the queue the “caller”, and blocks the task with the lowest priority memory phase between “caller” and “owner”, updating the “owner” when necessary. If “owner” is non preemptive, the “caller” is always blocked.

Similarly, Listing 2 shows a pseudocode describing `memphase_end`. When called, this function unblocks the next highest priority memory phase (if any), and assign it as the new “owner”.

4.3 User level API

The previous sub-section presented the mechanisms to invoke and schedule memory phases. It is important to note that, in our implementation, the kernel is not concerned about the contents of the memory phases, or even if they access the main memory or not. We

Listing 1 `memphase_start` pseudocode.

```
memphase_start(nonpreemptive)
      caller = getCaller()
      owner = getFirst(queue)
      if (nonpreemptive)
            setnonpreemptive(owner)
      insert(caller, queue)
      if owner != NULL
            if nonpreemptivemp(owner)
                block(caller)
            else if memprio(caller) > memprio(owner)
                block(owner)
            else
                block(caller)
```

Listing 2 `memphase_end` pseudocode.

```
memphase_end()
      caller = getCaller()
      removeFirst(caller, queue)
      owner = getFirst(queue)
      if owner != NULL
            unblock(owner)
```

provide the semantics of a memory phase in user-space, by creating functions that employ the `memphase_start` and `memphase_end` system calls to protect prefetch and write-back instructions, defining prefetch and write-back phases respectively.

Two variants of this user-space API are implemented:

- **Synchronous memory phases (S-MP)** (Section 4.3.1).
- **Asynchronous memory phases (A-MP)** (Section 4.3.2).

4.3.1 Synchronous Memory Phases (S-MP)

Synchronous Memory Phases (S-MP) are memory phases that are actively executed by the tasks that request them. To illustrate S-MP, Figure 7 shows an example with three tasks τ_1, τ_2, τ_3 in decreasing order of fixed memory phase priority, with each task executing in a dedicated core, and τ_1 released one time instant after τ_2 and τ_3. For simplicity, the labels for the phases do not show the sub-indices. We can see that the tasks follow the sequence P-Phase → E-Phase → W-Phase, each one executed by the requesting task in its core. In the figure we can also see that the MPS decides at each time to schedule the highest priority memory phase, preempting memory phases when necessary.

We implement two functions to request the start of Prefetch and Write-back phases, called `h_memphase_prefetch` and `h_memphase_writeback` respectively. A simplified version of their code is presented in Listing 3 and 4, respectively. We can see that these functions use

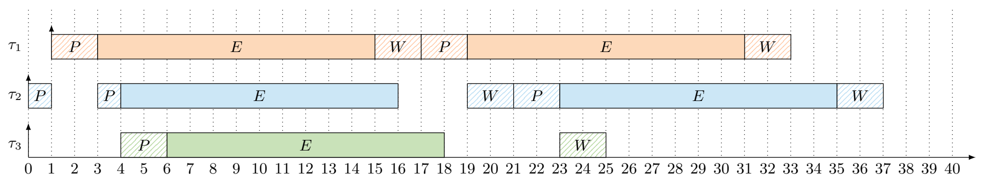


Figure 7 Taskset scheduled with synchronous memory phases (S-MP).

Listing 3 `memphase_prefetch` simplified code.

```
void h_memphase_prefetch(void *addr, size_t bytes) {
        memphase_start();
        prefetch(addr, bytes);
        memphase_end();
}
```

Listing 4 `memphase_writeback` simplified code.

```
void h_memphase_writeback(void *addr, size_t bytes) {
        memphase_start();
        writeback(addr, bytes);
        memphase_end();
}
```

the memory phase start and end system calls to protect the call to a prefetch or write-back function. The only requisite of the prefetch function is that they result in data/instructions copied to the task cache partition or private cache, while the write-back function must be able to copy the updated data into the shared memory and leave the cache partition in a clean state. These requirements represent basic functionalities that are commonly implemented in any commercial processor. Implementation details for our particular platform will be provided in Section 4.4.

In this initial version of the implementation we only target the prefetch and write-back of data (i.e. not instructions). The evaluation section will show how this limitation provides good results. Additionally, the current API is limited to contiguous memory prefetches and write-backs (i.e. defined by base address + memory size).

4.3.2 Asynchronous Memory Phases (A-MP)

An Asynchronous Memory Phase (A-MP) is a memory phase that is executed by a dedicated worker in the background, which we call the Memory Phase Worker (MPW). Crucially, the main benefit of A-MP is that the execution times of these background memory phases do not contribute to the execution times of the requesting tasks. It is worth noting that, contrary to previous proposals for background memory phases [37, 4], our implementation does not rely on any special hardware like scratchpad memories or support for DMA transfers from memory to cache, so it can be accomplished in a wider spectrum of available processors.

A-MP requires that the task cache partitions must be divided into two sub-partitions, which we call the A and B sub-partitions. Formally, given a task τ_i, we define A_i^A and A_i^B as its two sub-partitions, with $A_i = A_i^A \cup A_i^B$. The basic idea behind A-MP is that while an E-Phase is working on data from a given sub-partition, background memory phases are preparing the other sub-partition. By switching the executing sub-partition, we can effectively hide the execution times of the memory phases while continuously executing E-Phases. We extend the names of the phases to indicate in which sub-partition they are operating, e.g. P^A, E^A and W^A express a P, E and W Phases operating on sub-partition A, respectively.

The MPW is implemented as a task which just executes memory phases on behalf of other tasks, therefore it must have access to the same address space as the requesting tasks. If all the tasks share the same address space (e.g. single-page table), a single MPW is enough. Otherwise, each task must spawn its own MPW thread. Following the philosophy of HIPPEROS, by default we map the MPW(s) to the *master core*. As a result, if A-MP is used, the overheads due to the execution of the memory phases are localized in the *master core*.

Listing 5 A-MP API.

```
void h_memphase_init_a(void);
h_mp_request_t h_memphase_prefetch_a(void *addr, size_t bytes, u32 part);
h_mp_request_t h_memphase_writeback_a(void *addr, size_t bytes, u32 part);
bool h_memphase_finished_a(h_mp_request_t *request);
void h_cache_set_partition(u32 partition);
void h_cache_revert_partition(void);
```

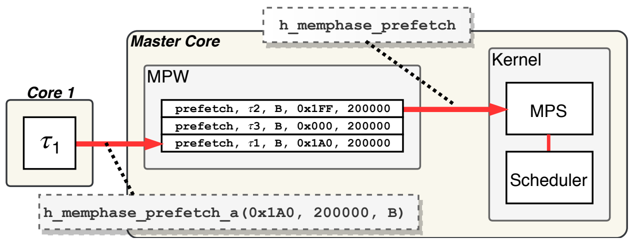

h_memphase_prefetch
Master Core
Kernel
MPW
Core 1
prefetch, τ2, B, 0x1FF, 200000
prefetch, τ3, B, 0x000, 200000
prefetch, τ1, B, 0x1A0, 200000
MPS
τ_1
Scheduler
h_memphase_prefetch_a(0x1A0, 200000, B)

Figure 8 Elements involved in A-MP.

Listing 5 shows the functions implemented to use A-MP. Before it can be used, the MPW must be initialized by using the API function `h_memphase_init_a`, after which it remains waiting for memory phase requests. A memory phase request is stored in a data type called `h_mp_request_t` which has 4 elements: type (i.e. prefetch or write-back), data memory address and size, and the cache sub-partition to use (A or B). Tasks can request an A-MP by using functions `h_memphase_prefetch_a` and `h_memphase_writeback_a` for P and W Phases respectively, which return a `h_mp_request_t` data structure. The parameters of these request functions are the memory address region of data to prefetch/writeback (`addr` and `bytes`), and which task cache sub-partition to target (`part`). Tasks can determine if an A-MP has finished by using the function `h_memphase_finished_a`.

Once a request is received by the MPW, it is stored in an internal FIFO queue. The MPW serves these requests by performing S-MP's. Figure 8 illustrates an example of how the A-MP requests operate, in which we can see that the MPW queue is filled with prefetch requests, which are served by requesting an S-MP with function `h_memphase_prefetch`.

It is important to note that the MPW is just another task executing in the system, and as such it has its own cache partition assignment. Accordingly, when it performs a memory phase on behalf of another task, it must temporarily change its own cache partition to the appropriate A or B sub-partition of the requesting task. This way the S-MP performed by the MPW will operate on the correct cache partition. Once the request has been served, the MPW cache partition returns to its default value. We implement two functions to dynamically set the task cache partition: `h_cache_set_partition` to set a specific partition, and `h_cache_revert_partition` to return to the default task partition. Details on how this dynamic cache partition mapping is implemented are detailed in Section 4.4.

It is worth mentioning that since the A-MP are just S-MP executed by the MPW, both S-MP and A-MP can coexist in the same system without further modifications. In our implementation we give the MPW memory phases the lowest priority, so they do not interfere with other tasks S-MP requests. Additionally, the MPW task itself has the highest priority, so any other task mapped in the *master core* would not preempt it and delay the execution of other tasks A-MP's.

To illustrate the benefits of A-MP, Figure 9 shows the same task-set as Figure 7, with the difference that now the tasks use A-MP. In the example, the tasks start by invoking an S-MP to prefetch the initial batch of data to sub-partition A of each task. This request is synchronous because the cache partitions start with a clean state, so there is no benefit in requesting an A-MP prefetch and wait for it to finish.

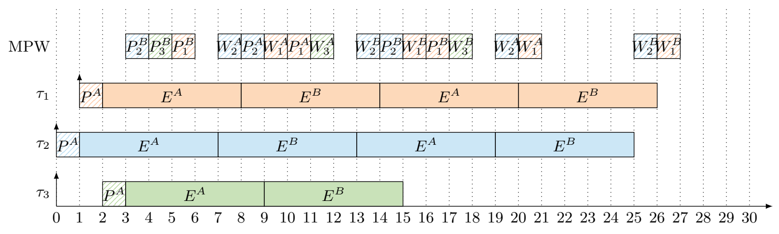

MPW
P_2^B P_3^B P_1^B W_2^A P_2^A W_1^A P_1^A W_3^A W_2^B P_2^B W_1^B P_1^B W_3^B W_2^A W_1^A W_2^B W_1^B
τ_1
P^A E^A E^B E^A E^B
τ_2
P^A E^A E^B E^A E^B
τ_3
P^A E^A E^B
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

Figure 9 Task-set scheduled with asynchronous memory phases (A-MP).

If we now focus on τ_1, we can see that it starts its first execution phase E^A at $t = 2$. At that moment, the task also requests an A-MP prefetch for its sub-partition B (P_1^B), which is served by the MPW at $t = 5$. By the time E^A finished, that prefetch request has already finished so E^B can start without delay ($t = 8$). In general, we can see that while a task is executing an E-Phase on a partition, the MPW is preparing the other sub-partition with the pertinent write-back and prefetch phases, which allows a continuous execution of E-Phases. As a result, the execution time of the tasks is just composed of the execution time of its E-Phases plus an initial synchronous P-Phase. It is worth noting that with A-MP, the phases operate on half the data size compared to S-MP, as they operate on a sub-partition.

In conclusion, A-MP has the potential to *drastically* reduce the execution times of the tasks, at the expense of adding workload in the *master core*. In practice, the *master core* only remains available for non real-time tasks, or real-time tasks with big slack times. Furthermore, it is trivial to see that the benefits of A-MP can only be realized if the E-Phases are long enough to completely cover the execution time-span of the background memory phases.

4.4 Implementation Details

We implemented the Memory Centric Scheduling elements described in Section 4 in version 18.09 of HIPPEROS. While HIPPEROS also supports ARMv8, Intel x86 and PowerPC, we focused our efforts on the widely available ARMv7 architecture. Specifically, we target the NXP i.MX6Q system-on-chip (SoC), composed of 4 Cortex A9 cores, a 16-way 1MB shared L2 cache with the L2C-310 cache controller, and 1 GB of DDR RAM.

Cache partitioning

As described in Section 3, our implementation of MCS requires that each task is guaranteed a dedicated cache partition, which is proposed to avoid intra-core and inter-core cache evictions. Any of the available cache partitioning solutions [24, 19, 6, 11] that provides that guarantee could be used with our implementation. Nonetheless, some caveats must be taken into account, which are described below.

When the cache partitioning can only be achieved at the core level, all the tasks in the same core share the same cache partition, and thus could evict lines of each other. Under this situation, our implementation of MCS is restricted to one task per core, or to multiple non preemptive tasks per core. To implement cache partitioning in this paper, and without loss of generality, we will exploit a feature in the L2C-310 cache controller called *lockdown by master*, which restricts the cache allocations of each core to a particular set of L2 cache-ways, thus effectively achieving core-level cache partitioning. As a consequence, to meet the requirements of our implementation of MCS, we will consider only one task per core. Nevertheless, this limitation does not curtail us from pursuing the objectives of this paper. For the evaluation of our implementation of MCS, we are mainly focused on studying the contention (or lack thereof) in the shared memory. Accordingly, we view the cores as mere producers of memory requests, with no regard to which particular task produced it. For this objective, a configuration of just one task per core is sufficient.

Special consideration must also be taken with systems with a high number of preemptive tasks. Here the cache partitions could get very small, and as a consequence more memory phases would be needed, increasing the overall overheads in the system. These MCS related overheads compound with other pre-existing preemption delays [5]. To mitigate this problem, our implementation of MCS supports non preemptive tasks, which allows core-level cache partitioning and therefore larger partition sizes.

Finally, we assign the HIPPEROS *master core* an L2 cache partition that is big enough to meet the memory requirements of the kernel. This way we can assume that the kernel does not interfere with the tasks in neither the L2 cache nor in shared memory.

Memory phases

We have considered two types of memory phases: prefetch and write-back phases.

The objective of the prefetch phases is to copy lines from the shared memory into the task cache partition. As we have previously stated, in this initial implementation we only target the prefetch of data. For the prefetch we use the ARM `PLD` instruction, which has two main caveats:

1. This instruction copies data to the L1 cache only. This is not a problem because, with *lockdown by master*, any eviction in the L1 cache is allocated into the task L2 cache partition. As a result, all the data prefetched with `PLD` would end up in the L1 cache (private to the core) or in the L2 cache partition.
2. `PLD` is generally defined as a hint instruction, that is, it is not guaranteed that it would produce any effects. However, in our evaluation we have confirmed that, at least in the i.MX6Q SoC we used, this instruction always performs its operation.

As an alternative, ARM defines an optional component in the Cortex-A9 core called the Preload Engine (PLE), which can be used to program loads of selected regions of memory into L2. This component nicely fits the objective of the prefetch phases, but unfortunately is not available in our SoC. It is important to note that we have disabled the speculative hardware prefetchers, which would interfere with our own P-Phases.

Regarding the write-back phases, their objective is two-fold: (1) to copy into the shared memory any data updated during a previous execution phase, and (2) leave the L2 cache partition in a clean state. This can be achieved with common flush cache operations, targeting the L1 private cache and L2 cache partition.

Memory phases worker (MPW)

As described in Section 4.3.2, the Memory Phases Worker (MPW) is a task that performs memory phases on behalf of other tasks. Therefore, it must dynamically change its cache partition to match that of the requesting task. With the *lockdown by master* feature described before, we can perform this partition switch by dynamically changing the core cache ways assignment, adding a necessary L1 flush before the switch to avoid inter-partition pollution.

5 Schedulability Analysis

From an analytical point of view, the main consequence of employing S-MP or A-MP is that the unpredictable interferences in the shared memory and cache are now replaced by predictable scheduling delays. This characteristic can lead to a simplification in the

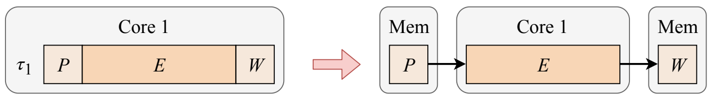


Figure 10 Transformation from PREM task to end-to-end flow (distributed task).

mathematical frameworks needed to determine execution and response time bounds, compared to the fully contended case. Essentially, with S-MP or A-MP, the response time of the tasks is composed of two main components:

1. The execution times of the phases, which are not going to suffer from contention in the shared memory or interferences due to any type of unplanned cache evictions. This removes a great source of uncertainty that usually leads to an over-inflation of the execution time estimations.
2. Scheduling delays: these can come from other tasks in the same core (intra-core), which is the classical scheduling problem [13], or from other cores (inter-core) due to the scheduling of the memory phases, which also have a predictable nature.

If we make the simplifying assumption that other sources of inter-core interferences besides the shared memory and cache are negligible, we can establish that the execution times of the task phases is not affected by the number of tasks in the system. In this situation, with S-MP or A-MP, the response times of any task would only depend on the number of higher priority tasks in its own core, and the number of higher priority memory phases system-wide. In the evaluation section (Section 6) we will see how this assumption mostly holds true in the measurements.

To define a formal response time analysis, and as already hinted by [39], we can draw similarities between PREM and the distributed task model. In distributed systems, the tasks, also sometimes called transactions or end-to-end flows, are formed by a sequence of sub-tasks, each executing in a different processing resource (e.g. processor or network). A sub-task can be a computational task in a processor, or a message scheduled and sent via a network, that could trigger a further sub-task in the recipient processor. Additionally, these sub-tasks are statically mapped to a processing resource due to the high costs that migration would induce in a distributed system. Accordingly each processing resource typically has its own scheduler for the sub-tasks it contains.

In view of this, we can model our PREM tasks as a distributed task: P-Phases and W-Phases are modelled as sub-tasks executing in a *memory* processing resource, and the E-Phases are sub-tasks executing in their original cores. Figure 10 illustrates this transformation with a simple task that uses S-MP, where “Mem” is the *memory* processing resource. For a task that uses A-MP, its equivalent distributed task is composed of just two sub-tasks: one for the initial S-MP needed, and another for the sum of all E-Phases.

The distributed task model has been extensively studied, with several analysis techniques proposed to calculate response times, a number of which are implemented in readily available open-source applications. One of these tools is MAST [25, 14], which implements the seminal holistic analysis [36], and several offset based analyses [26, 27] with different levels of precision and complexity that improves on the holistic analysis, all for sporadic tasks. Section 6 will compare these analytical techniques with actual measured response times. Additionally, compositional analysis techniques can be applied to use different scheduling policies for each processing resource [32][15]. This could enable for example the analysis of Memory Centric Scheduling with EDF memory phases and fixed task priorities execution phases.

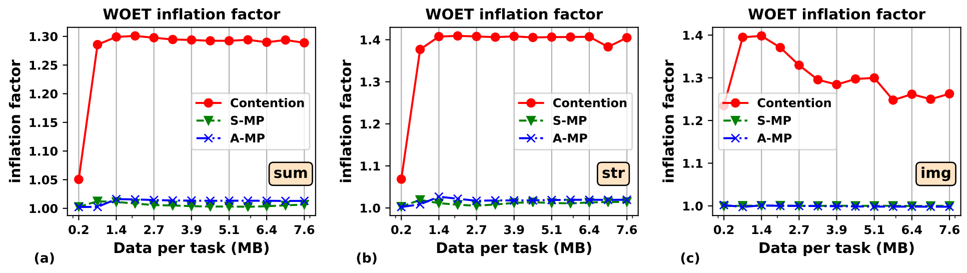


Figure 11 Inflation factors for (a) Sum tasks, (b) Str tasks, and (c) Img tasks.

6 Evaluation

In this section we present the evaluation results of the Memory Centric Scheduling implementation described in Section 5, based on actual execution time measurements of different types of tasks. The framework was implemented in HIPPEROS 18.09, and the hardware platform targeted is a Boundary Devices BD-SL-I.MX6 development board, which includes an NXP i.MX6Q SoC composed of 4 Cortex-A9 cores, 1MB of 16-Way L2 cache with the L2C-310 controller, and 1GB of DDR3 RAM. We label the cores as Core 0 to Core 3, and assign Core 3 as the HIPPEROS *master core*, which means that this core executes the system scheduler, the memory phases scheduler (MPS), and the memory phases worker (MPW) for A-MP.

The main focus is to evaluate how our implementation deals with the contention in the shared memory, but is not concerned about intra-core scheduling. Accordingly, we only map up to one task per core. The index of the task also indicates to which core it is mapped, e.g., τ_0 is mapped to Core 0. Since we only consider one task per core, the concepts of WCET and WCRT as defined in Section 3 became interchangeable during this evaluation.

We wrote three types of tasks: *sum*, which just sum batches of numbers; *str* which encrypts strings; and *img* which applies a Gaussian blur filter on images. We compare 3 scheduling configurations: **A-MP** (from Section 4.3.1), **S-MP** (from Section 4.3.2), and **Contention**. In the latter, the tasks do not invoke memory phases, and as such the hardware arbiters handle the contention in the access to the main memory. For A-MP and S-MP, the task indices also indicate the priority of its memory phases, with τ_0 having the highest priority. We vary the number of tasks in the system, and the task memory requirements from 200 KB to 7.8 MB. Each task is given a partition of 4 cache ways, which implies that for 200KB, the tasks just need 1 `Section`, while for 7.8MB the tasks need 40 `Sections`. To obtain statistically relevant results, a total of 430000 executions were performed.

The evaluation is based on measurements of the execution times of the tasks. We define $\text{woet}_i^K(m)$ as the worst-observed execution time (WOET) of τ_i, for a system with m total tasks, and a K scheduling configuration (C for contention, S for S-MP, and A for A-MP).

Sequential Tasks

We first study the inflation factors of each configuration. For a system with up to m tasks, we define the inflation factor of τ_i as the ratio between its WOET with $(m-1)$ co-runners and its execution time running alone (0 co-runners), that is, $\text{woet}_i(m)/\,\text{woet}_i(1)$. An inflation factor above 1 indicates that the task execution time is affected by contention delays in the shared memory. Figure 11 shows the inflation factors of τ_0, for different data sizes and the three types of tasks (*sum*, *str* and *img*). We can see that as expected, with Contention scheduling, the inflation factor clearly grows above 1, with a measured maximum of up to 1.4. On the other hand, we can observe that with S-MP and A-MP, the inflation factor remains at approximately 1, with a slight advantage to S-MP. It is worth remembering that τ_0 has the highest priority memory phases, which implies that here it does not suffer from any type of scheduling delays. This result also confirms that S-MP and A-MP can indeed isolate the execution times of the tasks with respect the number of tasks accessing the shared memory.

We now study the improvement factor of using A-MP and S-MP with respect to Contention. We define the improvement factor of τ_i as the ratio between its WOET with S-MP or A-MP and its WOET with Contention. Here we consider systems with 3 tasks, that is, the improvement factor is $\text{woet}_i(3)/\,\text{woet}_i^C(3)$. Therefore, an improvement factor below 1 indicates that the task WOET improves with respect to Contention. Figure 12 shows the improvement factor of τ_0, τ_1, τ_2 for different data sizes and focusing on *sum* tasks. With a general view of the figure we can reach two main conclusions. First, with S-MP or A-MP, the WOET improvement gets more apparent the higher the size of the data is, remaining constant above about 2 MB. Second, we can confirm that A-MP yields lower WOET than S-MP. This is expected, as the majority of the memory phases with A-MP do not contribute to the execution time of the task. In more detail, we also see that for τ_0 (highest priority memory phases), its WOET is always better with S-MP or A-MP than with Contention. This result, in addition to the inflation factors shown before, indicate us that with S-MP or A-MP we can achieve at the same time task execution time isolation and a reduction in the execution times. For τ_1, τ_2 the WOET with S-MP is increased with respect Contention for low data sizes (< 0.8 MB). This indicates that in this case, the overheads of S-MP (execution of memory phases and scheduling delays) cannot be compensated by the lower execution times of the fully cached E-Phases. On the other hand, In Figure 12b and c, we can identify that for high data sizes, S-MP also grants improvements in τ_1, τ_2 over Contention.

Until now we have focused on evaluating the WOET's. Another important factor in real-time systems is the variability of the execution times, also called jitter. Figure 13 shows the average observed execution times (AOET) of τ_0, with added error bars (black vertical lines) that show the WOET and best observed execution time (BOET) of each configuration. Additionally, the overlapping patterned bars indicate the portion of the execution time that is contributed by the "operation" portion of the task, which is the whole task in the case of Contention, and the E-Phases in S-MP and A-MP. In the figure we can first confirm that the jitter with Contention is clearly higher, especially when more tasks with more data are involved. This is expected, as in these situations, with more shared memory accesses, there is a higher chance of being delayed due to contention in those accesses, which varies between different executions. On the other hand, S-MP and A-MP provide execution times with no obvious jitter. This is the desired result, and it is expected as the main source of variability (contention in shared memory) is solved by software in a predictable and constant manner. Moreover, the figure also allows us to attest that the execution times of the E-Phases

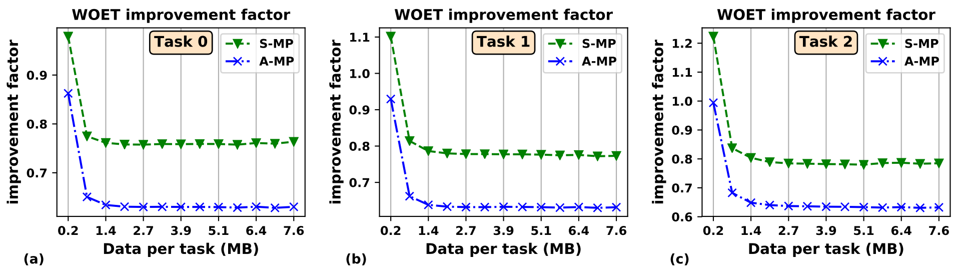


Figure 12 Improvement factors of sum tasks, for (a) Task 0, (b) Task 1 and (c) Task 2.

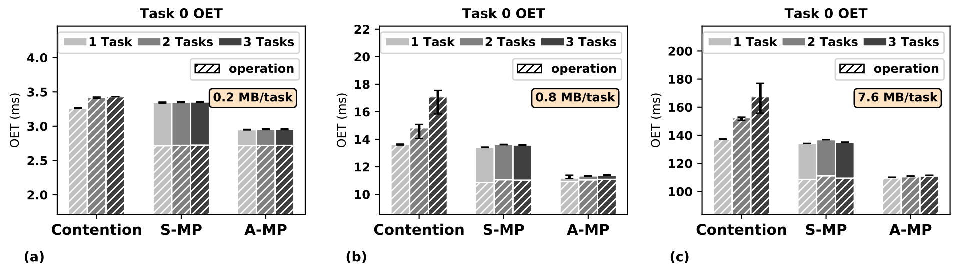


Figure 13 Average OET of Task 0, with maximum and minimum OET as error bars, for (a) 0.2 MB per task, (b) 0.8 MB per task, and (c) 7.6 MB per task.

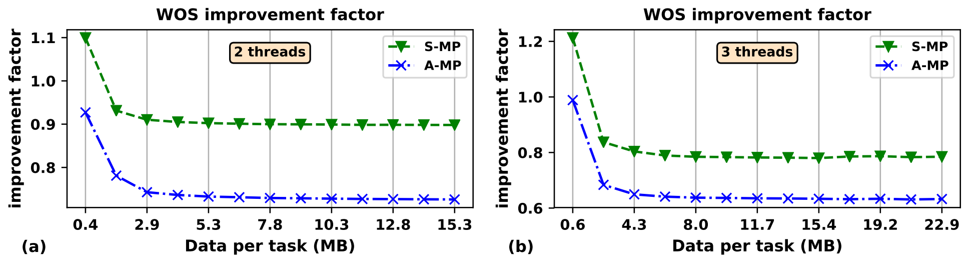


Figure 14 Improvement factors of parallel tasks with (a) 2 threads and (3) threads.

(patterned portion) is the same for S-MP and A-MP. As expected, the only difference between both originates from the higher number of memory phases that are included in the execution times with S-MP.

Multi-threaded Tasks

We now evaluate the benefits of using S-MP and A-MP with multi-threaded tasks. For this, we modify the *sum* tasks used before, so now they spawn 2 to 3 threads to process in parallel a portion of its data. Each parallel thread requests its own memory phases. We define the span of a parallel task as its execution time, which is the time interval between the first thread is spawned, until the last thread finishes. We assume that the work performed outside these parallel threads is negligible. Similarly to the WOET, we define the worst observed span (WOS) as the maximum measured span, denoted as wos^K for a K scheduling configuration. Figure 14 shows the improvement factors of the WOS for S-MP and A-MP over Contention, for parallel tasks with 2 and 3 threads, and different task data sizes. In the figure we see similar results as with sequential tasks before: (1) only S-MP for low data sizes sees and increase in the worst observed span times, and (2) A-MP gets a substantial reduction in the execution times compared to S-MP and Contention.

Schedulability Analysis

We finally compare the measured WOET's with the bounds obtained with analysis techniques originally created for distributed systems. We model the PREM tasks as showed in Section 5, and feed them as input to the MAST tool [14]. The models need worst-case execution times for each task phase. For this, we use the highest task phases measured execution times. Then, we apply three different analysis techniques: HOL, which is the original Holistic analysis by

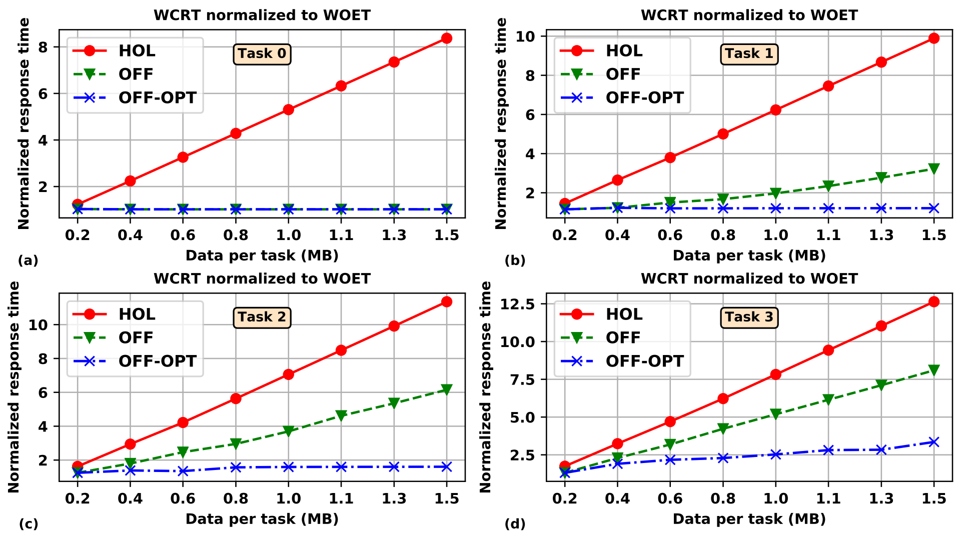


Figure 15 Analytical worst-case response times normalized to WOET, for (a) Task 0, (b) Task 1, (c) Task 2, and (d) Task 3.

Tindell [36]; OFF, which is a subsequent off-based analysis [26]; and OFF-OPT which is an optimization of the original offset-based analysis [27]. We only consider S-MP, as A-MP is modelled as a pure S-MP task with only one memory phase at the beginning (Section 5).

We consider systems with 4 tasks. In Figure 15 we show, for each task in the system, the worst-case response times obtained by these analysis techniques, normalized to the measured WOET's. First, we can confirm that the normalized response times are never below 1. This indicates that the analyses never underestimated the observed response times. Furthermore, we can attest that HOL provides the highest overestimation, followed by OFF and then OFF-OPT. For τ_0 (Figure 15a), which has the highest priority memory phases, both OFF and OFF-OPT recognized that its memory phases always execute without delay, and thus provide worst-case response times estimations very close to the WOET's. For the rest of the tasks, in which memory phases scheduling delays must be accounted for, OFF-OPT provides clearly the best results, with normalized response times clearly below 2 for τ_1, τ_2, and below 4 for τ_3. It is important to note that a higher task data size implies more memory phases. Additionally, a task with low priority memory phases (e.g. τ_3) is impacted by more higher priority memory phases, so the scheduling scenario to analyze increases in complexity.

We think that one of the main sources of overestimation of these analysis techniques may be due to task release phase considerations. The analytical concept of building a worst-case situation (i.e. critical instant) by releasing all the tasks at the same time does not always hold for distributed tasks [27]. Accordingly, part of the challenge of distributed analysis techniques is in finding the tasks release time phases that lead to the worst case, which may not have been arisen during the actual measured executions.

7 Conclusions and Future Work

In this paper we have tackled the problem of contention and interferences in the shared memory of multi-core processors, which is a great impediment for the adoption of this type of processors in real-time applications. Precisely, we have presented and tested an implementation of Memory Centric Scheduling (MCS). The main idea of MCS is to solve the access to the shared memory via a software-based dynamic scheduler, thus avoiding low level and non real-time hardware arbiters.

While previous papers have proposed MCS from a theoretical standpoint, this paper, to the best of our knowledge, is the first time it has been implemented in an actual RTOS supporting dynamic scheduling. The implementation was carried out in an asymmetric multi-core RTOS called HIPPEROS, which locates the scheduler in a dedicated core, called the *master core*.

Two variants of MCS were implemented that can be used in commercial processors: Synchronous Memory Phases (S-MP), in which the tasks execute the memory phases in the foreground, and Asynchronous Memory Phases (A-MP), where the memory phases are executed in the background by a dedicated task. We evaluated the implementation in a quad core commercial processor, and confirmed via measurements the theoretical benefits of Memory Centric Scheduling: the tasks effectively execute free of interferences in the shared memory sub-system.

Specifically, by scheduling the memory phases with fixed priorities, we showed that the tasks execution times were shielded from interferences from lower priority tasks. By extension, the highest priority task has execution times that do not depend on the number of tasks in the multi-core system. The main consequence of these isolation effects is that the execution times can be more easily bounded, compared to the fully contended case. Furthermore, we viewed that with MCS the worst-case observed execution times can be lower compared to the fully contended case, specially for tasks with high memory requirements.

We also applied existing and proven analysis techniques for distributed systems, by exploiting the similarities between the task model used in MCS and the distributed task model. We showed how these techniques can indeed provide safe bounds of the execution times of MCS systems, that are also in many cases very close to the observed values.

For future work we are planning: (1) to extend the evaluation to more than one task per core, analyzing also the benefits of non-preemptibility; (2) to compare with other approaches such as memory bandwidth regulators (e.g. MEMGUARD [42]); (3) to evaluate new scheduling schemes for the memory phases (e.g. LLF) and the MPW.

References

1 Ankit Agrawal, Renato Mancuso, Rodolfo Pellizzoni, and Gerhard Fohler. Analysis of Dynamic Memory Bandwidth Regulation in Multi-core Real-Time Systems. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 230–241. IEEE, December 2018.

2 Ahmed Alhammad and Rodolfo Pellizzoni. Schedulability analysis of global memory-predictable scheduling. In *Proceedings of the 14th International Conference on Embedded Software - EMSOFT '14*, pages 1–10, 2014.

3 Ahmed Alhammad and Rodolfo Pellizzoni. Time-predictable execution of multithreaded applications on multicore systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*, pages 1–6, New Jersey, 2014. IEEE Conference Publications.

4 Ahmed Alhammad, Saud Wasly, and Rodolfo Pellizzoni. Memory efficient global scheduling of real-time tasks. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 285–296, 2015.

5 Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems*, 48(5):499–526, September 2012.

6 Sebastian Altmeyer, Roeland Douma, Will Lunniss, and Robert I. Davis. On the effectiveness of cache partitioning in hard real-time systems. *Real-Time Systems*, 52(5):598–643, September 2016.

7 ARM. Arm® Architecture Reference Manual Supplement Memory System Resource Partitioning and Monitoring (MPAM). URL: `https://developer.arm.com/docs/ddi0598/latest`.

8 Muhammad Ali Awan, Pedro F. Souto, Benny Akesson, Konstantinos Bletsas, and Eduardo Tovar. Uneven memory regulation for scheduling IMA applications on multi-core platforms. *Real-Time Systems*, 55(2):248–292, April 2019.

9 Stanley Bak, Gang Yao, Rodolfo Pellizzoni, and Marco Caccamo. Memory-Aware Scheduling of Multicore Task Sets for Real-Time Systems. In *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 300–309. IEEE, August 2012.

10 Moris Behnam, Rafia Inam, Thomas Nolte, and Mikael Sjödin. Multi-core composability in the face of memory-bus contention. *ACM SIGBED Review*, 10(3):35–42, October 2013.

11 Bach D. Bui, Marco Caccamo, Lui Sha, and Joseph Martinez. Impact of Cache Partitioning on Multi-tasking Real Time Embedded Systems. In *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 101–110. IEEE, August 2008.

12 Felipe Cerqueira, Manohar Vanga, and Björn B. Brandenburg. Scaling global scheduling with message passing. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 263–274, April 2014.

13 Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):1–44, October 2011.

14 Michael González Harbour, Jose Javier Gutiérrez, José M. Drake, Patricia López Martínez, and Jose Carlos Palencia. Modeling distributed real-time systems with MAST 2. *Journal of Systems Architecture*, 59(6):331–340, June 2013.

15 Arne Hamann, Marek Jersak, Kai Richter, and Rolf Ernst. Design space exploration and system optimization with symTA/S - Symbolic timing analysis for systems. *Proceedings - Real-Time Systems Symposium*, pages 469–478, 2004.

16 Tobias Kalb, Lester Kalms, Diana Göhringer, Carlota Pons, Ananya Muddukrishna, Magnus Jahre, Boitumelo Ruf, Tobias Schuchert, Igor Tchouchenkov, Carl Ehrenstråhle, Magnus Peterson, Flemming Christensen, Antonio Paolillo, Ben Rodriguez, and Philippe Millet. *Developing Low-Power Image Processing Applications with the TULIPP Reference Platform Instance*, pages 181–197. Springer International Publishing, Cham, 2019.

17 Claire Maiza, Hamza Rihani, Juan M Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I Davis. A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. *ACM Computing Surveys*, 52(3), July 2019.

18 R. Mancuso, R. Dudko, and M. Caccamo. Light-PREM: Automated software refactoring for predictable execution on COTS embedded systems. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10, August 2014.

19 Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54. IEEE, April 2013.

20 Renato Mancuso, Rodolfo Pellizzoni, Marco Caccamo, Lui Sha, and Heechul Yun. WCET (m) estimation in multi-core systems using single core equivalence. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, pages 174–183. IEEE, 2015.

21 Renato Mancuso, Rodolfo Pellizzoni, Neriman Tokcan, and Marco Caccamo. WCET Derivation Under Single Core Equivalence With Explicit Memory Budget Assignment. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 76. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

22 Joel Matějka, Björn Forsberg, Michal Sojka, Zdeněk Hanzálek, Luca Benini, and Andrea Marongiu. Combining PREM Compilation and ILP Scheduling for High-performance and Predictable MPSoC Execution. In *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM'18, pages 11–20, New York, NY, USA, 2018. ACM.

23 Sparsh Mittal. A Survey of Techniques for Cache Partitioning in Multicore Processors. *ACM Computing Surveys*, 50(2):1–39, May 2017.

24 Frank Mueller. Compiler Support for Software-based Cache Partitioning. In *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, &Amp; Tools for Real-time Systems*, LCTES '95, pages 125–133, New York, NY, USA, 1995. ACM.

25 University of Cantabria. MAST. URL: `https://mast.unican.es/`.

26 Jose Carlos Palencia and Michael Gonzalez Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, pages 26–37. IEEE Comput. Soc, 1998.

27 Jose.C. Palencia and Michael Gonzalez Harbour. Exploiting precedence relations in the schedulability analysis of distributed real-time systems. In *Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No.99CB37054)*, pages 328–339. IEEE Comput. Soc, 1999.

28 Antonio Paolillo, Olivier Desenfans, Vladimir Svoboda, Joël Goossens, and Ben Rodriguez. A New Configurable and Parallel Embedded Real-time Micro-Kernel for Multi-core platforms. *OSPERT 2015*, pages 25–27, 2015.

29 Antonio Paolillo, Paul Rodriguez, Vladimir Svoboda, Olivier Desenfans, Joël Goossens, Ben Rodriguez, Sylvain Girbal, Madeleine Faugère, and Philippe Bonnot. Porting a safety-critical industrial application on a mixed-criticality enabled real-time operating system. In *Proceedings of the 5th Workshop on Mixed-Criticality Systems*, December 2017.

30 Antonio Paolillo, Paul Rodriguez, Nikita Veshchikov, Joël Goossens, and Ben Rodriguez. Quantifying Energy Consumption for Practical Fork-Join Parallelism on an Embedded Real-Time Operating System. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, RTNS '16, pages 329–338. ACM, 2016.

31 Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A Predictable Execution Model for COTS-Based Embedded Systems. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279. IEEE, April 2011.

32 Juan M. Rivas, Jose Javier Gutiérrez, Jose Carlos Palencia, and Michael González Harbour. Schedulability analysis and optimization of heterogeneous EDF and FP distributed real-time systems. *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 195–204, 2011.

33 HIPPEROS SA. The real-time OS for high performance embedded systems. `https://www.hipperos.com/maestro/`. 2019-02-04.

34 Ahmad Sadek, Ananya Muddukrishna, Lester Kalms, Asbjørn Djupdal, Ariel Podlubne, Antonio Paolillo, Diana Goehringer, and Magnus Jahre. Supporting Utilities for Heterogeneous Embedded Image Processing Platforms (STHEM): An Overview. In Nikolaos Voros, Michael Huebner, Georgios Keramidas, Diana Goehringer, Christos Antonopoulos, and Pedro C. Diniz, editors, *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, pages 737–749, Cham, 2018. Springer International Publishing.

35 R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo. A Real-Time Scratchpad-Centric OS for Multi-Core Embedded Systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11, April 2016. `doi:10.1109/RTAS.2016.7461321`.

36 Ken Tindell and John Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40(2-3):117–134, April 1994.

37 Saud Wasly and Rodolfo Pellizzoni. Hiding Memory Latency Using Fixed Priority Scheduling. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 75–86, 2014.

38 Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Emiliano Betti, and Marco Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems*, 48(6):681–715, November 2012.

39 Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Heechul Yun, and Marco Caccamo. Global Real-Time Memory-Centric Scheduling for Multicore Systems. *IEEE Transactions on Computers*, 65(9):2739–2751, 2016.

40 Gang Yao, Heechul Yun, Zheng Pei Wu, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Schedulability Analysis for Memory Bandwidth Regulated Multicore Real-Time Systems. *IEEE Transactions on Computers*, 65(2):601–614, February 2016.

41 Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166. IEEE, April 2014.

42 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013.

Industrial Application of a Partitioning Scheduler to Support Mixed Criticality Systems

Stephen Law
Rolls Royce Control Systems, Birmingham, UK
forename.surname@Rolls-Royce.com

Iain Bate
The University of York, York, UK
forename.surname@york.ac.uk

Benjamin Lesage
Rolls Royce Control Systems, Birmingham, UK
The University of York, York, UK
forename.surname@york.ac.uk

Abstract

The ever-growing complexity of safety-critical control systems continues to require evolution in control system design, architecture and implementation. At the same time the cost of developing such systems must be controlled and importantly quality must be maintained.

This paper examines the application of Mixed Criticality System (MCS) research to a DAL-A aircraft engine Full Authority Digital Engine Control (FADEC) system which includes studying porting the control system's software to a preemptive scheduler from a non-preemptive scheduler. The paper deals with three key challenges as part of the technology transitions. Firstly, how to provide an equivalent level of fault isolation to ARINC 653 without the restriction of strict temporal slicing between criticality levels. Secondly extending the current analysis for Adaptive Mixed Criticality (AMC) scheduling to include the overheads of the system. Finally the development of clustering algorithms that automatically group tasks into larger *super-tasks* to both reduce overheads whilst ensuring the timing requirements, including the important task transaction requirements, are met.

2012 ACM Subject Classification Computer systems organization → Real-time operating systems; Software and its engineering → Real-time schedulability; Hardware → Safety critical systems

Keywords and phrases MCS, DO-178C, Real-Time

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.8

1 Introduction

Real time embedded software tasks developed for safety critical systems, such as civil aircraft engine controls, are typically developed according to a specific Development Assurance Level (DAL) [28]. The DAL indicates a criticality level for a component and is assigned based on the consequence to the system's safety that a failure of this component could cause. This paper shall consider the model presented in [28], that defines DAL-A as the highest criticality level and DAL-E the lowest. It is typically assumed that the amount of effort assigned to producing enough evidence to prove the correct operation of a software component is directly proportional to its DAL [32]. However, in practice it is still essential that lower DAL software operates as expected to deliver the desired customer capability. Put simply a task's criticality is not necessarily related to its "importance".

In accordance with DO-178C [28], where two components developed against different DALs are integrated in the same system, it is necessary to guarantee that high DAL components have temporal and spatial isolation from "unproven" low DAL components.

There are two forms of partitioning that must be employed [29]: temporal partitioning, which is concerned with the response time of a component, and spatial partitioning, which is concerned with the hardware and memory space of a component. This paper is principally concerned with the former, however in order to develop a safety critical system it is vital to consider both together.

In the literature a Mixed Criticality System (MCS) is a system which combines software of multiple DALs on the same processor. The technical objective of which is to provide sufficient evidence that a low DAL component cannot jeopardise any high DAL component's temporal or functional requirements, while still providing a level of service to the low DAL component. One approach to MCS development is to deploy the partitioned architecture defined by the ARINC 653 standard [1]. The standard defines a partitioned model principally aimed at the development of Integrated Modular Avionics (IMA), but is capable of supporting partitions developed against different DALs. The issue with the ARINC 653 approach is that the solution defined for temporal partitioning, essentially a two-level scheduler with time division, makes the approach difficult to apply to a complex control system [3]. This is because it can lead to the introduction of higher completion jitter, longer end-to-end transaction response times and in general it can be difficult to accommodate a complex task schedule into fixed time partitions [3].

Instead, this paper examines the potential for applying a pre-emptive MCS scheduling approach, available in the literature, to a real industrial safety critical avionics application. The application chosen for prototyping was a Rolls-Royce DAL-A avionics engine control system. This system was taken off a certified in-service application without modification or simplification. The primary aim was to investigate whether a set of less critical monitoring functionality could be executed on the same processor as the DAL-A control software whilst still meeting its certification requirements. In order to support these requirements a move to a preemptive system was required in order to allow the introduction of temporal partitioning protection. Therefore a secondary aim was to study the migration of the existing non-preemptive fixed priority scheduling approach [8] to using preemptive scheduling. This offered further advantages such as responsiveness, avoidance of high blocking terms, however at the expense of greater RTOS overheads and complexity.

This work considers AMC+, the Adaptive Mixed Criticality (AMC) [5] variant, which uses a simple mode change protocol and recovery point on an idle tick [31]. AMC+ is chosen as it is the simplest fixed-priority scheduling approach for MCS that allows the system to return to low mode after a switch to high-criticality mode.

The paper describes the challenges involved in certifying a real industrial system with AMC+ and as part of doing so three main contributions are made. The first contribution, in section 3, is a process that explains how the scheduler can be constructed that when combined with timing watchdogs allows AMC+ to be efficiently implemented whilst giving the necessary temporal isolation between high and low criticality tasks. In section 4, given a suitable RTOS and watchdog mechanisms, the standard AMC+ schedulability analysis is extended to account for the real overheads of the Rolls-Royce aircraft engine control system, referred to in this paper as a Full-Authority Digital Engine Controller (FADEC). The extensions form the second contribution of the paper.

The main contributions of the paper, in section 5, are: demonstrating how real world requirements of task dependencies and jitter requirements can be handled; and a task clustering mechanism based around the tasks' deadlines that mitigates the overheads of the system such that more individual tasks are schedulable. This includes adherence to tight jitter requirements placed on certain tasks, as well as transaction requirements involving multiple tasks. Even though the work is targeting the control system, the benefits of the proposed clustering algorithms are also demonstrated on a larger-scale evaluation.

Sections 3, 4 and 5 are preceded by a Related Work section and followed by the conclusions.

2 Background

The purpose of this section is to consider related work to this paper, explain the standard system model for AMC+, the standard schedulability analysis for AMC+, and finally the existing Rolls-Royce control system's approach to scheduling.

2.1 Related Work

This section considers the related work on two topics: reducing the cost of RTOS overheads and MCS. The work on reducing the cost of overheads has two main approaches: firstly making the analysis less pessimistic and secondly reducing the actual overheads. In terms of the analysis pessimism, the majority of work has been in the area of Cache-Related Preemption Delays (CRPD) where an understanding is derived of the impact of on-the-cache contents and which parts of the software cannot preempt each other [24]. The focus in this work is not reducing the pessimism of the analysis, although such approaches could be applied as part of future work, but instead reducing the size of the RTOS overheads.

Overlooking the obvious aim that any RTOS or scheduler must be designed to be as efficient as possible; two main areas of research have been performed on reducing the occurrences of overheads. Firstly work has looked at minimising the number of priority levels, e.g. [2], which can lead to a reduction in the number of task context switches. The second approach is grouping a number of software components (tasks in the original Rolls-Royce control software) into larger schedulable tasks (referred to here as super-tasks). This approach is the same philosophy as adopted in AUTOSAR systems where runnables are grouped to form tasks as part of reducing overheads [11, 16, 26]. The approach however suffers from restrictions that prevent their application in the current case; they either ignore transactions between tasks [11], or require the possibility to duplicate tasks shared between transactions [16]. This paper identifies, for the system studied, that the most appropriate method was the latter of the two mentioned here, that is reducing the number of tasks scheduled by the RTOS. This paper extends this research by examining task sets incorporating task transactions and jitter requirements.

Vestal [32] was one of the first publications to consider the schedulability of a MCS. The work draws the comparison that the reliability of the Worst Case Execution Time (WCET) figure used for each task is commensurate to its criticality. This is based on the observation that lower DAL tasks are not developed, or verified, to the same rigour that higher DAL tasks are, and therefore the output WCET figures cannot be expected to be as reliable.

Building off Vestal's work, Baruah et al. [5] introduced three models for Mixed Criticality Scheduling - partitioned criticality, Static Mixed Criticality (SMC) and Adaptive Mixed Criticality (AMC).

Partitioned criticality [5] is the simplest form of mixed criticality scheduling, where priorities are assigned according to each task's criticality. Accordingly a task with a higher criticality will always be scheduled with a higher priority than another task of lower criticality. This approach ensures a timing error in a low DAL task cannot affect the temporal requirements of a high DAL task, therefore requiring no run time monitoring. However, because the scheduler will always execute a high DAL task if one is ready, it makes it significantly

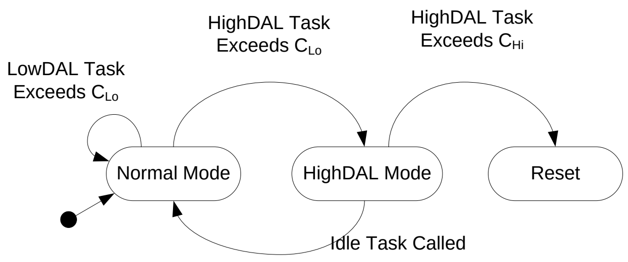


Figure 1 AMC+ State Flow Diagram.

more difficult to meet low DAL task deadlines [5]. Furthermore as Baruah et al. [5] point out, the argument for avoiding run time monitoring is voided by the fact that many safety critical systems already incorporate run time monitoring for the purpose of error detection.

SMC and AMC [5] on the other hand assign task priorities according to their temporal requirements, regardless of criticality. The algorithms following Vestal's work assume that each task has a WCET bound (C_i^L) for each criticality level ($L = \{A, B, C, D, E\}$) where $C_i^A \geq C_i^B \geq C_i^C \geq C_i^D \geq C_i^E$. In most cases, previous works consider only two WCET bounds, C_i^{HI} and C_i^{LO}, where $C_i^{HI} \geq C_i^{LO}$. In this paper, DAL-A tasks have two execution times, C_i^{HI} and C_i^{LO}, and DAL-C tasks have just C_i^{LO}. Extensive software testing in representative engine operation simulations provide confidence that C_i^{LO} *should* not be exceeded, however this cannot be proved, and so C_i^{HI}, produced through WCET analysis, provides a safe upper bound.

SMC allows low or high DAL tasks to execute up to their C_i^{LO} or C_i^{HI} respectively; but they are then prevented from executing further [4]. This offers a stop dead point where any task must cease executing and provides adequate protection for high DAL tasks from low DAL tasks.

The AMC protocol builds off this; however whereas SMC de-schedules one task if it executes for longer than C_i^L, AMC de-schedules *all* low DAL tasks if *any* high DAL task executes for longer than its C_i^{LO}. While the original paper did not explicitly define a recovery point, an obvious route back to re-enabling low DAL tasks is to use the Idle Task or state of the system. This is referred to in this paper as AMC+ and is based around the simple mode change protocol in [31]. The AMC+ protocol is achieved through a scheduler mode change which is summarised in Figure 1.

There are a great number of refinements to the AMC+ model, e.g. the Bailout protocol in [9, 10], however these are largely independent to the contributions of this paper. That is, the papers lacked details of how the implementation would ensure the properties needed for certification, the analysis did not consider overheads as part of the analysis, and the papers did not consider how overheads could be reduced. Therefore, they are not covered here.

A number of papers have considered system development of an MCS. Sousa et al [30] identify the overheads induced by a multi-core task-split mixed criticality system, a number of the overheads identified and integrated into the schedulability analysis are similar to the overheads identified in this paper. However, in this paper we go further by presenting a full process for how overheads can be analysed and reduced.

Freitag et al [17] divides tasks of different criticalities across different cores on a multi-core processor, in order to simplify system proof, the system supervisor analyses the interference induced by low criticality cores on high criticality cores, disabling the low criticality core if required. Herman et al [19] perform an analysis of the development of a mixed criticality multi-core system, however the initial analysis does not progress far enough to support actual development, for instance through proof of the effect of overheads on the schedulability of the system. Finally, Paolillo et al [27] examine the benefits of porting an industrial case study to a mixed criticality system, finding that the potential low criticality task utilisation is high, but also identifying how the identification of sound task WCETs had a significant effect on the service afforded to low criticality tasks. The paper however did not progress far enough to explore how such a system could be implemented and certified.

The focus of this paper is on applying an industrial avionics control system against the mixed criticality model originally presented by Vestal [32] as this provides a model well suited to industrial application. The specific implementation chosen is the AMC+ algorithm introduced by Baruah et al. [5]. The following sub-sections present more formally the baseline system model and schedulability analysis of AMC+.

2.2 System Model

A system is defined as a collection of tasks denoted by τ_i where $1 \leq i \leq N$. Each task τ_i is denoted by a **deadline** D_i, a **period** T_i, a **criticality level** L_i, and one or many **WCETs** C_i. A task is said to have a hard deadline ($D_i \leq T_i$) if its execution must meet every deadline, whereas a soft deadline allows deadlines to be missed without having an adverse impact on the safe operation of the component.

Other parameters which describe a task include the release **jitter** J_i which denotes the maximum permissible variation of the period T_i for the release of the task. Once a task has been scheduled it may be assigned a **priority** P_i where $1 \leq P_i \leq N$. It is possible for the execution of one task τ_i to be reliant on the completion of another task τ_j, such an interaction is described as a **transaction**. Transactions are formed in order to aid the proof of system level timing requirements, where it may need to be proven that the system performs a set sequence of activities in order, and within a set period of time.

Finally, a task τ_i is said to be schedulable if its **worst case response time (WCRT)** R_i, is less than its deadline D_i.

2.3 Static Schedulability Analysis

The static schedulability analysis assessed in this implementation follows the AMC-rtb algorithm introduced by Baruah et al. [5]. The schedulability analysis is performed in three stages. First, the response time of each task is assessed in the high and low modes. Finally, the response times of the high criticality tasks are assessed during a mode change from low to high.

The low and steady high mode response time analysis equations are defined below, where $hp(i)$ is the set of higher priority tasks than task τ_i and $hpH(i)$ is the set of high DAL higher priority tasks than task τ_i. Respectively $hpL(i)$ for the set of low DAL higher priority tasks. These equations calculate the high-criticality mode WCRT, R_i^{HI}, and the low-criticality mode WCRT, R_i^{LO}.

$$R_i^{LO} = C_i^{LO} + \sum_{j \in hp(i)} \left(\left\lceil \frac{R_i^{LO}}{T_j} \right\rceil C_j^{LO} \right) \tag{1}$$

$$R_i^{HI} = C_i^{HI} + \sum_{j \in hpH(i)} \left(\left\lceil \frac{R_i^{HI}}{T_j} \right\rceil C_j^{HI} \right) \tag{2}$$

Where each equation should be recursively solved. Low criticality tasks are not considered when in the high-criticality mode as they are de-scheduled by the system. The sufficient mode change analysis [5] then defines the response time analysis for a high DAL task during a low-to-high mode change as follows:

$$R_i^* = C_i^{HI} + \sum_{j \in hpH(i)} \left(\left\lceil \frac{R_i^*}{T_j} \right\rceil C_j^{HI} \right) + \sum_{k \in hpL(i)} \left(\left\lceil \frac{R_i^{LO}}{T_k} \right\rceil C_k^{LO} \right) \tag{3}$$

This ensures the interference from low DAL tasks is capped as R_i^* must be greater than R_i^{LO}.

2.4 Existing Control System

"Visual Fixed Priority Scheduler" (VisualFPS) is a task attribute assignment and scheduling analysis tool framework developed initially by Bate and Burns [8] and then used by Rolls-Royce on all their FADECs since 2002 [15].

The current FADEC approach features a non-preemptive scheduler where all tasks are released by a clock tick which has a period equal to the greatest common divisor of the tasks' periods [8]. Timing protection is provided by a hardware timing watchdog that counts down from the clock tick period. If the counter is not reset before it reaches zero then the processor is reset, re-initialising the system. When combined with a dual lane architecture each with independent power supplies, sensors and actuators, the use of a hardware timing watchdog ensures the likelihood of a processor or software fault leading to a hazardous safety event is acceptably low.

The current system is controlled using a non-premptive scheduler supported by a hardware timing watchdog. It consists of over 200 tasks, designed against an extensive set of system timing requirements [8]. Certification is achieved using timing information from a measurement-based timing analysis process [23] [22] [25].

The timing requirements for the task set include independent task requirements of period T_i, deadline D_i and in some cases completion jitter J_i as well as dependent task transaction requirements. The transaction requirements are further complicated by two factors. Firstly some tasks appear in more than one transaction, and secondly within a transaction it may be the case that some tasks have different periods. For example a transaction may run a sequence of tasks with periods of 25, 50, 50, 25, 100 and then 25 respectively. An important decision taken is to use a repeatable algorithm (i.e. that always produces the same results) that takes all the requirements and uses them to calculate the deadline for each task. Task priorities are then assigned using the Deadline Monotonic Priority Ordering (DMPO) algorithm where the task with the shortest deadline is given the highest priority. If all deadlines are met, all the timing requirements are met; the method ensures the schedule is correct by construction. This approach has a further advantage, key to industry, that by incorporating the timing requirements for each task into its design-time calculated deadline; the system can be easily proved, reviewed and understood by engineers and system integrators [8].

An aim of this work is to change the processing platform, scheduling mechanisms, and tooling by only the minimum amount necessary, as the tooling is well understood and accepted by engineers and certification authorities respectively.

Each task is designed to communicate with a common interface, and follows a format of input-process-output, furthermore tasks are designed to execute upon the data that is currently available and will not wait until fresh data is available. Where fresh data is required to move between tasks this is generally controlled by a transaction. This approach has the advantage of simplifying access to shared resources between tasks, and is therefore not considered further as part of this paper.

3 Achieving Sufficient Temporal Isolation

This section introduces the requirements surrounding the development of a MCS, both from a certification and a system integration point of view.

3.1 Certification Requirements

The paper focuses on the development of software for avionics systems. Accordingly, only the guidelines detailed in DO-178C [20] are explored in this section. However the guidelines are considered similar to those detailed in other software domains such as ISO26262 and IEC61508 [18].

DO-178C Section 2.4 defines five requirements for partitioning as follows:

1. A partitioned software component should not be allowed to contaminate another partitioned software component's code, input/output (I/O), or data storage areas
2. A partitioned software component should be allowed to consume shared processor resources only during its scheduled period of execution
3. Failures of hardware unique to a partitioned software component should not cause adverse effects on other partitioned software components
4. Any software providing partitioning should have the same or higher software level as the highest level assigned to any of the partitioned software components
5. Any hardware providing partitioning should be assessed by the system safety assessment process to ensure that it does not adversely affect safety

These requirements form the basis for this mixed criticality assessment and they are explored further in the following sections.

3.2 Target Processor

The target processor for this analysis is the Rolls-Royce in-house processor. The Rolls-Royce processor is a packaged device that integrates a core, memory, IO and tracepoint interfaces. Being targeted at the safety-critical embedded sector, the device is DO-254 – Level A compliant. It has extensive single-event-upset protection and is suitable for harsh environments. The processor does not incorporate a data or instruction cache due to their impact on timing predictability.

The processor has been carefully designed to ensure that each instruction's execution is time-invariant; in other words each instruction will take the same time to execute, regardless of the data its operation is performed upon. These design features further ensure that previous processor state has no effect on the current operation of the device.

The use of such a deterministic processor allows worst case timing measurements of software components, including the scheduler, to be taken during normal operation, without the need for special builds [22,23]. These measurements are used in two ways. For High-DAL tasks the measurements are input into a hybrid measurement based WCET analysis tool and used for the task's C_i^{HI}. Secondly for both High-DAL and Low-DAL tasks the Maximum Observed Execution Time (or High Water Mark - HWM) obtained during the extensive software test regime is used as the task's C_i^{LO}.

Together with supervisor and user mode spatial partitioning control, this determinism provides the basis for an argument to be made that different software partitions executing upon the Rolls-Royce processor cannot affect each other from a spatial or temporal point of view.

3.3 Partitioning

The aim of this assessment was to identify how a set of low DAL monitoring tasks can be integrated alongside a set of high DAL-Control tasks. The assumption that should be made for a low criticality task is that the task may execute, if allowed, for longer than its observed HWM; its C_i^{LO} is assumed to be optimistic. However, due to the rigorous testing regime the software undergoes and the extensive in-test and in-flight monitoring it is known that it is rarely exceeded.

From a certification point of view as the evidence to prove otherwise may not have been produced to the same level as a high DAL component, then a certification authority must assume a low DAL component is more likely to contain an error. So as guided by the requirements noted in Section 3.1, partitioning must be employed to prove that any errors that occur in a low DAL partition, cannot propagate to a high DAL partition. Partitioning is thus employed between high and low DAL tasks.

The scheduler proposed in this work implements two key protection mechanisms to implement a DO-178C partitioned architecture: the use of timer driven interrupts, and the use of processor memory protection. Figure 2 shows the statechart for the interrupt driven scheduler.

A timer driven interrupt is employed both to control the release of new tasks by invoking a scheduler tick, and to interrupt low DAL components when they reach their C_i^{LO}. As the interrupt handler prepares to switch in a task, one of the final operations is to set the interrupt timer to the lowest of either 1) the time to the next task release, or 2) in the case of a low DAL task the allowed execution time remaining.

High DAL tasks are not regulated in the same way. If a high DAL task executes beyond its C_i^{LO} then it is permitted to continue, but the next time the scheduler executes it will identify the need to move into the high DAL mode. This is controlled by the "Handle Overrun" block within Figure 2. Return to the low mode is controlled by a high DAL idle task.

Secondly, processor memory protection is employed in a User/Supervisor arrangement. All tasks execute in a design-defined protected area of memory, with access to different hardware features or memory regions either permitted or restricted as necessary. Should any task execute outside these fixed boundaries, then an interrupt is raised and the interrupt handler handles the data error. This is enforced by the scheduler setting the proper User Mode when returning to a task as illustrated in Figure 2.

The memory protection employed ensures that each component executing on the processor cannot execute outside its design time defined boundaries, thus providing protection for high DAL tasks from low DAL task promiscuous memory or hardware interactions. Whereas the timer driven interrupt restricts the execution of low DAL tasks, ensuring a low DAL task cannot execute beyond its C_i^{LO}.

The processor is further protected by the *accepted and proven in use* hardware timing watchdog, which ensures any timing requirement not met is detected sufficiently quickly. Finally, at a system level the control system is designed around a full dual redundant architecture.

The earlier certification requirements are now re-visited with the reasons they are met in italics.

1. A partitioned software component should not be allowed to contaminate another partitioned software component's code, input/output (I/O), or data storage areas
   The use of appropriately written and tested software makes this less likely, in addition the control system processor provides memory and other hardware protections through the use of a supervisor/user mode configuration. Furthermore, the processor is built upon a deterministic platform.

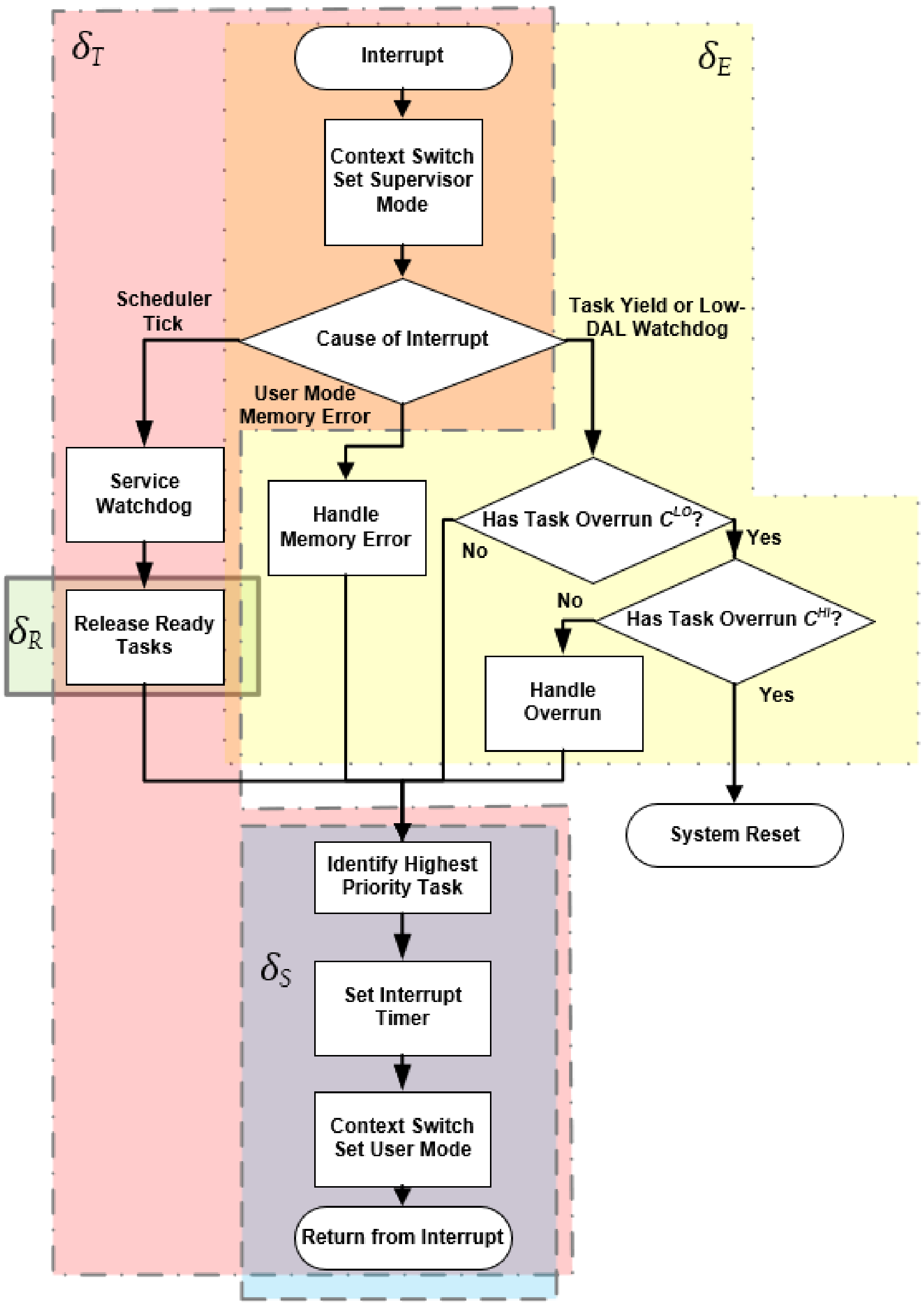


Figure 2 Partitioned Scheduler Statechart.

2. A partitioned software component should be allowed to consume shared processor resources only during its scheduled period of execution
   The software timing interrupt, controlled by the scheduler, should prevent this by bounding the execution time of low criticality tasks according to the task's C_{LO}. High criticality tasks are not interrupted except by a scheduler tick, this is based on the trusted WCET analysis process followed for high criticality tasks.
3. Failures of hardware unique to a partitioned software component should not cause adverse effects on other partitioned software components
   If a failure prevents the software timing interrupt providing the expected protection then the hardware timing watchdog, which is accepted and proven in use, combined with a two-lane (duplex) architecture will ensure acceptable safety. Furthermore, the spatial partitioning employed shall ensure a task cannot interact with address regions outside of its permitted bounds
4. Any software providing partitioning should have the same or higher software level as the highest level assigned to any of the partitioned software components
   The RTOS, interrupt handler, scheduler and software timing watchdog should all be developed to the highest DAL and are executed as protected "supervisor" mode components.
5. Any hardware providing partitioning should be assessed by the system safety assessment process to ensure that it does not adversely affect safety
   The processor timing supervision component has been verified to DO-254 DAL-A and has been used on multiple certified systems.

4 Extending AMC+ Analysis to Allow for Overheads

In order to include the execution time of the scheduler shown in Figure 2 into the response time analysis for the system, the overheads were broken down into three constituent parts as described below:

1. Tick Overhead - δ_T (Figure 2 - dot/dash line, red). Calculated using the measured worst case tick time C_{TICK}. It includes:
   - The pre-emption of the executing task.
   - The handling of system services, e.g. the watchdog.
   - The context switch, and calling, of the highest priority task.
   - The release of any tasks into the ready state. Measured separately as C_{REL}. (Figure 2 - solid line, green)
2. Start Task Time - δ_S (Figure 2 - dashed line, blue). Calculated using the measured worst case task switch in time C_{START}. It includes:
   - The initial time taken to context switch each task into the executing state. Except for the highest priority task, which is accounted for in the tick overhead.
3. Stop Task Time - δ_E (Figure 2 - dotted line, yellow). Calculated using the measured worst case task switch in time C_{END}.
   - The end time taken when a task finishes executing and returns to the scheduler.

Task releases are fixed to only occur on a scheduler tick, and the scheduler tick is the only component that can interrupt another task. The execution time of each overhead was measured during normal system operation, which included at certain points, the schedule's critical instance. This ensures the maximum execution time for each overhead was captured firstly by ensuring the maximum number of tasks were in the released state at certain points, and that the maximum number of preemption points are observed.

The release overhead was measured and recorded against the number of tasks being released. This allowed the release overhead of each task to be assessed, which proved to be linear against the number of tasks being released as per the design goal.

The credibility of this maximum observed overhead time is based on the following implementation details:

- The use of a time deterministic target processor.
- Tasks are only released on the system tick. The system tick period is equal to the greatest common divisor of the tasks' period. All other task periods in the system are a harmonic of the tick period.
- Each overhead is measured while the system executes a full system test campaign on a full simulation rig.
- The RTOS is carefully designed to ensure the task release overhead is linearly proportional to the number of tasks in the system.

Finally, each overhead was factored into the analysis through synthetic tasks, in the same way originally introduced by Burns et al [13]. This method of essentially viewing certain overheads as tasks provides a safe and suitable method for taking account of the periodicity of the overheads, and allows the overheads to be placed at the appropriate place in the schedule to ensure correct analysis of interference.

The effect that the tick overhead has on the response time of a task can then be calculated as follows:

$$\delta_T^{MODE} = \left\lceil \frac{R_i^{MODE}}{T_{TICK}} \right\rceil C_{TICK} + \sum_{j \in MODE(i)} \left(\left\lceil \frac{R_i^{MODE}}{T_j} \right\rceil C_{REL} \right) \tag{4}$$

In Equation 4, as in the following Equations 5 and 6 the value of R_i^{MODE} used should either be R_i^{LO}, R_i^{HI} or R_i^* depending on whether the low mode, high mode or mode change response time is being calculated.

Secondly, the set of higher priority tasks used in each equation (denoted as $j \in MODE(i)$ or $j \in hpMODE(i)$) should be limited to those tasks permitted to execute in order to avoid undue pessimism. The scheduler tick occurs in all scheduler modes, as well as during a mode change. However the release overhead for low DAL tasks will only occur in the low mode, or during a mode change from low criticality to high. This low DAL task release overhead cannot be ignored during a mode change because the release of these tasks occurs before the highest priority task begins to execute.

The start and stop overheads of each task are calculated as follows.

$$\delta_S^{MODE} = \sum_{j \in hpMODE(i)} \left(\left\lceil \frac{R_i^{MODE}}{T_j} \right\rceil C_{START} \right) \tag{5}$$

$$\delta_E^{MODE} = \sum_{j \in hpMODE(i)} \left(\left\lceil \frac{R_i^{MODE}}{T_j} \right\rceil C_{END} \right) \tag{6}$$

Equations 1 and 2 can therefore be extended as follows:

$$R_i^{LO} = C_i^{LO} + C_{START} + \delta_T^{LO} + \sum_{j \in hp(i)} \left(\left\lceil \frac{R_i^{LO}}{T_j} \right\rceil C_j^{LO} \right) + \delta_S^{LO} + \delta_E^{LO} \tag{7}$$

$$R_i^{HI} = C_i^{HI} + C_{START} + \delta_T^{HI} + \sum_{j \in hpH(i)} \left(\left\lceil \frac{R_i^{HI}}{T_j} \right\rceil C_j^{HI} \right) + \delta_S^{HI} + \delta_E^{HI} \tag{8}$$

Finally Equation 3 can be extended as follows:

$$R_i^* = C_i^{HI} + C_{START} + \delta_T^* + \sum_{j \in hpH(i)} \left(\left\lceil \frac{R_i^*}{T_j} \right\rceil C_j^{HI} \right) + \sum_{k \in hpL(i)} \left(\left\lceil \frac{R_i^{LO}}{T_k} \right\rceil C_k^{LO} \right) + \delta_S^* + \delta_E^* \quad (9)$$

This overhead model is built on the assumption that a switch from a low to a high DAL task, and the associated context switch, takes a constant time. On the deterministic Rolls-Royce aerospace company processor this is the case. However for less deterministic processors, or more complex context or thread switching mechanisms this model, and associated priority assignment mechanism, it may need to be adapted in a similar way to the work presented by Davis et al. [14].

This section has introduced an overhead model for calculating the response time of tasks in a preemptive mixed criticality system. The following section will now examine how these overheads can be minimised through appropriate system design.

5 Clustering to Reduce the Overheads of a Pre-Emptive Mixed Criticality System

The current Rolls-Royce control system architecture consists of a large number of tasks, carefully designed to reduce the effects of task blocking in the current non pre-emptive scheduler (§ 2.4). The system follows a correct by construction design approach; each task‘s deadline is calculated as part of the design process to ensure that if all tasks meet their deadline (as confirmed through response time analysis), then the system as a whole will meet its temporal requirements. This relies on a task attribution assignment which calculates each task’s deadline based on its period, transaction and completion jitter requirements. This system can then be said to comply with its timing requirements provided all tasks meet their deadlines.

The overhead assessment and implementation rules defined in Section 4 illustrated how the approach of using a large number of individual scheduled tasks is less desirable for a pre-emptive model. This is because firstly the RTOS overheads of the pre-emptive RTOS are significantly higher than the overheads of the non-pre-emptive system due to the introduction of context switching and task monitoring, but also because the overheads increase with the number of tasks, and their associated releases, in the system. By reducing the number of tasks called from the RTOS, the number of tasks N is reduced to $N_{CLUSTER}$ (referring to Equation 4) reducing the overhead induced by task releases. Furthermore, the number of higher priority tasks $hpMODE(i)$ is reduced in Equations 5 and 6, reducing the start and stop task induced overheads.

This meant that simply porting the existing control system task set to the new architecture produced a system whose overheads exceeded 40% of the total system utilisation, making the system unschedulable. Therefore, it is necessary to condense the set of tasks in order to reduce the RTOS overheads to a level where a schedulable system can be defined.

There are two aspects that must be considered when porting components from one architecture to another. The first is the correct handling, and protection, of data transfers that are conducted across the system. The second is in the correct allocation of tasks to fulfil the temporal requirements of the compiled system. This paper is concerned principally with the latter, with the former being considered in parallel work.

A number of methods were investigated for controlling the clustering, all methods were defined using some or all of the temporal requirements placed on the tasks within a system. Each method follows the same basic process of placing tasks into a defined order, where

Algorithm 1 Task Clustering Algorithm.

```
/* Calculate task deadlines based on each task's timing requirements */
UnOrderedTasks = CalculateTaskDeadlines(Period, Jitter, Transactions)
/* Create OrderedTaskSet according to the defined clustering method */
switch (ClusteringMethod) is
 case Period:
   /* Order tasks based on their period, lowest period taking the highest priority */
   OrderedTaskSet = OrderByPeriod(UnOrderedTasks)
 case Transaction:
   /* Order tasks based on their transaction requirements. Transactions with the lowest overall
   deadline take the highest priority */
   OrderedTaskSet = OrderByTransaction(UnOrderedTasks[Task $\in$ Transactions])
   /* Order tasks without transaction requirements based on their jitter requirements first, and
   finally any remaining tasks by their period. */
   OrderedTaskSet += OrderByJitter(UnOrderedTasks[Task $\notin$ OrderedTaskSet])
   OrderedTaskSet += OrderByPeriod(UnOrderedTasks[Task $\notin$ OrderedTaskSet])
 case Jitter:
   /* Order tasks based on their Jitter requirements. Prioritising tasks with small Jitter require-
   ments */
   OrderedTaskSet = OrderByJitter(UnOrderedTasks)
   OrderedTaskSet += OrderByTransaction([(Task $\notin$ OrderedTaskSet) $\in$ Transaction])
   OrderedTaskSet += OrderByPeriod([Task $\notin$ OrderedTaskSet])
 case Deadline:
   /* Order tasks based on their Deadline, shortest Deadline taking the highest priority */
   OrderedTaskSet = OrderByDeadline(UnOrderedTasks)

Move OrderedTaskSet[0] into SuperTask[0]
SuperTask[0].Period = OrderedTaskSet[0].Period
SuperTask[0].Criticality = OrderedTaskSet[0].Criticality

j = 0
for i in 1..OrderedTaskSet.Length do
   if OrderedTaskSet[i].Period is NOT a harmonic of OrderedTaskSet[i-1].Period
   or OrderedTaskSet[i].Criticality $\neq$ SuperTask[j].Criticality then
      j++
      Move OrderedTaskSet[i] into SuperTask[j]
      SuperTask[j].Period = OrderedTaskSet[i].Period
      SuperTask[j].Deadline = OrderedTaskSet[i].Deadline
      SuperTask[j].Criticality = OrderedTaskSet[i].Criticality
   else
      Move OrderedTaskSet[i] into SuperTask[j]
      SuperTask[j].Deadline = min(OrderedTaskSet[i].Deadline, SuperTask[j].Deadline)
      SuperTask[j].Period =
      GreatestCommonDivisor(OrderedTaskSet[i].Period, SuperTask[j].Period)
   end if
end for

Apply DMPO to SuperTask set
```

Algorithm 2 Extension to Support Deadline_D.

```
29: if Task[i].Period is NOT a harmonic of Task[i-1].Period
    or Task[i].Criticality ≠ SuperTask[j].Criticality
    or Task[i].Deadline ≠ SuperTask[j].Deadline
```

nominally the first task would form the highest priority, and the last task the lowest. The next step is to step through the ordered set breaking it down into super-tasks. The methods studied are introduced in Algorithm 1 and described below:

- No Clustering - no clustering is performed
- Period - Tasks were ordered according to period, then one super-task was formed for each different task period
- Jitter - Tasks with completion jitter requirements were first organised into super-tasks, before all remaining tasks were ordered and grouped using their transaction requirements (if applicable) and finally period.
- Transactions (Trans) - Tasks with cross-task transactional requirements were first placed into super-tasks along with the other tasks in their transaction. The remaining tasks were then sorted into super-tasks first using their jitter requirements (if applicable) and finally periods.
- Deadline_D (D_D) - Tasks were first ordered using their computed deadline, then the ordered set was divided into super-tasks across task period boundaries. Finally, tasks with different deadlines are not clustered together.
- Deadline_P (D_P) - As Deadline_D, however tasks with different deadlines were permitted to be clustered together.

In all cases no super-task contains tasks of different criticalities, this is vital to comply with the partitioning approach introduced in section 3.3. Finally, if any two tasks' timing requirements are identical, then their order is decided arbitrarily.

For brevity, Deadline_D is not shown explicitly in Algorithm 1. Unlike Deadline_P, Deadline_D does not cluster together tasks that have different deadlines. Deadline_D is implemented by replacing line 29 of Algorithm 2 with line 29 of Algorithm 1.

As the deadline of a task is derived as described in [7], this in one respect means the Deadline clustering methods are an implicit amalgamation of the Jitter, Transactions and Period clustering methods.

In all cases the deadline for each super-task was set to the lowest deadline of any task inside the super-task. The set of RTOS super-tasks were then prioritised using the DMP Oprotocol [8].

In addition to the above stated clustering methods a search based optimisation algorithm was used to attempt to identify an appropriate task set, however in no cases did it deliver improved results. The reason is the transaction requirements have to be carefully handled, which meant a guided random algorithm did not perform well. Therefore it is not included in this paper.

6 Evaluation & Results

The various approaches to task clustering and their impact on the RTOS overheads were assessed in three ways. Firstly, the clustering methods were each applied to a real Rolls-Royce DAL-A control system, which was then complimented using a set of DAL-C monitoring functions; thus providing a Mixed Criticality System for analysis. Secondly, the clustering

Table 1 Clustering Results When Applied to the Rolls-Royce Aircraft Engine Control System.

	#SuperTasks	Sched Tasks	Trans Pass?	δ_S	δ_E	δ_T	δ_{SUM}
NoClustering	228	24.1%	Yes	17.6%	21.1%	6.3%	45.0%
Period	17	85.5%	No	2.5%	3.0%	2.1%	7.6%
Transaction	10	9.2%	Yes	4.0%	4.8%	2.5%	11.4%
Jitter	53	38.2%	No	13.1%	15.7%	5.1%	33.9%
Deadline_D	167	40.4%	Yes	11.7%	14.1%	4.7%	30.5%
Deadline_P	15	100.0%	Yes	0.8%	0.9%	1.6%	3.3%

algorithms were applied to a publicly available aircraft engine control case study taken from [6]. Finally, a random task set generator was used to produce a large number of tasksets, each of which was clustered using the set of clustering algorithms.

The actual RTOS overhead figures used through the temporal analysis are as measured on the Rolls-Royce control system application, and were obtained during a system test campaign on an aircraft engine control system test rig, and are not discussed further in this paper. However, in order to ensure the analysis was not influenced by these figures, the study also investigated how the proposed methods responded to varying RTOS overheads.

6.1 "Current" Rolls-Royce Engine Control System Example

The Rolls-Royce control software example used for this analysis has already been certified as a DAL-A system. The system consists of a large number of tasks, each of which has a measured HWM and an analysed WCET, obtained using a hybrid-measurement based approach [23]. The HWM and WCET were used for the C_i^{LO} and C_i^{HI} respectively. Additional monitoring tasks were added to the system simulating DAL-C functionality, which increased the number of tasks in total to 228. The results from applying each clustering algorithm to the engine control system are shown in Table 1. The table shows the percentage of tasks whose worst case response time is lower than its deadline (Schedulable Tasks), whether all transactions have maintained the correct order (Transactions Fulfilled), and the utilisation of the RTOS.

Each task has a defined period, approximately 5% of tasks have completion jitter requirements and approximately 50% of tasks form part of a transaction. Each transaction requirement, which consists of between 2 and 11 tasks, reflects a specific execution, or priority, order that must be maintained to ensure compliance to system level timing requirements. These transactions are often interconnected, and can feature tasks with different periods. The task periods are taken from the set (2.5, 5, 10, 12.5, 25, 50, 100, 200, 500)ms.

The results in Table 1 showed that the *Deadline_P* clustering method was the only algorithm able to generate a schedulable system, this was in-spite of the fact that it was not the algorithm that produced the task set with the smallest number of Super Tasks. The *Period* and *Transaction* clustering algorithms failed to prioritise tasks with jitter requirements, and so those tasks presented worst case response times that would have failed to meet their tight timing requirements. Whereas the *Jitter* clustering algorithm failed to correctly order transactions, and created a system with a larger number of high rate super-tasks, leading to a higher RTOS utilisation which left the system unschedulable. The *Deadline_D* method correctly ordered transactional tasks, and prioritised tasks with jitter requirements, however as it did not group together tasks with different deadlines, it created a system with a prohibitively large RTOS overhead.

Table 2 Clustering Results When Applied to an Aircraft Engine Control Case Study.

	#SuperTasks	Sched Tasks	Trans Pass?	δ_S	δ_E	δ_T	δ_{SUM}
NoClustering	71	94.4%	Yes	3.6%	4.3%	2.4%	10.4%
Period	5	53.5%	No	0.3%	0.3%	1.5%	2.1%
Transaction	3	59.2%	Yes	2.2%	2.6%	2.0%	6.8%
Jitter	8	38.0%	Yes	4.4%	5.3%	2.6%	12.3%
Deadline_D	33	97.2%	Yes	2.0%	2.4%	2.0%	6.4%
Deadline_P	2	100.0%	Yes	0.1%	0.1%	1.4%	1.5%

In comparison to the original system, this partitioned approach allowed low DAL tasks totalling 44% utilisation to be added into the system without compromising schedulability across all clustering algorithms. This would not have been possible in the existing legacy system and was only made possible as the analysis was able to capitalise on the difference between each high DAL task's C_i^{LO} and C_i^{HI}.

6.2 Public Domain Engine Control System Example

The second case study used for this analysis has been taken directly from a publicly available aircraft engine control example, as documented in [6]. The task set features 71 individual tasks, conjoined by 24 different transactions. All tasks in the original system were schedulable with all transactions being met.

The results in Table 2 were similar to those produced by the Rolls-Royce case study in section 6.1, and showed that only the *Deadline_P* clustering algorithm was able to produce a schedulable system with all transactions being met. Similarly to the Rolls-Royce case study, the RTOS overhead was considerable lower with clustering than without, but in order to produce a fully schedulable solution it was necessary to prioritise *Transactions* and *Jitter* requirements equally, as performed by the *Deadline_D* and *Deadline_P* algorithms.

6.3 Random Task Set Generation Assessment

The random task set generator is based on a version of the UUniFast algorithm [12], and was extended, as detailed below, to feature jitter requirements and transaction requirements. The random task set generation assessment was performed at varying target utilisations from 30% to 100% (at an interval of every 5%), with a varying number of tasks (10, 50, 100). Each clustering technique was then applied to each generated task set. Finally the result was statically analysed to confirm every task's response time was less than its deadline and that each transaction was correctly ordered. 1000 tests were then performed for each test configuration.

Key characteristics of the real engine control software were identified (and simplified) to constrain the generated tasksets as follows:

- Harmonic periods from the set (2.5, 5, 10, 12.5, 25, 50, 100, 200, 500)ms, inline with the real system used in section 6.1.
- 5% of tasks randomly chosen to contain a jitter requirement. If part of a transaction only a task at the beginning or end of the transaction was given a jitter requirement.
- Transactions consisting of three tasks, randomly chosen from the existing set. The number of transactions in the system was set to one fifth of the number of tasks.
- The C_i^{LO} for each task was randomly defined based on the system level target utilisation. Each task's C_i^{HI} was randomly selected from the range $C_i^{LO} \leq C_i^{HI} \leq 2C_i^{LO}$.

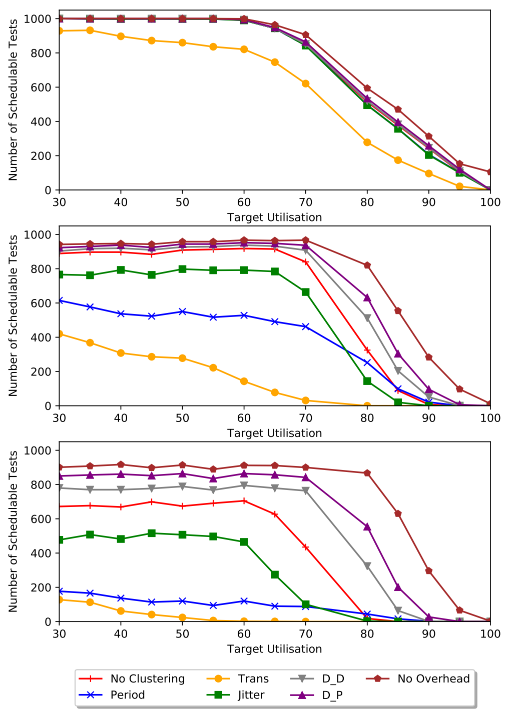


Figure 3 Schedulability of a 10, 50 and 100 Task System at Varying Target Utilisations.

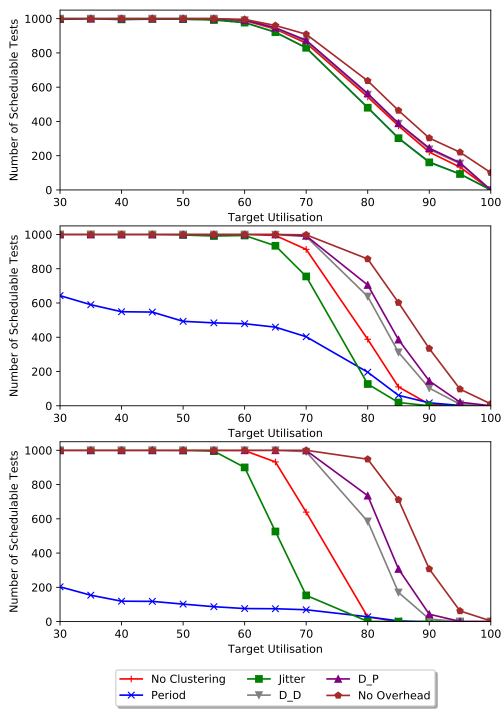


Figure 4 Schedulability of a 10, 50 and 100 Task System With No Transactions.

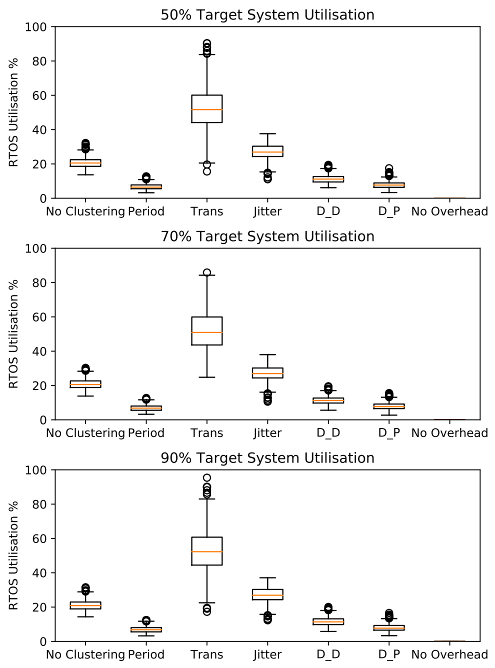


Figure 5 RTOS Overheads Calculated for each Clustered System.

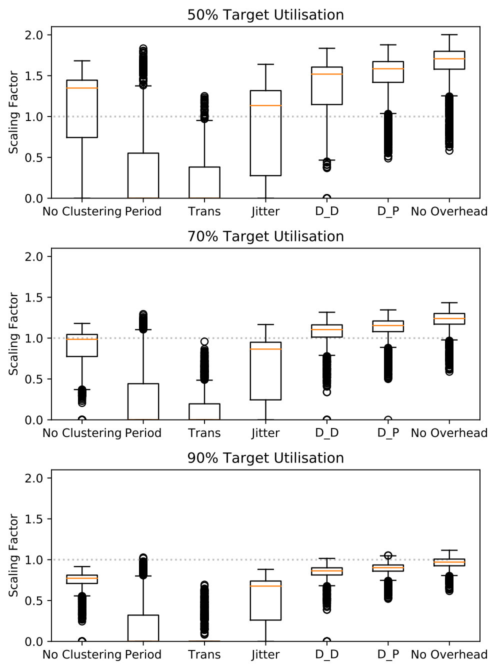


Figure 6 Maximum WCET Scaling Factor to Provide a Schedulable System.

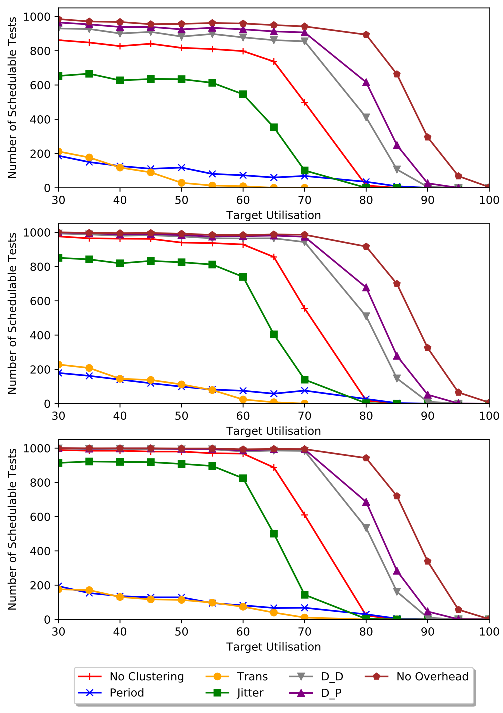


Figure 7 Number of Schedulable Tasks with Varying Transaction Rates [10%, 25% and 50%].

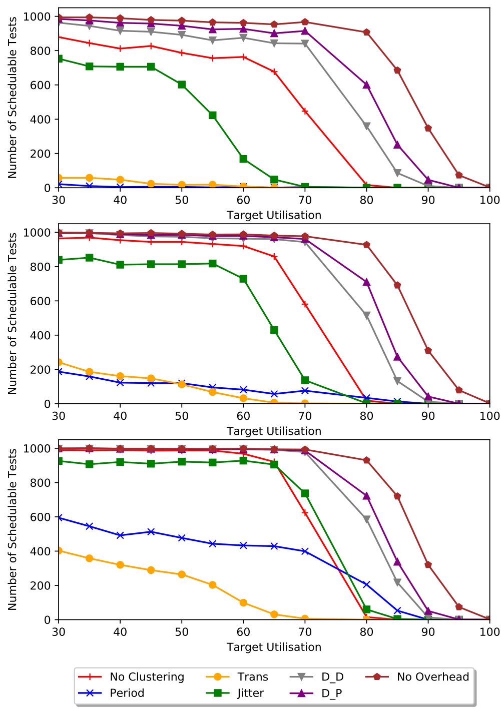


Figure 8 Number of Schedulable Tasks with Varying Jitter Rates of [0%, 5% and 10%].

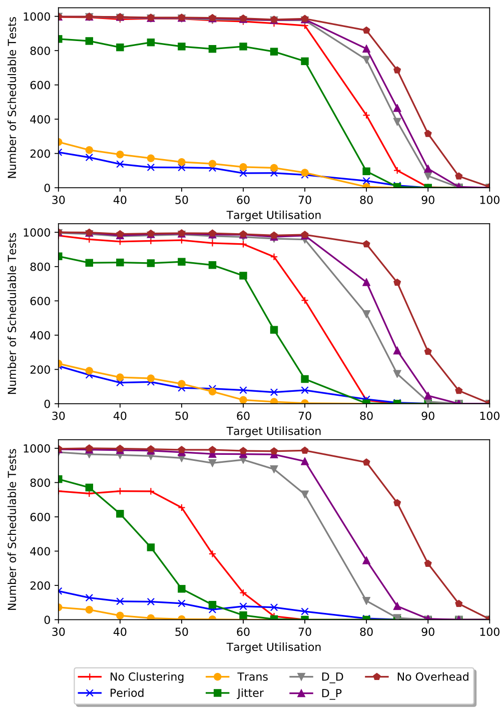


Figure 9 Number of Schedulable Tasks with Low, Medium and High RTOS Overheads.

This follows principles similar to the approach defined in [21] where key characteristics are extracted from a real application and fed into a generator to derive representative benchmarks. However, the tasksets in [21] follow the AUTOSAR runnable model and do not include transactions which have a profound effect on the scheduling approach.

Figure 3 shows the number of schedulable tests out of the 1000 executed for each clustering algorithm at varying target utilisation configurations. The experiments showed that for a small task system there was not a great difference across the different methods, with the exception of the *Transaction* method. From the inspection of the results this was largely because the *Transaction* method fails to take account of tasks with tight jitter requirements, which consequently receive lower priorities and fail their response time analysis.

For a system with 50 tasks, as shown in the second plot of Figure 3, the difference between the clustering methods is more profound. Neither the *Transactions* nor the *Period* methods are able to generate reliably schedulable systems, failing to take account of jitter requirements. The *Jitter* algorithm fares better, but a general failure to preserve transactions causes the schedulability of the solutions to suffer as the task set utilisation grows. The only algorithms able to track near the *No Overhead* ideal are the *Deadline* algorithms, with the *Deadline_P* faring best as it is able to minimise RTOS overheads by producing systems with less super-tasks. This hypothesis is further supported in Figure 5 which shows the RTOS overhead produced by each clustering method.

These results are amplified as the task set size grows to 100 tasks, where again the only algorithm following a similar trend to the *No Overhead* ideal is the *Deadline_P* algorithm. One irregularity with the results is the fact that for 50 and 100 task systems no clustering algorithms are able to achieve a 100% set of schedulable tests. This is because of the effect of transactions as shown by Figure 4 which shows the same test as shown by Figure 3, however without Transactions.

Figure 6 shows, for a 100 task system at varying target utilisation (50%,70%,90%), the analysed maximum possible WCET inflation factor, or sensitivity. That is, the maximum figure that every C_i^{LO} and C_i^{HI} can be multiplied by before the system is no longer schedulable. This figure is determined by sensitivity analysis. Therefore a value above or below one would indicate an increase or decrease (for an initially unschedulable system) in task times respectively. The results showed the *Deadline_P* clustering method maintaining the highest inflation factor across all three target utilisations with other algorithms, in particular *Period* and *Transaction*, tracking inflation factors close to zero. The results further indicate that even *No Clustering* is frequently better than *Jitter*, *Period* and *Transaction*.

Comparing Figure 5 to Figures 3 and 6; even though *Period* tended to have the lowest overhead, it tended to produce less schedulable solutions. This is because the algorithm frequently produces a system with the lowest number of RTOS tasks, however these tasks do not take account of jitter or temporal requirements, and so is in general not schedulable. Either because tasks with jitter requirements have high response times, or because transaction orders are not maintained. This shows that the aim of this clustering operation is not necessarily to simply minimise RTOS overheads.

In order to further review the effectiveness of the different clustering algorithms the analysis was extended through application to different systems with varying transactions rates (Figure 7), varying jitter rates (Figure 8) and varying overheads (Figure 9). This analysis shows how the clustering algorithms performed when presented with different system configurations which moved beyond the assumptions introduced by the avionic control system.

Again the results showed that the *Deadline_P* was reliably the best clustering method, it was shown to be reliable while other clustering algorithms' performance varied significantly across the different system parameters.

7 Conclusion

This paper has considered the application of a MCS scheduler to a DAL-A avionics engine control system. In particular, the paper examined how such a system should be developed and analysed to prove schedulability in the face of real life overheads. The partitioned architecture provides robustness against spatial and temporal infringements by low DAL tasks, and is capable of achieving a greater software utilisation than existing legacy systems.

The paper further considers the temporal requirements surrounding porting of an existing non-preemptive system. The study aimed to identify the most efficient implementation for porting a large system to a new pre-emptive Mixed Criticality System.

The work in this paper has targeted the AMC approach of [31], however the key contributions should be largely independent (or at least applicable to those that build off fixed priority scheduling) of which scheduling approach is used. The reasons are: the architectural approaches for detecting timing overruns and performing mode switches are independent of the policy; the overheads added to the *standard* analysis are only dependent on the architecture; and finally the method for reducing the number of pre-emptions and task releases is also independent of the scheduling approach. It is worth noting the benefits of the clustering algorithm would almost certainly be greater for systems with more complex processors as the preemption overheads would also include Cache-Related Preemption Delays.

For future work this study shall develop the implementation explored in this paper toward a full dynamic study of the operation and performance of the scheduled system. This will include assessing the quality of service provided to low DAL tasks and validating that a promiscuous low DAL task cannot affect the temporal performance of the system.

References

1 Airlines Electronic Engineering Committee. Avionics Application Software Standard Interface Part 1 - Required Services. *ARINC Specification 653 Part 1-3, Aeronautical Radio, Inc.*, 2010.

2 N. Audsley. On priority assignment in fixed priority scheduling. *Information Processing Letters*, 79(1):39–44, 2001.

3 N. Audsley and A. Wellings. Analysing APEX applications. In *17th IEEE International Real-Time Systems Symposium, (RTSS)*, pages 39–44, December 1996.

4 S. Baruah and A. Burns. Implementing mixed criticality systems in Ada. In *International Conference on Reliable Software Technologies*, pages 174–188. Springer, 2011.

5 S. Baruah, A. Burns, and R. Davis. Response-time analysis for mixed criticality systems. In *32nd IEEE International Real-Time Systems Symposium, (RTSS)*. IEEE, 2011.

6 I. Bate. *Scheduling and timing analysis for safety critical real-time systems*. PhD thesis, Citeseer, 1999.

7 I. Bate and A. Burns. An Approach to Task Attribute Assignment for Uniprocessor Systems. In *11th Euromicro Conference on Real-Time Systems*, pages 46–53, 1999.

8 I. Bate and A. Burns. An Integrated Approach to Scheduling in Safety-Critical Embedded Control Systems. *Real-Time Systems Journal*, 25(1):5–37, July 2003.

9 I. Bate, A. Burns, and R. Davis. A bailout protocol for mixed criticality systems. *IEEE Transactions on Software Engineering*, 2015.

10 I. Bate, A. Burns, and R. Davis. An enhanced bailout protocol for mixed criticality embedded software. *IEEE Transactions on Software Engineering*, 43(4):298–320, 2017.

11 A. Bertout, J. Forget, and R. Olejnik. Automated runnable to task mapping. Technical report, HAL, May 2013.

12 E. Bini and G. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

13 A. Burns, K. Tindell, and A. Wellings. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Transactions on Software Engineering*, 21(5):475–480, 1995.

14 R. Davis, S. Altmeyer, and A. Burns. Mixed Criticality Systems with Varying Context Switch Costs. In *Proceedings IEEE Real Time and Embedded Technology and Applications Symposium (RTAS)*, 2018.

15 R. Davis, I. Bate, G. Bernat, I. Broster, A. Burns, A. Colin, S. Hutchesson, and N. Tracey. Transferring real-time systems research into industrial practice: Four impact case studies. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2018.

16 H. Faragardi, B. Lisper, K. Sandström, and T. Nolte. An efficient scheduling of AUTOSAR runnables to minimize communication cost in multi-core systems. In *7th International Symposium on Telecommunications (IST)*, pages 41–48, September 2014.

17 Johannes Freitag, Sascha Uhrig, and Theo Ungerer. Virtual Timing Isolation for Mixed-Criticality Systems. In *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

18 P. Graydon and I. Bate. Safety Assurance Driven Problem Formulation for Mixed-Criticality Scheduling. In *Proceedings of the Workshop on Mixed-Criticality Systems*, pages 19–24, 2013.

19 Jonathan L Herman, Christopher J Kenna, Malcolm S Mollison, James H Anderson, and Daniel M Johnson. RTOS support for multicore mixed-criticality systems. In *18th Real Time and Embedded Technology and Applications Symposium*, pages 197–208. IEEE, 2012.

20 B. Korel. Automated software test data generation. *IEEE Transactions on software engineering*, 16(8):870–879, 1990.

21 S. Kramer, D. Ziegenbein, and A. Hamann. Real world automotive benchmarks for free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2015.

22 S. Law and I. Bate. Achieving appropriate test coverage for reliable measurement-based timing analysis. In *28th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2016.

23 S. Law, M. Bennett, I. Ellis, S. Hutchesson, G. Bernat, A. Colin, and A. Coombes. Effective Worst-Case Execution Time Analysis of DO178C Level A Software. *Ada User Journal*, 36(3):182–186, 2015.

24 C. Lee, H. Hahn, Y. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE transactions on computers*, 47(6):700–713, 1998.

25 B. Lesage, S. Law, and I. Bate. TACO: An industrial case study of Test Automation for COverage. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, RTNS '18, pages 114–124, 2018.

26 E. Oklapi, M. Deubzer, S. Schmidhuber, E. Lalo, and J. Mottok. Optimization of real-time multicore systems reached by a Genetic Algorithm approach for runnable sequencing. In *Proceedings of the International Conference on Applied Electronics (AE)*, pages 233–238. IEEE, 2014.

27 Antonio Paolillo, Paul Rodriguez, Vladimir Svoboda, Olivier Desenfans, Joël Goossens, Ben Rodriguez, Sylvain Girbal, Madeleine Faugere, and Philippe Bonnot. Porting a safety-critical industrial application on a mixed-criticality enabled real-time operating system. In *Proc. 5th Workshop on Mixed Criticality Systems (WMC), RTSS*, pages 1–6, 2017.

28 RTCA. Software Considerations in Airborne Systems and Equipment Certification. *DO-178C*, 2011.

29 J. Rushby. Partitioning for Safety and Security: Requirements, Mechanisms, and Assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999. Also issued by the FAA.

30 Paulo Baltarejo Sousa, Konstantinos Bletsas, Eduardo Tovar, Pedro Souto, and Benny Åkesson. Unified overhead-aware schedulability analysis for slot-based task-splitting. *Real-Time Systems*, 50(5-6):680–735, 2014.

31 K. Tindell and A. Alonso. A very simple protocol for mode changes in priority preemptive systems. Technical report, Universidad Politecnica de Madrid, 1996.

32 S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE International Real-Time Systems Symposium, (RTSS)*. IEEE, 2007.

From Iteration to System Failure: Characterizing the FITness of Periodic Weakly-Hard Systems

Arpan Gujarati
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany
arpanbg@mpi-sws.org

Mitra Nasri
Delft University of Technology, Delft, The Netherlands
m.nasri@tu-delft.nl

Rupak Majumdar
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany
rupak@mpi-sws.org

Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany
bbb@mpi-sws.org

Abstract

Estimating metrics such as the Mean Time To Failure (MTTF) or its inverse, the Failures-In-Time (FIT), is a central problem in reliability estimation of safety-critical systems. To this end, prior work in the real-time and embedded systems community has focused on bounding the probability of failures in a single iteration of the control loop, resulting in, for example, the worst-case probability of a message transmission error due to electromagnetic interference, or an upper bound on the probability of a skipped or an incorrect actuation. However, periodic systems, which can be found at the core of most safety-critical real-time systems, are routinely designed to be robust to a single fault or to occasional failures (case in point, control applications are usually robust to a few skipped or misbehaving control loop iterations). Thus, obtaining long-run reliability metrics like MTTF and FIT from single iteration estimates by calculating the time to first fault can be quite pessimistic. Instead, overall system failures for such systems are better characterized using multi-state models such as *weakly-hard constraints*. In this paper, we describe and empirically evaluate three orthogonal approaches, PMC, MART, and SAP, for the sound estimation of system's MTTF, starting from a periodic stochastic model characterizing the failure in a single iteration of a periodic system, and using weakly-hard constraints as a measure of system robustness. PMC and MART are exact analyses based on Markov chain analysis and martingale theory, respectively, whereas SAP is a sound approximation based on numerical analysis. We evaluate these techniques empirically in terms of their accuracy and numerical precision, their expressiveness for different definitions of weakly-hard constraints, and their space and time complexities, which affect their scalability and applicability in different regions of the space of weakly-hard constraints.

2012 ACM Subject Classification Computer systems organization → Embedded and cyber-physical systems; Computer systems organization → Real-time systems; Computer systems organization → Reliability; Theory of computation → Probabilistic computation

Keywords and phrases reliability analysis, MTTF/FIT analysis, weakly-hard constraints

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.9

Related Version `https://www.mpi-sws.org/tr/2019-001.pdf`

1 Introduction

Zero risk of failure in the presence of intolerable errors, such as errors due to electromagnetic interference (EMI), can never be achieved [4]. Instead, since environmental sources of interference are stochastic in nature, probabilistic analyses are often used to bound the probabilities of failure to within societally acceptable levels of risk. For example, prior work has investigated the effects of EMI on real-time systems (RTS), resulting in the worst-case probability of an EMI-induced message transmission error [14], or an upper bound on the probability of a skipped or an incorrect actuation in a single iteration of a control loop [22].

However, probabilistic analyses for a single message or iteration are often insufficient for answering whole-system reliability questions in the context of system operation times, and for many certification standards, e.g., ISO-26262 for the automotive domain or DO-178C for the avionics domain. For example, given a fleet of one million autonomous cars, each with an average operating time of five hours a day, if it is desired that less than ten cars break down due to EMI in a year, a reliability metric like *Failures-In-Time* (FIT) [46] – i.e., the expected number of failures in one billion operating hours – is much more useful.

A simplistic approach to obtain such long-run reliability metrics from single iteration estimates is to calculate the time to first fault; this can be done analytically if the probability distribution is known, or through simulation otherwise. However, periodic systems, which can be found at the core of most safety-critical RTS, are routinely designed to be robust to a single fault or to a few occasional failures (a classic example being real-time control applications, which are usually robust to a few skipped or misbehaving control loop iterations). Hence, for such well-engineered RTS, this simplistic approach can be excessively pessimistic. Instead, overall system failures of such temporally robust RTS are better characterized using multi-state models such as *weakly-hard constraints*.

Weakly-hard constraints are widely used and well-studied, especially in the context of temporal requirements [24, 10, 15, 38, 36, 11, 20, 34, 17, 16, 26]. They capture properties related to a discrete sequence of events (or iterations) rather than properties required per each individual event (or a single iteration). For instance, an (m, k) constraint, one of the simplest forms of weakly-hard constraints, requires a periodic system to have at least m successful iterations in any window of k consecutive iterations. It is non-trivial to obtain closed-form FIT bounds for such stateful specifications. Simulation-based methods do not yield exact answers, may even unsafely under-approximate the true failure rate, and scale poorly, especially when analyzing low-probability events. The reliability modeling literature (see related work in §8) focuses mostly on *spatial redundancy*, i.e., analysis of systems with redundant components, but does not explore analysis of periodic systems with intermittent iteration failures, a form of *temporal redundancy* that is common in RTS, but not in general-purpose systems.

In this paper, we bridge the gap between analyzing the failure probability of a single iteration in a time-sensitive periodic system, e.g., network control systems, and analyzing the overall reliability of the system while considering its robustness to a few failed iterations. That is, we consider the problem of soundly and accurately estimating the FIT rate, or its inverse, the *Mean Time To Failure* (MTTF), of a periodic control system with respect to failure models expressed as one or more weakly-hard constraints, given that bounds on the reliability of a single iteration have been computed, e.g., using the techniques of [14, 22].

Any such analysis to upper-bound the FIT rate (*FITness* analysis) must be *generic* enough to support complex weakly-hard requirements in order to stand for the needs of larger and more complicated systems. Further, a FITness analysis must be *accurate*, ideally, *exact*, to minimize pessimism in the final system reliability. Last, but not least, a FITness analysis must

Table 1 Approaches to FIT derivation.

Approach	Accuracy	Scalability	Expressiveness
PMC	Exact	Poor	General system, all properties
MART	Exact	Poor	IID systems, all properties
SAP	Approximate	Good	IID systems, single (m, k) constraint

be *scalable* with respect to the problem size, since capturing asymptotic control properties requires dealing with large problem windows. To respond to each of these requirements, we propose and compare three approaches for FITness analysis: PMC, MART, and SAP.

PMC (**P**robabilistic **M**odel **C**hecking) models the problem as an expected reward problem in a discrete-time Markov chain, which can be solved using state-of-the-art probabilistic model checkers such as PRISM [31] and STORM [18]. PMC is able to express complex robustness constraints as well as sophisticated system models with state-dependent probabilities of failure, such as in [34]. For the common special case of Bernoulli systems, where failure probabilities are independently and identically distributed (IID), martingale theory [32] allows for a direct approach that we call **Mart**. It constructs a system of linear equations, whose solution gives the expected time to failure, and is therefore able to use powerful linear algebra routines such as LAPACK [5] and BLAS [1]. Like PMC, MART provides an *exact* analysis and can support general weakly-hard constraints, but both PMC and MART have limited scalability. To scale to large window-size constraints (see §2 for an example), we introduce **SAp** (**S**ound **Ap**proximation), an empirically-driven, scalable, and yet sound approach designed to evaluate a *single* (m, k) constraint. The tradeoffs of the three proposed techniques are summarized in Table 1.

Our main contribution is a *systematic exploration and empirical evaluation* of the aforementioned methods, each of which is sound by construction, for different points in the weakly-hard constraint design space. We show that for (m, k) constraints where m is close to k, exact analyses scale well, but an implementation must account for numerical imprecisions, especially when failure probabilities are low. On the other hand, SAP is scalable across the entire range of m (for a given k) and, in our experiments, provides safe approximations within a factor of two of the exact answer (when both can be computed). While algorithms for computing reliability measures using Markov chains or martingale theory are not new (see, e.g., [39]), to the best of our knowledge, our paper is the first in applying these techniques in the context of weakly-hard periodic RTS, which are at the heart of many safety-critical systems where sound reliability assessments are essential, and in empirically evaluating the performance-accuracy tradeoffs in this context.

The rest of the paper is organized as follows. We start with an example to motivate the reliability analysis problem studied in this paper (§2). A periodic stochastic model of the system and a formal model of weakly-hard constraints are provided in §3. The three FITness analysis approaches, PMC, MART, and SAP, are discussed in §4–§6, respectively. Results from a comprehensive evaluation of these approaches are discussed in §7. Finally, we conclude with a discussion of related and future work in §8 and §9, respectively.

2 Motivation

As mentioned in §1, this work bridges the gap between single-iteration analyses and full-system reliability analyses for weakly-hard real-time systems. To explain this further, and to motivate our problem statement along with the specific assumptions that we make, we discuss below the steps involved in the end-to-end reliability analysis of a network control system (NCS).

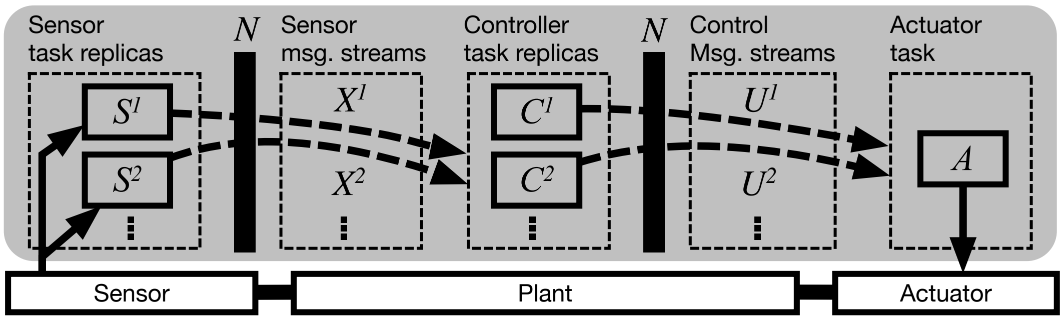


Figure 1 A single-input single-output networked control loop. Solid boxes denote hosts. Each dashed box denotes a task replica set or a set of message streams transmitted by a task replica set. Dashed arrows denote message streams broadcasted over the shared network N.

Consider the single-input single-output NCS illustrated in Fig. 1. For mitigating the effects of environmentally-induced transient faults, the NCS consists of active replicas of its sensor and controller tasks. The safety-certification objective is to ensure that despite the presence of transient faults, the system is expected to provide its intended service for at least X hours, where the threshold X is typically determined in a domain-specific manner, and based on whether the system runs in a continuous mode. In other words, the objective is to ensure that the system's MTTF is lower-bounded by X.

The first step in the reliability analysis is to understand the manifestation of transient faults as errors (i.e., program-visible effects of faults). For example, faults on the network can cause retransmission of messages, which may manifest as deadline violation errors if the bandwidth consumed by the retransmissions exceeds the available slack. An upper bound on the probability of such errors can be quantified *a priori* by considering the peak fault rates expected in practice. The next step is to evaluate error propagation in the system. For example, redundant controller tasks with majority voting on the actuator side may mask a few deadline violation errors experienced by the control command messages, but if there are too many such errors, their effect may propagate to the plant actuation stage. Through exhaustive enumeration and evaluation of all error scenarios, the probability of one or more errors affecting the final plant actuation can thus also be upper-bounded. See [22, 44, 25, 14, 19, 43] for such analyses for actual system configurations.

In a nutshell, the above steps provide us with an upper-bound on the per-iteration failure probability, and these bounds are typically independent of the iteration number, since worst-case scenarios and peak fault rates are used in every step of the analysis. Thus, treating an iteration failure as a full-system failure, the per-iteration failure probability bound could be used to estimate the MTTF by calculating the time to first failed iteration. For example, if the NCS loop operates at a frequency of 100 Hz (i.e., with a time period of 10 ms), and its iteration failure probability is upper-bounded by 10^{-10}, its MTTF evaluates to 10^8 seconds (equivalent to a FIT rate of 36 000). However, this simplistic approach can be quite pessimistic, as evident from the estimated FIT rate, which is extremely high. For instance, if the NCS loop functions correctly despite at most one failed iteration in every four consecutive iterations, the estimated FIT rate drops by several orders of magnitude to 1.08×10^{-5}.[1]

[1] The FIT rate of 1.08×10^{-5} for $m = 3$, $k = 4$, $T = 10\,\text{ms}$, and $P_F = 10^{-10}$ is computed using PMC. In particular, PMC as realized with PRISM yields an MTTF of $T/(3\,600\,000) \times (P_F^3 - 3P_F^2 + 3P_F + 1)/(P_F^4 - 3P_F^3 + 3P_F^2)$ hours. The FIT rate is then computed as $\text{FIT} = 10^9/(\text{MTTF in hours})$.

In general, prior studies have shown that a control system can be (and typically is) designed to withstand occasionally failing iterations, without compromising its intended service (i.e., the first iteration failure does not denote a full-system failure). For example, Majumdar et al. [33] describe a NCS where the control system continues using the previous iteration parameters in case the current iteration is dropped. Using networked control techniques [13], they also provide methods to estimate a minimum dropout rate tolerated by a control system without compromising its stability (e.g., an inverted pendulum control system with mass 0.5 kg, length 0.20 m, and sampling time 10 ms remains asymptotically stable with at least 76.51 % successful iterations). Such constraints directly translate to weakly-hard models. For instance, the inverted pendulum control system could be safely modeled using $m = 77$ and $k = 100$, or using $m = 766$ and $k = 1000$.

In many cases, however, a single asymptotic constraint of the form $m = 77$ and $k = 100$ may not be sufficient to satisfy other performance specifications (such as settling time), and must be appended with an additional short-range "liveness" constraint. For example, given a sampling time of 10 ms, the inverted pendulum control system would surely crash if it experienced 33 consecutive dropouts. In such cases, the temporal robustness of the control system is better modeled using either a harder constraint (e.g., $m = 4$ and $k = 5$ instead of $m = 77$ and $k = 100$) or multiple constraints (e.g., using both $m_1 = 766$ and $k_1 = 1000$ as well as $m_2 = 1$ and $k_2 = 4$) [12]. The objective of this paper is thus to use the temporal robustness property of control systems, modeled using weakly-hard constraints, for estimating their long-run reliability from the per-iteration failure probabilities.

In the subsequent sections, we provide techniques to estimate the MTTF and FIT of temporally robust periodic RTS with such weakly-hard constraints from their per-iteration failure probability bounds. While these bounds account for the maximum possible background interference, system components that are not being analyzed are assumed to execute reliably. This does not imply that the proposed analysis is not useful if a dependent component fails or if dependent components have different robustness criteria, rather it provides a FIT rate for one subsystem, which can then be composed with the FITs of other dependent, dependee, or unrelated subsystems, e.g., using a fault tree analysis. This is a common way of decomposing the reliability analysis of the whole system into manageable components.

3 System Model

We model the problem of computing a system's MTTF as the expected stopping time of a stochastic process.[2] To that end, we model a periodic system S abstractly as a stochastic process $(X_n)_{n \geq 0}$ evolving in discrete time. We assume that S is periodic with a period of T time units, i.e., the observation X_n is emitted at time nT. Each random variable X_n is boolean-valued: $X_n = 1$ indicates that S executes correctly in its n^{th} period and $X_n = 0$ indicates S executes incorrectly. An *execution* of system S is a string in $\{0,1\}^*$ denoting an outcome of the stochastic process $(X_n)_{n \geq 0}$. We emphasize that S is *not* just a single, periodic task, but the entire system, divided into logical iterations. For example, as in the system described in §2, one iteration of the system may involve end-to-end execution of a set of periodic real-time tasks and message exchanges (with period T each). The proposed analyses can also be used to analyze multi-rate systems by analyzing each task (or sets of tasks sharing the same rate) individually and adding their respective FIT bounds.

[2] In probability theory [8], a *stochastic process* is defined as a family of random variables R_t, where t ranges over an arbitrary set I. For a given stochastic process, a *stopping time* is a specific type of random variable, whose value is defined as the time at which the stochastic process exhibits a certain behavior of interest (e.g., in this paper, a violation of the system's weakly-hard constraints).

Failure probabilities in system S can be modeled as a *Bernoulli system*, where each observation X_n is an independent, identically distributed (IID) Bernoulli variable, with $Pr[X_i = 0] = P_F$ and $Pr[X_i = 1] = 1 - P_F$. Such a system represents a periodic system where errors occur independently in each iteration, and the probability of error in each iteration is (bounded by) P_F. It can also represent periodic systems where errors in multiple iterations are dependent, but the bound P_F derived for each iteration is independent of the iteration (this is possible if P_F is derived pessimistically assuming the worst-possible error scenario, which is a common approach in the analysis of hard real-time systems, e.g., [14]).

Alternatively, to capture history-dependence in failures and more accurate iteration-specific error scenarios, the failure probabilities can be modeled more expressively using a *discrete-time labeled Markov chain* [9]. In this case, the system is modeled as a set of states Q and a probabilistic transition function $P : Q \times Q \mapsto [0, 1]$, where $P(s_{n+1}, s_n)$ specifies the probability with which the system transitions from state s_n at any step n to state s_{n+1} at step $n + 1$. Each state is labeled with a Boolean variable denoting success (1) or failure (0), and observation X_n is the label of the (random) state at step n.

Next, we formalize *robustness specifications* to capture the intuition that a periodic RTS, such as one hosting a well-designed control application, continues to provide overall acceptable service despite individual iteration failures, as long as there are not "too many" such iteration failures. In particular, we characterize the set of safe executions for which a periodic system is guaranteed to provide its service as a prefix-closed[3] set of executions $\mathcal{R} \subseteq \{0, 1\}^*$. Thus, the intersection of two robustness specifications is again a robustness specification.

In this paper, we focus on the classic (m, k), $\langle m, k\rangle$, and $\overline{\langle m\rangle}$ robustness specifications, usually called *weakly-hard* specifications, which have been originally proposed in the context of *firm* real-time systems that can tolerate a limited number of deadline misses [11]. Let $\pi_1(s)$ denote the number of 1's (successful iterations) in any string (system execution) $s \in \{0, 1\}^*$. Let $u, v, w, w' \in \{0, 1\}^*$ each denote an execution of system S. Formally, an execution w is (m, k) robust if every window of size k has at least m successes, i.e., $\forall u, v, w' : w = uw'v \wedge |w'| = k \Rightarrow \pi_1(w') \geq m$; it is $\langle m, k\rangle$ robust if every window of size k has at least m consecutive successes, i.e., $\forall u, v, w' : w = uw'v \wedge |w'| = k \Rightarrow \exists u', v' : w' = u'1^m v'$; and it is $\overline{\langle m\rangle}$ robust if there are never more than m consecutive failures, i.e., $\nexists u', v' : w = u'0^{m+1}v'$.

For a given system, one can be interested in several robustness specifications simultaneously, e.g., to express both asymptotic properties (such as "no more than 5% failed iterations") and short-term requirements (such as "no more than two iteration failures in a row"). Thus, for example, we can ask that a system is (m_1, k_1) robust and also $\overline{\langle m_2\rangle}$ robust. This just means that executions of the system satisfy both the (m_1, k_1) constraint and the $\overline{\langle m_2\rangle}$ constraint. In general, given a set of robustness specifications, an execution is considered correct if it satisfies all the specifications in the set.

Given a periodic system S and its robustness specification $\mathcal{R}$, we next define the reliability metrics MTTF and FIT. Let a *system failure* denote an execution that is not in $\mathcal{R}$. For example, for a system with a robustness specification $(2, 5)$, an execution 010100100 denotes a failure (since the last five iterations consist of only one successful iteration). We assume that S stops if it encounters a system failure, and therefore to compute the MTTF and FIT we are interested in a failing execution whose proper prefixes (i.e., prefixes excluding the last iteration) satisfy the robustness specification. Accordingly, given a robustness specification $\mathcal{R}$, we define the *stopping time* of system S as a random variable

$$N(S, \mathcal{R}) = \min \left\{ n \geq 0 \;\middle|\; \begin{array}{l} X_0 \dots X_n \notin \mathcal{R} \wedge \\ \forall i < n \; X_0 \dots X_i \in \mathcal{R} \end{array} \right\}. \tag{1}$$

[3] In a *prefix-closed* set, if an execution belongs to the set, all its prefixes also belong to the set.

The *Mean Time To Failure* (MTTF) is the expectation of the stopping time multiplied by the period T of the system,

$$\text{MTTF} = T \sum_{n=0}^{\infty} n \cdot Pr[N(S, \mathcal{R}) = n]. \tag{2}$$

As mentioned before, the *Failures-In-Time* (FIT) metric is the inverse of the MTTF, with a human-friendly scale factor, to the effect that the FIT represents the expected number of failures in one billion operating hours. Thus, FIT $= 10^9/(\text{MTTF in hours})$.

In §4–§6, we propose three approaches for FIT derivation: PMC, Mart, and SAp. To explain the techniques in detail, we initially focus on a single (m, k) robustness specification, and discuss the applicability of the respective technique for evaluating a generic set of robustness specifications such as $\{(m_1, k_1),\ \langle m_2, k_2\rangle,\ \overline{\langle m_3\rangle}\}$ at the end of each section. Wherever a Bernoulli system is considered, P_F is used to denote the probability of a failed iteration, and $P_S = 1 - P_F$ is used to denote the complement of P_F.

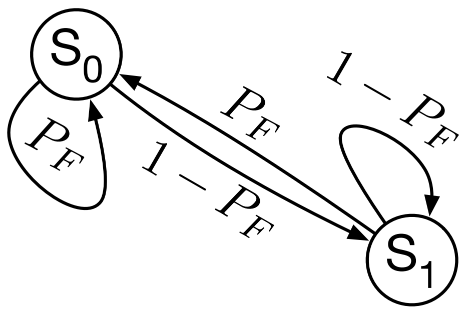
4 PMC: Markov Chain Analysis

We start with the most general method PMC, which is based on discrete-time Markov chains. PMC uses two Markov chains, one for modeling the system, and another (referred to below as the monitor Markov chain) for modeling the weakly-hard robustness constraints as a function of the system's execution history. Therefore, PMC is able to account for both sophisticated system models with state-dependent probabilities as well as complex robustness specifications.

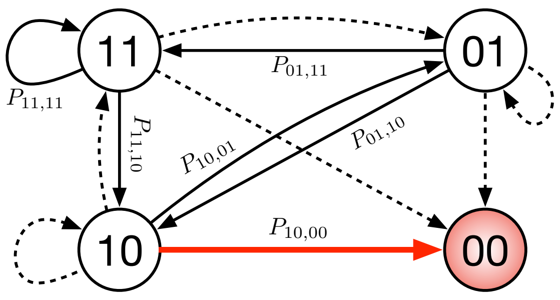
We explain the Markov chain constructions in detail in the following. Our observation is that computing the MTTF reduces to finding the expected total reward in an *absorbing* Markov chain (explained below). Conceptually, our method works for any *regular* robustness specification, i.e., robustness specifications that can be accepted by a finite automaton, but we focus our discussion on the class of weakly-hard robustness specifications, which we expect to be most widely used in practice, and also for concreteness.

Suppose that the system S is modeled as a Markov chain $M = (Q, P, L, s_i)$, where Q denotes a finite set of system states, $P : Q \times Q \mapsto [0, 1]$ denotes the transition probability matrix, $L : Q \mapsto \{0, 1\}$ denotes the state labels with 1 and 0 corresponding to *success* and *failure* (respectively), and $s_i \in Q$ denotes the initial state. For example, if S is a Bernoulli system, then M, as illustrated in Fig. 2(a), consists of states s_0 and s_1 and transition probabilities $P(s_0, s_0) = P(s_1, s_0) = P_F$ and $P(s_0, s_1) = P(s_1, s_1) = 1 - P_F$.

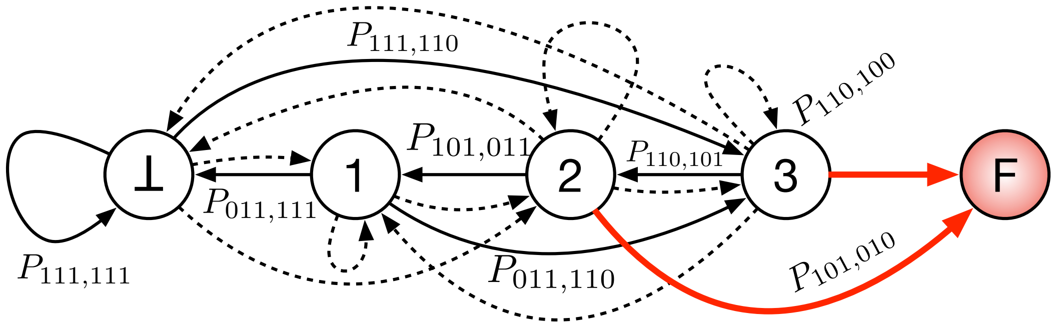
Given the Markov model M and a robustness specification $\mathcal{R} = (m, k)$, we run a *monitor* Markov chain, denoted $\mathit{Monitor}(M, \mathcal{R}) = (Q', P', L', q_i)$, along with M. The monitor tracks a finite execution history of M of length k to decide whether S has *failed*, i.e., whether there were more than $k - m$ failures in the last k steps. Thus, Q' consists of 2^k states, and each state $q \in Q'$ is labeled with a unique label $L'(q) \in \{0, 1\}^k$, e.g., a label of $1^{k-1}0$ implies that every iteration but the last one was successful. Every time M takes a step, the monitor state is updated to reflect the past k steps of M's execution. Thus, the transition probability of $\mathit{Monitor}(M, \mathcal{R})$ from state q with label w to state q' with label w' is $P'(q, q') = P(s, s')$ if system S can transition from history w to w' by transitioning from state s to s'; otherwise, it is $P'(q, q') = 0$. The initial state $q_i \in Q'$ is labeled 1^k to model absence of any failure during system start. In addition, since system S stops as soon as it encounters an execution that does not satisfy (m, k) robustness (recall from §3), we define $\mathit{Bad}(m, k) = \{q \mid q \in Q' \wedge L'(q) \notin \mathcal{R}\}$ as the set of all "bad" states in Q' and make them *absorbing*, i.e., once the monitor enters a state in $\mathit{Bad}(m, k)$, it does not transition into another state.



(a) Markov chain for a Bernoulli system.



(b) Monitor (Type 1) for $k = 2$.



(c) Monitor (Type 2) for $(2, 3)$.

Figure 2 PMC approach. In inset (b), $P_{x_1x_2,\, y_1y_2}$ is a shorthand for transition probability $P'(q, q')$ where states q and q' have labels $L'(q) = x_1x_2$ and $L'(q') = y_1y_2$, respectively. In inset (c), $P_{x_1x_2x_3,\, y_1y_2y_3}$ is a shorthand for $P'(q, q')$, where states q and q' correspond to execution histories $x_1x_2x_3$ and $y_1y_2y_3$, respectively. Since the Type 2 monitor is represented more concisely, the node labels in inset (c) are not equal to the execution histories, e.g., label "3" indicates an execution history of "110" where the latest iteration has failed. In insets (b) and (c), transitions with zero probability are marked with dashed arrows, and states in *Bad*$(1, 2)$ and *Bad*$(2, 3)$ are colored red.

As an example, the monitor representation for $\mathcal{R} = (1, 2)$ is illustrated in Fig. 2(b). All execution histories of length $k = 2$ belong to set $\{11, 10, 01, 00\}$, and hence the monitor Markov chain consists of four nodes, each labeled with a unique execution history from this set. An execution history of 01 denotes that the latest iteration was a success (1), whereas the iteration before that was a failure (0). Thus, depending on whether the next iteration is a success or a failure, the system can transition from the state labeled 01 into either a state labeled 11 or 10, respectively. All other transitions from this state have zero probability. Since the robustness constraint $\mathcal{R} = (1, 2)$ defines a robust system execution as one in which at least one out of every two consecutive iterations is successful, the set of bad states in this example is a singleton corresponding to the state labeled 00 with two consecutive failures.

Given the monitor Markov chain construction described above, we reduce the MTTF computation to deriving the expected number of steps until the monitor enters a bad state. For this, assume that each step of the monitor has a reward of 1. We define the *expected number of steps* E as the expected reward until any state in *Bad*(m, k) is reached, starting from the initial state $q_i \in Q'$. E can be obtained using probabilistic model checkers such as PRISM [31] and STORM [18]. Thus, if system S has period T, and E is the expected number of steps until a state in *Bad*(m, k) is reached, the MTTF of S with respect to robustness specification (m, k) is given by MTTF $= T \times E$.

Note that the monitor representation discussed above is independent of m. While the monitor's simple structure makes it trivial to implement, its $O(2^k)$ space complexity can be detrimental in practice. Fortunately, for the common case where $k - m \ll k$, e.g., $(98, 100)$, the monitor representation can be optimized to be much more space efficient. Since the system stops as soon as the (m, k) constraint is violated, we need not keep any executions that have more than $k - m$ failures. In other words, it suffices to store a limited history as a string of length $k - m$, where each element in the string is from $\{1, \ldots, k\} \cup \{\perp\}$, representing the positions along the previous k steps when a failure occurred ($\perp$ is used in case we have seen fewer than $k - m$ failures). Furthermore, we can coalesce all states in $Bad(m, k)$ into a single "bad" state. The space complexity of the resulting monitor is only $O((k+1)^{(k-m)} + 1)$. For example, Fig. 2(c) illustrates the monitor representation for $\mathcal{R} = (2, 3)$, which consists of only five states whereas otherwise it would have required eight states.

Similarly, for $m \ll k$, we can optimize the model by storing a history as a string of length m, where each element in the string is from $\{1, \ldots, k\}$. We refer to the three representations, i.e., the default one, the optimized version for $k - m \ll k$, and the optimized version for $m \ll k$, as Type 1, Type 2, and Type 3 models, respectively.

Compared to the aforementioned monitor representations for an (m, k) robustness specification, monitor representation for $\langle m, k \rangle$ and $\overline{\langle m \rangle}$ robustness specifications are both simpler and more efficient (we do not formally define these due to space constraints). For $\langle m, k \rangle$ robustness, the monitor needs to keep track of positions corresponding to **(i)** the latest run of 1's of length at least m and **(ii)** the current run of 1's of length at most m. For (i), since the beginning and the end of run can be any element in a window of size k, a string of length two belonging to $\{1 \ldots k\}^2$ is needed, whereas for (ii), since the current run must always include the latest element, a string of length one belonging to $\{1 \ldots m\}$ is sufficient. In both cases, $\perp$ can be used to denote the absence of a run, resulting in a space complexity of $O((k+1)^2 \cdot (m+1))$. For $\overline{\langle m \rangle}$ robustness, the monitor can be simplified even further, since we only need one accumulator to store the current sequence of consecutive 0's, and so the space complexity is $O(m)$. For multiple specifications, i.e., for robustness constraints of the form $\mathcal{R} = \{(m, k), \langle m', k' \rangle, \overline{\langle m'' \rangle}\}$, we run the monitor for each specification in parallel, and set Bad to denote states where *some* monitor is in a bad state.

Implementation of the PMC approach using the PRISM probabilistic model checker, along with a discussion of model construction times and model solving times are provided in §7.

5 Mart: The Martingale Approach

Computing the MTTF using PMC reduces to the problem of solving a system of linear equations [9]. In the special case of Bernoulli systems, there is a direct and elegant approach to deriving an equivalent system of linear equations whose solution provides the expected stopping time of the system (i.e., the MTTF), without going through the process of Markov chain modeling. Thus, even though this approach, denoted MART, does not model history-dependent failures like PMC, it is easy to implement scalably on top of mature linear algebra libraries such as LAPACK [5] and BLAS [1].

We now summarize MART for (m, k) robustness. The first step in MART is similar to enumerating the "bad" states of the monitor Markov chain in the PMC approach. In particular, we list all *failure strings* that correspond to a violation of the (m, k) constraint, i.e., all strings in $\{0, 1\}^{\leq k}$ in which at least $k - m + 1$ failures occur. We do this by fixing the last position to be a failure and then choosing all possible combinations of $k - m$ indices from the set $\{1, \ldots, k\}$. In the second step, given an exhaustive list of failure strings, we reduce the problem of computing MTTF to that of computing the *expected waiting time* until one of the failure strings is realized by the system execution.

To find the expected waiting time, we use an elegant algorithm from the theory of occurrence patterns in repeated experiments proposed by Li [32]. Li's algorithm translates the failure strings into a set of linear equations, such that solving these linear equations directly yields an expected waiting time for each individual failure string (i.e., until a specific failure string is realized by the system) as well as an expected waiting time until any of the failure strings manifests. To compute the MTTF, we require only the latter. We summarize Li's algorithm and MTTF derivation using the algorithm in the following.

Let $\Pi = \{\pi_1, \pi_2, \ldots\}$ be the set of failure strings obtained in the first step. Let $|\pi_i|$ denote the length of a string $\pi_i \in \Pi$, and let $\pi_{i,j}$ denote the j^{th} character in string π_i. Key to Li's algorithm is a combinatorial operator '$*$' (see Eq. 2.3 in [32]) between any pair of strings π_a and π_b from Π, which is defined as follows.

$$\pi_a * \pi_b = (\delta_{1,1}\delta_{2,2}\ldots\delta_{x,x}) \;+\; (\delta_{2,1}\delta_{3,2}\ldots\delta_{x,x-1}) \;+\; \ldots \;+\; (\delta_{x-1,1}\delta_{x,2}) \;+\; (\delta_{x,1}), \tag{3}$$

$$\text{where } x = |\pi_a|,\ y = |\pi_b|, \text{ and } \delta_{i,j} = \begin{cases} \frac{1}{P_F} & \text{if } i \in [1,x],\ j \in [1,y],\ \pi_{a,i} = \pi_{b,j} = 0 \\ \frac{1}{P_S} & \text{if } i \in [1,x],\ j \in [1,y],\ \pi_{a,i} = \pi_{b,j} = 1 \\ 0 & \text{otherwise.} \end{cases}$$

Using this operator, the expected waiting time e_0 until any one of the sequence patterns in Π occurs for the first time satisfies the following linear system of equations,

$$\begin{bmatrix} 0 & 1 & 1 & \ldots & 1 \\ -1 & \pi_1 * \pi_1 & \pi_2 * \pi_1 & \ldots & \pi_n * \pi_1 \\ -1 & \pi_1 * \pi_2 & \pi_2 * \pi_2 & \ldots & \pi_n * \pi_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ -1 & \pi_1 * \pi_n & \pi_2 * \pi_n & \ldots & \pi_n * \pi_n \end{bmatrix} \begin{bmatrix} e_0 \\ e_1 \\ e_2 \\ \vdots \\ e_n \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \tag{4}$$

where $n = |\Pi|$. Thus, if S has period T, the MTTF is given by $e_o \times T$. As mentioned before, Li's algorithm also yields the expected waiting times for each individual failure string in $\pi_1, \pi_2, \ldots, \pi_n \in \Pi$, which are given by $e_1, e_2, \ldots, e_n$, respectively.

For example, consider a system with period 5 ms, iteration failure probability bounded by $P_F = 0.1$, and robustness specification $(2,3)$, i.e., at most one 0 is allowed in any execution of length three. The set of all failure strings in $\{0,1\}^{\leq 3}$ that violate $(2,3)$ robustness and end in a failure is $\Pi = \{00, 010, 100\}$. Using Eq. 3, $\pi_2 * \pi_2$ is computed as follows.

$$\begin{aligned} \pi_2 * \pi_2 &= \delta_{1,1}\delta_{2,2}\delta_{3,3} + \delta_{2,1}\delta_{3,2} + \delta_{3,1} \\ &= \delta_{1,1}\delta_{2,2}\delta_{3,3} + \delta_{3,1} && \{\text{since } \pi_{2,2} \neq \pi_{2,1},\ \delta_{2,1} = 0\} \\ &= 10 \cdot \delta_{2,2} \cdot 10 + \delta_{3,1} && \{\text{since } \pi_{2,1} = \pi_{2,3} = 0,\ \delta_{1,1} = \delta_{3,3} = 1/P_F = 10\} \\ &= 10 \cdot \delta_{2,2} \cdot 10 + 10 && \{\text{since } \pi_{2,3} = \pi_{2,1} = 0,\ \delta_{3,1} = 1/P_F = 10\} \\ &= 10 \cdot \frac{10}{9} \cdot 10 + 10 = \frac{1090}{9} && \{\text{since } \pi_{2,2} = 1,\ \delta_{2,2} = 1/P_S = 10/9\} \end{aligned}$$

Other $\pi_a * \pi_b$'s can be similarly computed, resulting in the following system of linear equations:

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ -1 & 110 & 10 & 110 \\ -1 & 10 & 1090/9 & 10 \\ -1 & 0 & 100/9 & 1000/9 \end{bmatrix} \begin{bmatrix} e_0 \\ e_1 \\ e_2 \\ e_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \tag{5}$$

which yields $e_0 = 62.63$ and MTTF $= e_0 \times 5 = 313.15$ ms.

With the Mart approach, accounting for a generic set of robustness specifications, such as $\{(m_1, k_1), \langle m_2, k_2\rangle, \overline{\langle m_3\rangle}\}$, is relatively straightforward in comparison to PMC. We need to modify only the first step of Mart to obtain an appropriate set of failure strings that corresponds to a violation of any of the robustness specifications, which is used as before to instantiate the system of linear equations defined in Eq. 4. However, we must ensure that any two patterns $\pi_a, \pi_b \in \Pi$ do not contain one another [32]. This is possible if, for example, the failure patterns for constraints $(95, 100)$ and $\overline{\langle 3\rangle}$ are merged. For such cases, the longer pattern is removed from Π, since the shorter pattern occurs first.

6 SAp: An Approximate Analysis

The PMC and Mart approaches presented in §4 and §5, respectively, can be used to determine the exact value of MTTF for systems with multiple different types of weakly-hard robustness specifications. Unlike Mart, PMC even allows estimating the MTTF of systems that do not resemble a Bernoulli process. However, neither PMC nor Mart scale to large values of m and k. Thus, with scalability being the primary motivation, we present next an approximate analysis SAp. Like Mart, SAp can be used only for Bernoulli systems. However, unlike PMC and Mart, SAp is applicable only for a single (m, k) robustness constraint; it does not support constraints of the form $\langle m, k\rangle$ or $\overline{\langle m\rangle}$, or combinations thereof. Most importantly, though, SAp is *sound*, that is, it estimates an approximate value of the MTTF that lower-bounds the actual MTTF (as given by the exact analyses PMC and Mart).

Before we describe SAp, recall the definition of MTTF from §3. For brevity, let $g(n) = Pr[N(S, \mathcal{R}) = n]$, which is a factor in the integrand in Eq. 2. SAp consists of two key steps. In the first step, we derive a lower bound on $g(n)$, denoted $g_{LB}(n)$. For this, we split the (m, k) robustness specification into three conditions, compute an exact or lower bound on the probability for each of these conditions, and then compute a product of these probabilities. In the second step, we integrate $n \cdot g_{LB}(n)$ numerically (but in a sound manner) to strictly lower-bound the MTTF of system S. We discuss both these steps in detail next.

For S to violate the (m, k) specification for the first time during its n^{th} iteration, the following three conditions must hold. $\boldsymbol{(E_1)}$ The n^{th} iteration must fail; $\boldsymbol{(E_2)}$ exactly $k - m$ iterations must fail out of the $k - 1$ iterations between the $(n - k + 1)^{\text{th}}$ and the $(n-1)^{\text{th}}$ iteration; and $\boldsymbol{(E_3)}$ fewer than $k-m+1$ iterations must fail out of any k consecutive iterations, among the first $n - 1$ iterations. Then $g(n) = Pr(E_1) \times Pr(E_2) \times Pr(E_3)$. Now, $Pr(E_1) = P_F$, and summing over all possible combinations of $k - m$ iteration failures in $k - 1$ consecutive iterations yields $Pr(E_2) = \binom{k-1}{k-m} P_F^{(k-m)} P_S^{(m-1)}$.

However, obtaining the exact value of $Pr(E_3)$ is challenging. To tackle this challenge, we use the *a-within-consecutive-b-out-of-c:F* model [30, §11.4] (or *a/Con/b/c:F* in short), proposed originally for a system that consists of c $(c \geq a)$ linearly ordered components and that fails iff at least a $(a \leq b)$ components fail among any b consecutive components. Thus, in terms of the (m, k) constraint, for $a = k - m + 1$, $b = k$, and $c = n - 1$, a successful execution of an a/Con/b/c:F system is equivalent to condition E_3, and the *reliability* of an a/Con/b/c:F system, whose approximations have been well studied in the past, yields $Pr(E_3)$. In particular, since we are interested in a sound approximation, we reuse the reliability lower bound $R_{LB}(a, b, c)$ of the a/Con/b/c:F system as proposed by Sfakianakis, et al. [42].

Sfakianakis et al.'s analysis [42] breaks the problem into smaller subproblems for which exact analyses are available and that can be computed quickly. However, neither Sfakianakis et al. nor any prior work explicitly enumerates the reliability definitions for an exhaustive set of parameters, i.e., which covers all possibles values of parameters a, b, and c. Therefore,

Table 2 Reliability lower bound of a linear a/Con/b/c:F system with IID components. **Type** indicates whether the reliability definition is an exact value or a lower bound (LB).

Case	Definition	Type	Source
$a = 0$	$R_1(a,b,c) = 0$	Exact	–
$a = 1$	$R_2(a,b,c) = P_S^c$	Exact	–
$a = 2,\ c \leq 4b$	$R_3(a,b,c) = \sum_{i=0}^{\lfloor \frac{c+b-1}{b} \rfloor} \binom{c-(i-1)(b-1)}{i} P_F^i P_S^{c-i}$	Exact	[30, §11.4.1] (Eq 11.10)
$a = 2,\ c > 4b$	$R_4(a,b,c) = R_3(a,b,b+t-1)(R_3(a,b,b+3))^u$ where $t = (c-b+1) \bmod 4$ and $u = \left\lfloor \frac{c-b+1}{4} \right\rfloor$	LB	[30, §11.4.1] (Eq. 11.16)
$a > 2,\ c \leq 2b,$ $a = b$	$R_5(a,b,c) = \begin{cases} 1 & 0 \leq c < a \\ 1 - P_F^a - (c-k)P_F^a P_S & a \leq c \leq 2a \end{cases}$	Exact	[30, §9.1.1] (Eq. 9.20)
$a > 2,\ c \leq 2b,$ $a \neq b,\ c \leq b$	$R_6(a,b,c) = \sum_{i=c-a+1}^{c} \binom{c}{i} P_S^i P_F^{c-i}$	Exact	[30, §7.1.1] (Eq. 7.2)
$a > 2,\ c \leq 2b,$ $a \neq b,\ c > b$	$R_7(a,b,c) = \sum_{i=0}^{a-1} \binom{b-s}{i} P_F^i P_S^{b-s-i} M(a',s,2s)$ where $s = c - b$ and $a' = a - i$, and $M(a',s,2s) = \begin{cases} 1 & a' > s \\ R_2(a',s,2s) & a' = 1 \\ R_3(a',s,2s) & a' = 2 \\ R_5(a',s,2s) & a' > 2 \wedge a' = s \\ R_7(a',s,2s) & a' > 2 \wedge a' \neq s \end{cases}$	Exact	[30, §11.4.1] (Eq. 11.14)
$a > 2,\ c > 2b$	$R_8(a,b,c) = R_\phi(a,b,b+t-1)(R_\phi(a,b,b+3))^u$ where $t = (c-b+1) \bmod 4$ and $u = \left\lfloor \frac{c-b+1}{4} \right\rfloor$, and $R_\phi(a,b,c) = \begin{cases} R_5(a,b,c) & a = b \\ R_6(a,b,c) & a \neq b \wedge a \leq b \\ R_7(a,b,c) & a \neq b \wedge a > b \end{cases}$	LB	[30, §11.4.1] (Eq. 11.16)

we provide an unambiguous definition of the reliability lower bound $R_{LB}(a,b,c)$ that draws from Sfakianakis et al.'s analysis for large values of c and from other prior works for some special cases and smaller values of c. Note that in many cases, there are multiple ways to define $R_{LB}(a,b,c)$, in which case we prefer a definition that can be quickly computed. We summarize our definition of $R_{LB}(a,b,c)$ in Table 2.[4] Using this reliability lower bound and the definitions of $Pr(E_1)$ and $Pr(E_2)$, a lower bound $g_{LB}(n)$ on $g(n)$ is given by

$$g_{LB}(n) = \binom{k-1}{k-m} P_F^{(k-m+1)} P_S^{(m-1)} R_{LB}(k-m+1, k, n-1)\,. \tag{6}$$

[4] In our definition of $R_{LB}(a,b,c)$ in Table 2, notice that while we are interested in a reliability lower bound, we point to Eq. 11.16 in [30, §11.4.1], which refers to an upper bound. This mismatch is due to a slight inconsistency in how the textbook chapter [30, §11.4.1] adopts the result from the original paper by Sfakianakis et al. [42]. Notations L and U in Table I in [42] denote lower and upper bounds (respectively) on the failure rate of the system. Eq. 11.16 in [30, §11.4.1] uses the same notation. Thus, UB_a in Eq. 11.16 in [30, §11.4.1] actually refers to an upper bound on the system failure probability, and not an upper bound on the system reliability (although the text in the chapter may seem contradictory). Since we require a lower bound on the system reliability, and since system reliability is one minus its failure probability, we use $1 - UB_a$, where UB_a is defined as in Eq. 11.16 in [30, §11.4.1].

The next step is to use $g_{LB}(n)$ for lower-bounding the system's MTTF. This requires solving Eq. 2 with $g_{LB}(n)$ in place of $Pr[N(S, \mathcal{R}) = n]$. Unfortunately, we were not able to obtain a closed-form solution with current symbolic solvers due to the complicated definition of $g_{LB}(n)$. In particular, $g_{LB}(n)$ is defined in terms of $R_{LB}(k - m + 1, k, n - 1)$, which is a recursive expression with complex definitions of its subproblems, as can be seen from Table 2. Therefore, similar to numerical integration methods, we adopt an empirical solution for MTTF derivation that is both fast and reasonably accurate. We empirically compute the value of function $g_{LB}(n)$ at finitely many sampling points $d_0, d_1, d_2, \ldots, d_D \in \mathbb{N}$ such that $d_0 = k - m + 1$, and $d_0 < d_1 < d_2 < \ldots < d_D$. Using the empirically-determined values $g_{LB}(d_0)$, $g_{LB}(d_1)$, $\ldots$, $g_{LB}(d_D)$, we derive a lower bound on the MTTF in Lemma 1 below.

The derivation in Lemma 1 depends on the property that $g_{LB}(n)$ (defined in Eq. 6) decreases with increasing n, which in turn requires that $R_{LB}\,(a,\, b,\, c)$ decreases with increasing c (since all the terms except $R_{LB}\,(k - m + 1,\, k,\, n - 1)$ in the definition of $g_{LB}(n)$ are independent of n). While this property trivially holds for cases $a = 0$ and $a = 1$, proving the property for cases $a = 2$ and $a > 2$ is not trivial. We have provided a detailed proof of this monotonicity property for the more general case $a \geq 2$ online [21, Section IV.B].

▶ **Lemma 1.** *A lower bound on the MTTF of system S with period T and robustness specification $\mathcal{R} = (m, k)$ is given by $T \sum_{i=0}^{D-1} (g_{LB}(d_{i+1}) \times (d_{i+1} - d_i) \times (d_i))$.*

Proof. From Eq. 2, MTTF is defined as $T \sum_{n=0}^{\infty} n \cdot Pr[N(S, \mathcal{R}) = n]$. Since $g_{LB} \leq g(n) = Pr[N(\mathcal{S}, \mathcal{R}) = n]$, we lower-bound the MTTF as $MTTF \geq T \sum_{n=0}^{\infty} (n \times g_{LB}(n))$.

Next, we split the summation range $(0, \infty)$ in the above equation into a finite number of subintervals $(0, d_0]$, $(d_0,\, d_1]$, $\ldots$, $(d_{D-1},\, d_D]$, and $(d_D,\, \infty)$. Further, since all terms under the summation are non-negative, and since we are interested in a lower bound, we drop the summation terms corresponding to subintervals $(0, d_0]$ and $(d_D,\, \infty)$. Thus, we obtain another lower bound MTTF $\geq T \sum_{i=0}^{D-1} \sum_{n=d_i}^{d_{i+1}} (n \times g_{LB}(n))$.

Now, since $g_{LB}(n)$ is decreasing with increasing n, for each interval $(d_i,\, d_{i+1}]$, we replace $g_{LB}(n)$ with $g_{LB}(d_{i+1})$, which is a constant with respect to n. With this replacement, we get MTTF $\geq T \sum_{i=0}^{D-1} (g_{LB}(d_{i+1}) \times \sum_{n=d_i}^{d_{i+1}} n)$. Finally, summing the arithmetic progression, and using inequalities $d_{i+1} - d_i + 1 > d_{i+1} - d_i$ and $d_i + d_{i+1} > 2d_i$, we get the desired bound:

$$\begin{aligned} \text{MTTF} &\geq T \sum_{i=0}^{D-1} \left(g_{LB}(d_{i+1}) \times \sum_{n=d_i}^{d_{i+1}} n \right) \\ &\geq T \sum_{i=0}^{D-1} \left(g_{LB}(d_{i+1}) \times (d_{i+1} - d_i) \times (d_i) \right). \end{aligned}$$ ◀

Since scalability is the primary motivation for SAp, we choose $D \ll d_D$, so that the MTTF lower bound can be quickly computed using Lemma 1. We further choose the sampling points $d_1, \ldots, d_D$ to minimize the amount of pessimism introduced by numerical integration. Another source of inaccuracy is the use of the reliability lower bound $R_{LB}(a, b, c)$ proposed by Sfakianakis et al. [42], which inherently introduces some pessimism. We discuss the choice of sampling points in detail in §7, and compare SAp with PMC and Mart in terms of accuracy.

As mentioned before, SAp is customized for a single (m, k) constraint and does not apply to $\langle m, k \rangle$ or $\overline{\langle m \rangle}$ robustness specifications. We leave similar approximate analysis for the other robustness constraints as future work.

Table 3 MTTF values derived using PRISM engines.

Engine	Iterations	Epsilon	MTTF for $P_F = 10^{-2}$	MTTF for $P_F = 10^{-10}$
Explicit	10^{04}	10^{-06}	–	–
	10^{09}	10^{-06}	3.36×10^{05}	0.23×10^{15}
	10^{09}	10^{-10}	3.41×10^{05}	1.21×10^{17}
Exact	N/A	N/A	3.41×10^{05}	3.33×10^{29}

7 Evaluation

The objective of this section is threefold. We discuss implementation choices and challenges, compare the three types of Markov chain models discussed in §4, and then explore the scalability versus accuracy tradeoffs of PMC, MART, and SAP. Since the approximate analysis SAP is not applicable to generic robustness specifications as defined in §3, and since (m, k) constraints are the limiting factor when it comes to scaling up the analysis, we focus on Bernoulli systems and a single (m, k) constraint in the evaluation. In the end, we revisit the strengths and weaknesses of each approach. All experiments were carried out on Intel Xeon E7-8857 v2 machines with 4×12 cores and 1.5 TB of memory.

7.1 Implementation Choices

In the following, we highlight important implementation choices that affect the accuracy and speed of the analyses. We realized PMC using the state-of-the-art probabilistic model checker PRISM [31].[5] However, configuring PRISM properly to ensure that the estimated results are both accurate and sound is not trivial. PRISM provides many different configuration options that affect the method used for linear equation solving (e.g., Jacobi, Gauss-Seidel, etc.), the model checking engine (MTBDD, Sparse, Hybrid, or Explicit), parameters for precision tuning (i.e., the *epsilon* value and maximum number of iterations for convergence checks during iterative linear solving), and even options to select *exact* (with arbitrary precision) or *parametric* model checking (where some model parameters are not fixed). Choosing the right set of options is thus important because they can significantly affect the estimated MTTF.

With the parametric model checking option, PRISM outputs the MTTF as a function of parameter P_F, e.g., denoting P_F as q, the MTTF for robustness specification $(2, 4)$ is:

$$T \times \frac{q^5 - 3q^4 + 3q^3 - 2q^2 - q - 1}{q^6 - 3q^5 + 4q^4 - 3q^3}. \tag{7}$$

Parametric model checking is thus an ideal choice since it allows for fast reliability analysis across a range of failure probabilities without the need to build and check the model repeatedly. However, as we show later, parametric model checking is also the costliest analysis approach. Thus, for scalability purposes, we also considered both exact and non-exact model checking.

We observed that non-exact model checking resulted in significant inaccuracy. For example, Table 3 reports the MTTF results for specification $(2, 4)$ obtained with non-exact model checking (using PRISM's Explicit engine) and with exact model checking (currently implemented by PRISM as a special case of parametric model checking). The non-exact engine

[5] Our implementation of PMC using PRISM for a robustness specification of $(5, 10)$ is explained in the Appendix, which is available online as part of an extended tech report [23]. An empirical comparison of PRISM with STORM [18], a more recent probabilistic model checker, is also provided in the Appendix.

Table 4 Percentage errors in FIT ($\mathcal{R} = (8, 10)$ and $y = 1.23456789$).

Precision	$P_F = y \cdot 10^{-10}$	$P_F = y \cdot 10^{-30}$	$P_F = y \cdot 10^{-50}$
10	-2.20×10^{-00}	-3.96×10^{-01}	-1.42×10^{-00}
20	$+1.81 \times 10^{-04}$	-2.70×10^{-04}	$+3.04 \times 10^{-04}$
30	$+3.39 \times 10^{-07}$	-5.26×10^{-07}	$+1.36 \times 10^{-06}$
40	-2.75×10^{-10}	$+1.20 \times 10^{-09}$	-2.00×10^{-09}
50	-1.89×10^{-14}	$+2.99 \times 10^{-13}$	-4.80×10^{-13}

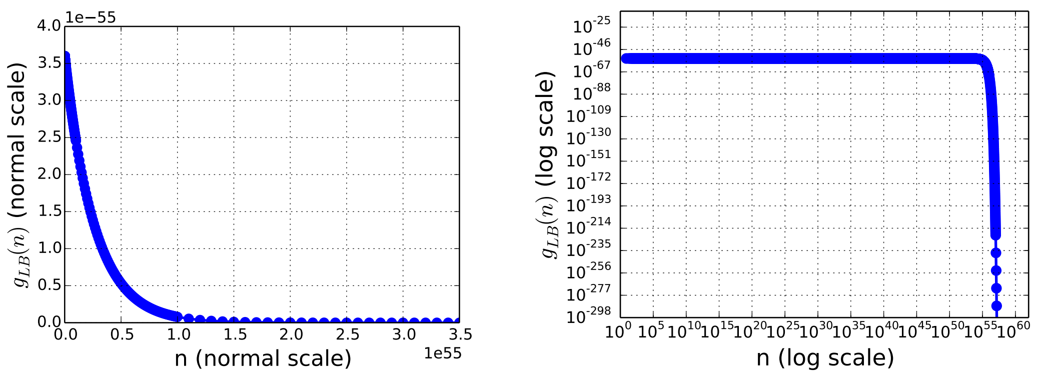
did not converge (first row of the table) for default configuration options. For $P_F = 10^{-10}$, even upon decreasing the epsilon value and increasing the maximum number of iterations, the estimated MTTF is several orders of magnitude off from the exact value, indicating the sensitivity of non-exact model checking to small probabilities. In our evaluation of PMC, we thus worked only with parametric and exact model checking. We denote these variants of PMC as PMC-P and PMC-E, respectively.

The MART approach was implemented in C++ using the *Elemental* [2] library, since it uses LAPACK-based routines [5] for solving linear equations, allows for arbitrary precision using the GNU MPFR library [3], and also allows for parallel computing using OpenMPI [7]. SAP was implemented in Python using the *mpmath* [6] library for arbitrary precision. Thus, for MART and SAP, unlike for PMC-E, we could explicitly set the global working *precision*, i.e., the number of decimal digits used to represent the floating point significand.

However, the choice of the global working precision was not obvious. Table 4 reports the percentage errors in the estimated FIT when the precision is varied from 10 to 50, with respect to the FIT estimated using a precision of 1000. The results indicate that low precision may result in significant errors if P_F is also small, and sometimes, the results can even be unsafe (i.e., resulting in negative errors). In general, estimating a precision that is safe to use based on the computations involved requires rigorous analysis, e.g., [28]. To be on the safe side, we used a precision of 1000 bits for both MART and SAP, which ensured that any remaining errors were of negligible magnitude.

Finally, when implementing SAP, recall that we need a mechanism to choose an appropriate set of data points d_0, d_1, d_2, ..., d_D over which to run the empirical computations. We discuss this mechanism with the help of an example. Let $m = 3$, $k = 10$, and $P_F = 10^{-7}$. In Fig. 3(a), we illustrate $g_{LB}(n)$ given these parameters. Since the MTTF lower bound derived using SAP depends on $g_{LB}(n)$, the key idea is to ensure that points d_0, d_1, d_2, ..., d_D are sufficient to trace the shape of function $g_{LB}(n)$, and that the magnitude of $g_{LB}(n)$ is negligible beyond $n = d_D$. The first point d_0, as mentioned before, is set to $(k - m + 1)$. To compute the last point d_D, i.e., the point at which $g_{LB}(n)$ becomes negligible, we observed the logarithm of function $g_{LB}(n)$ for $n \in \{1, 10^1, 10^2, 10^3, \ldots\}$. That is, we plotted the function $g_{LB}(n)$ on a logarithmic scale for both the x- and y-axes as in Fig. 3(b), and then determined a threshold at which the curve starts falling rapidly (e.g., $d_D \approx 10^{55}$ in Fig. 3(b)).

The intermediate points $d_1, d_2, \ldots, d_{D-1}$ were chosen such that the step size $d_{i+1} - d_i$ between any two consecutive points d_i and d_{i+1} **(i)** is small enough to closely track the function $g_{LB}(n)$, and **(ii)** yet still proportional to the order of magnitude of d_i, to avoid evaluating an exponential number of points. For example, while generating Fig. 3, the step size was 1 for $n \in (10, 100]$ and 10^{52} for $n \in (10^{53}, 10^{54}]$.



(a) Normal-scale axes. **(b)** Log-scale axes.

Figure 3 Sampling points $g_{LB}(d_0)$, $g_{LB}(d_1)$, ..., $g_{LB}(d_D)$ for $m = 3$, $k = 10$, $P_F = 10^{-7}$, and $T = 10\,\text{ms}$ in (a) normal scale and (b) log scale. In this example, $D = 5050$ and $d_D = 9.90 \times 10^{57}$.

7.2 Evaluating PMC Model Types

Recall from §4 that we introduced three different types of Markov chain models – Type 1, Type 2, and Type 3 – each resulting in a different asymptotic model size. Does the use of one model over the other affect the computation times or even the model building times in practice? To answer this question, we measured the asymptotic model sizes for $k = 20$ and $m \in [1, k-1]$, and compared the measurements with the model size and build time statistics reported by PRISM. We also measured the checking time statistics for $k = 10$ (since model checking for $k = 20$ frequently timed out). We summarize the results for PMC-E in Fig. 4.

Fig. 4a plots the asymptotic size for each model type, indicating that none of the models is an optimal choice for all parameters. Fig. 4b reports the number of elements in the transition matrix as reported by PRISM. The number of transition matrix nodes varies with m in the same way as the asymptotic model size, but the absolute numbers are less than the asymptotic sizes. This is because PRISM already prunes some states that are unreachable during the build process. Figs. 4c and 4d illustrate the time to build and check the models, respectively. The model construction time for each model type is proportional to the respective model size. The model checking time, however, is independent of the model type, since the models are equivalent and result in the same set of linear equations.

In summary, to achieve maximum scalability, it is important to choose a model that requires the minimum time for construction. In the subsequent experiments, we thus use the asymptotic model sizes as a guideline to choose the appropriate model type for an (m, k) specification. That is, if $k = 20$, based on Fig. 4a we use the Type-3 model if $m \leq 4$, the Type-2 model if $m \geq 16$, or the Type-1 model otherwise.

7.3 The Scalability vs. Accuracy Tradeoff

We start by evaluating the scalability of the analyses PMC-P, PMC-E, Mart, and SAp by measuring the analysis duration for each $k \in [2, 20]$ for four different configurations of m and P_F: **(i)** $m = \lfloor k/2 \rfloor$ and $P_F = 10^{-10}$ (Fig. 5a); **(ii)** $m = \lfloor k/2 \rfloor$ and $P_F = 10^{-20}$ (Fig. 5b); **(iii)** $m = k-2$ and $P_F = 10^{-10}$ (Fig. 5c); and **(iv)** $m = k-2$ and $P_F = 10^{-20}$ (Fig. 5d). Since evaluating (m, k) requires maximum time if $m = k/2$ and minimum time if m is close to either 1 or $k-1$ (see Fig. 4d), results for (i) and (iii) indicate the minimum and maximum scalability that can be achieved by the analyses; whereas results for (ii) and (iv) help us to understand the impact, if any, of P_F's value on the analysis scalability.

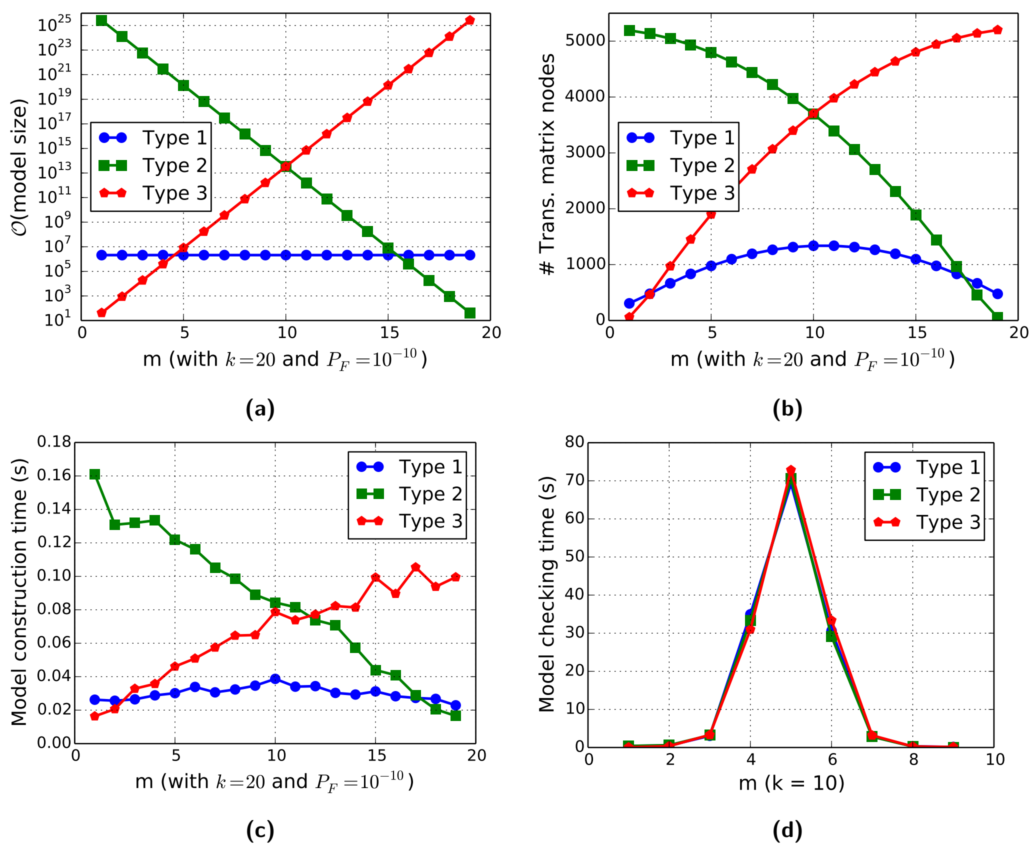


Figure 4 Comparing the three PMC model types using PMC-E. While measuring the checking time statistics, $k = 10$ was used, since model checking for $k = 20$ frequently timed out.

First, as evident from each graph, and as expected, PMC-P, PMC-E, and MART do not scale well in comparison to SAP. For configurations of type (i) and (ii), where $m = \lfloor k/2 \rfloor$ (see Figs. 5a and 5b), PMC-P and PMC-E scale only up to $k = 9$ and $k = 11$, respectively. The MART approach performs better and scales up to $k = 15$, mainly because it gives up exactness (but still guarantees soundness owing to its very high precision). In contrast, SAP easily scales up to the maximum value of $k = 20$. Also, notice that while SAP's analysis time grows exponentially in k (the y-axis is log scale), PMC's and MART's analysis times grow super-exponentially. For configurations of type (iii) and (iv) where $m = k - 2$ (Figs. 5c and 5d), PRISM-based analyses scale better than in the first two configurations because the Type-3 model allows for a concise representation of the (m, k) specification in this case and hence fast building of the model. SAP's scalability also improves significantly in this case because the recursion involved in computing $R_{LB}(k - m + 1, k, n - 1)$ for the empirical data points is eliminated in this case. Between configurations (i) and (ii), as well as between configurations (iii) and (iv), only the failure probability P_F is changed from 10^{-10} to 10^{-20}. As a result, PMC-E takes an order of magnitude more time. This is because lower probabilities require more space for exact representation, and hence more time for computations on these representations. SAP is also affected since the number of data points to be measured is larger in this case. MART is unaffected because irrespective of P_F, it uses a precision of 1000. PMC-P is also unaffected since it is independent of P_F.

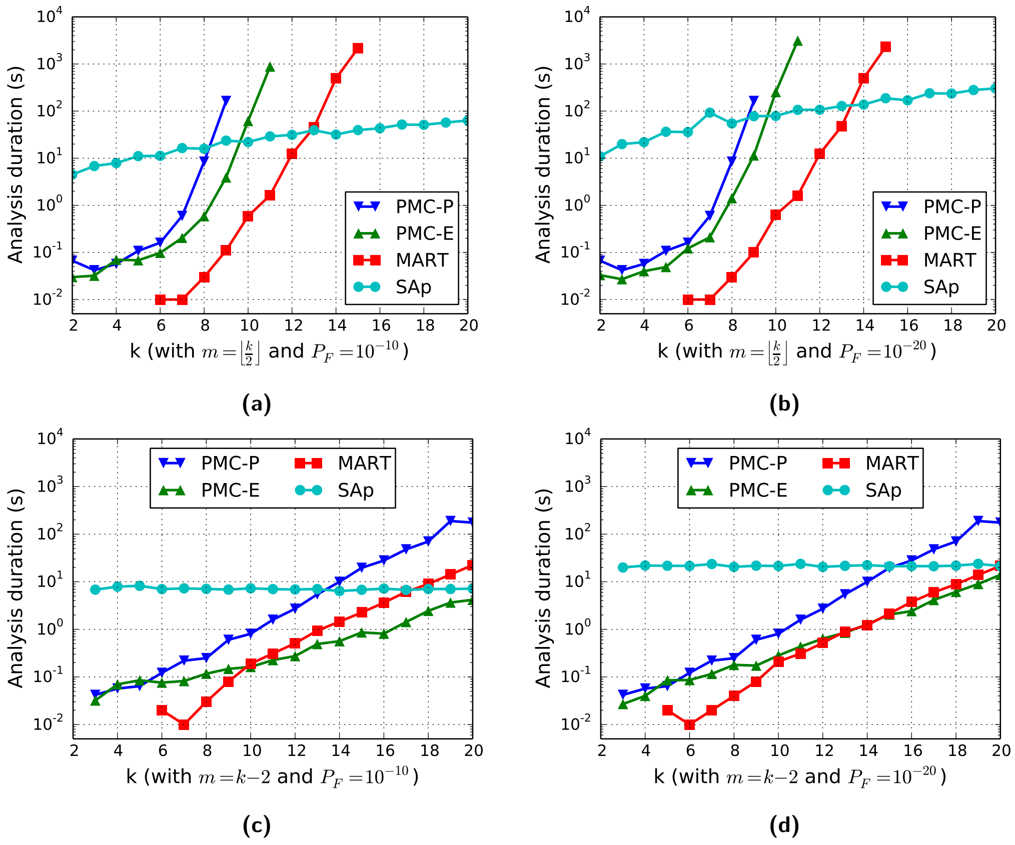
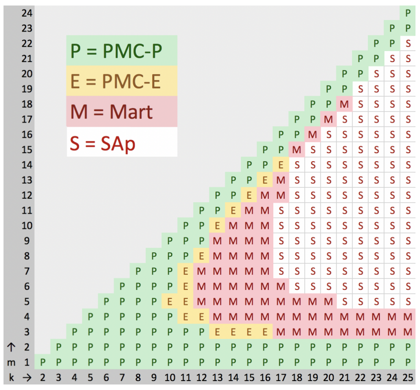


Figure 5 Comparing analysis duration for PMC-P, PMC-E, MART, and SAP. The analysis duration for MART for $k \leq 5$ was extremely small and is hence not illustrated. The configuration $k = 2$ in (c) and (d) was ignored since $(0, 2)$ is not a valid (or rather a trivial) specification.

To summarize the discussion on analysis scalability, we illustrate in Fig. 6a for each $k \in [1, 25]$ and $m \in [2, k-1]$ whether analyses PMC-P, PMC-E, MART, and SAP finished on time, i.e., within a one-hour timeout window. For each cell, P denotes that PMC-P was successful, E denotes that PMC-P timed out but PMC-E was successful, M denotes that both PMC-P and PMC-E timed out but MART was successful, and S denotes that only SAP was successful. Clearly, the results indicate that exact analyses can be used only if $k \leq 15$, or else if m is either very small or very large relative to k. Thus, for larger values of k, an approximate analysis, such as SAP, is needed, that trades some accuracy for scalability. But is SAP accurate enough to be useful at very large values of k? And is it accurate for small values of k so that the costly exact analyses may not be needed at all? To answer these questions, we evaluate next SAP's accuracy with respect to MART and PMC.

In Fig. 6c (similar in structure to Fig. 6a), we report the percentage error in the MTTF obtained using SAP versus that obtained from either PMC or MART (PMC was preferred, if available) for each $k \in [2, 12]$ and $m \in [1, k-1]$. As expected, SAP always resulted in a lower, pessimistic MTTF than PMC and MART since it is sound by construction. Thus, error signs are not explicitly denoted in the figure.

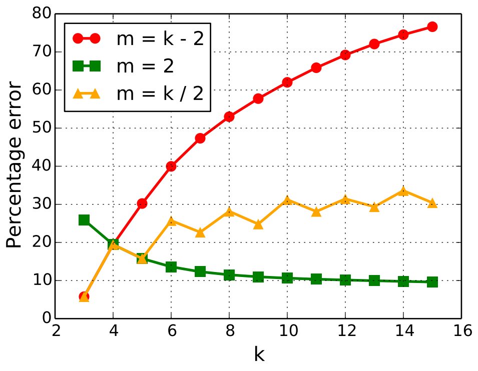
We make the two key observations regarding SAP's accuracy. First, even for small values of k, the relative errors are significant (see the red cells in Fig. 6c denoting specifications with relative error greater than 50%). This validates the need for an exact analysis whenever



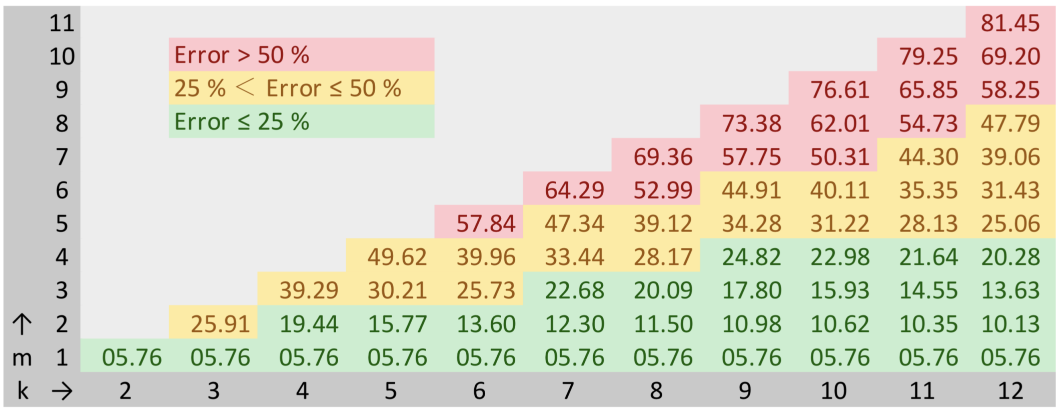
P = PMC-P
E = PMC-E
M = Mart
S = SAp

24																								P
23																							P	P
22																						P	P	S
21																					P	P	S	S
20																				P	P	S	S	S
19																			P	P	S	S	S	S
18																		P	P	M	S	S	S	S
17																	P	P	M	S	S	S	S	S
16																P	P	M	S	S	S	S	S	S
15															P	P	M	S	S	S	S	S	S	S
14														P	P	E	S	S	S	S	S	S	S	S
13													P	P	E	M	S	S	S	S	S	S	S	S
12												P	P	E	M	M	S	S	S	S	S	S	S	S
11											P	P	E	M	M	S	S	S	S	S	S	S	S	S
10										P	P	E	M	M	M	S	S	S	S	S	S	S	S	S
9									P	P	P	M	M	M	M	S	S	S	S	S	S	S	S	S
8								P	P	P	E	M	M	M	M	S	S	S	S	S	S	S	S	S
7							P	P	P	E	M	M	M	M	M	S	S	S	S	S	S	S	S	S
6						P	P	P	P	E	M	M	M	M	M	M	S	S	S	S	S	S	S	S
5					P	P	P	P	E	E	M	M	M	M	M	M	M	M	M	S	S	S	S	S
4				P	P	P	P	P	P	E	E	M	M	M	M	M	M	M	M	M	M	M	M	M
3			P	P	P	P	P	P	P	P	P	E	E	E	E	M	M	M	M	M	M	M	M	M
↑ 2		P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P
m 1	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P
k →	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

(a)



(b)



Error > 50 %
25 % < Error ≤ 50 %
Error ≤ 25 %

11											81.45
10										79.25	69.20
9									76.61	65.85	58.25
8								73.38	62.01	54.73	47.79
7							69.36	57.75	50.31	44.30	39.06
6						64.29	52.99	44.91	40.11	35.35	31.43
5					57.84	47.34	39.12	34.28	31.22	28.13	25.06
4				49.62	39.96	33.44	28.17	24.82	22.98	21.64	20.28
3			39.29	30.21	25.73	22.68	20.09	17.80	15.93	14.55	13.63
↑ 2		25.91	19.44	15.77	13.60	12.30	11.50	10.98	10.62	10.35	10.13
m 1	05.76	05.76	05.76	05.76	05.76	05.76	05.76	05.76	05.76	05.76	05.76
k →	2	3	4	5	6	7	8	9	10	11	12

(c)

Figure 6 Quantifying the scalability vs. accuracy tradeoff. **(a)** Scalability results for different values of m and k. **(b)** SAp's accuracy trend for $m = 2$, $m = k/2$, and $m = k - 2$. **(c)** Summary of SAp's accuracy with respect to Mart and PMC for different values of k and m.

feasible. Second, the relative errors are higher if the ratio m/k is closer to one. To investigate this further, we also plot the percentage errors for $m = k - 2$, $m = 2$, and $m = k/2$ with respect to k in Fig. 6b. From this figure, we observe that in all evaluated cases, the MTTF estimated with SAp was within an order of magnitude of the exact MTTF. Since in the context of reliability analyses the order of magnitude is typically of prime interest (rather than the exact value), we conclude that SAp is reasonably accurate for large values of k.

7.4 Discussion

Mart outperforms both PMC-P and PMC-E, which is not surprising. In fact, for the scenario with IID iteration failure probabilities that we evaluated, Mart directly represents the underlying system of linear equations without needing to construct a model. PMC's benefits lie in its ability to express non-IID iteration failure probabilities. SAp on the other hand scales much better than PMC and Mart, at the cost of acceptable, but non-zero pessimism. To conclude, PMC, Mart, and SAp are useful alternatives for reliability evaluation depending on the values of m and k. PMC and Mart are ideal to evaluate *short-range* safety properties that are usually applied on short window lengths e.g., such as "there should not be more than 3 consecutive failures in any window of 10 iterations" [17]. In contrast, SAp can evaluate asymptotic properties that are defined over a large window of events and reflect minimum acceptable longterm *quality-of-service levels*, e.g., such as "at least 90% of actuation commands must be applied on the plant in every 100 iterations" [33, 41].

Although we focused on a binary failure type in this paper, i.e., each iteration was categorized either as a successful iteration or a failed iteration, one could also use fine-grained label types for each iteration, such as deadline violation, message loss, miscomputation, and so on. That is, an execution of system S could be modeled as a string in $\{0 \dots \lambda\}^*$, instead of a string in $\{0, 1\}^*$, where λ is the number of failure categories. Both PMC and MART easily extend to such systems. In contrast, SAP has limited extensibility in its current form, since our objective when designing SAP was primarily to scale the evaluation of (m, k) specifications that are widely used in practice. However, the same blueprint could be used to safely approximate other types of robustness specifications as well, i.e., by breaking each specification into smaller events, computing the product of respective event probabilities (or a lower bound), and then reusing Lemma 1 for MTTF estimation. We leave such extensions for future work.

8 Related Work

Weakly-hard constraints have been widely studied in the context of firm real-time systems to represent robustness of a time-sensitive task against occasional timing failures [24, 10, 15, 38, 36, 16]. In particular, the focus has been on **(i)** analyzing task schedulability according to a given weakly-hard (usually (m, k)) constraint [36, 37], **(ii)** design of online schedulers to meet these constraints [24, 10, 15, 16], and **(iii)** co-design approaches to find the schedulable set of (m, k) parameters that maximizes an application's quality of service [45, 29, 17]. Most recently, Pazzaglia et al. [34] introduced state-based representation of the evolution of a control system with respect to deadline misses, and showed the merits of having multiple (m, k) constraints for a control application. In contrast, Huang et al. [26] focused on the safety verification problem of nonlinear weakly-hard systems by modeling them using hybrid automata. Huang et al. [27] also discuss new research directions in applying weakly-hard constraints to general-purpose networked systems. None of these papers provides a means for bounding a system's MTTF with respect to its weakly-hard specification.

In the general reliability literature, there is a long tradition of work on deriving a system's MTTF if the occurrence of failures is described by well-known probability distributions (see [30] for a comprehensive overview). Similarly, the problem of evaluating the reliability of series- or parallel-redundant systems, both with and without repairs, in the context of robustness specifications such as *k-out-of-n*, *consecutive-k-out-of-n*, multidimensional *consecutive-k-out-of-n*, etc. is well understood, e.g., see [35, 40]. However, the available techniques in this domain do not directly apply to the problem studied in this paper. Either the constraints cannot be reduced to these techniques or symbolically integrating the applicable technique over an infinite domain is not trivial. Further, for multiple weakly-hard specifications, a model-based approach helps to accurately account for common failure sequences.

9 Conclusion

We proposed methods for safely bounding the MTTF of periodic systems with stochastic faults w.r.t. weakly-hard robustness specifications. Empirical evaluations showed that an exact (even parametric) analysis is feasible when $k - m$ is small, and that an approximate analysis is scalable and (within the parameters of our experiments) produces bounds within $2\times$ of the exact bounds. In future work, it would be interesting to consider more expressive models of stochastic faults [39], such as continuous-time models and dynamic fault trees. We also plan to extend SAP for evaluating other types of robustness constraints. Finally, a holistic analysis of periodic systems that are composed of multiple subsystems with possibly different periods and different weakly-hard constraints, as opposed to analyzing each of these subsystems independently, would help reduce pessimism in the overall reliability assessment.

References

1 BLAS (Basic Linear Algebra Subprograms). URL: `http://www.netlib.org/blas/`.

2 Elemental: distributed-memory dense and sparse-direct linear algebra and optimization — Elemental. URL: `http://libelemental.org/`.

3 The GNU MPFR Library. URL: `https://www.mpfr.org/`.

4 IEC 61158-1:2014 | IEC Webstore. URL: `https://webstore.iec.ch/publication/4624`.

5 LAPACK – Linear Algebra PACKage. URL: `http://www.netlib.org/lapack/`.

6 mpmath - Python library for arbitrary-precision floating-point arithmetic. URL: `http://mpmath.org/`.

7 Open MPI: Open Source High Performance Computing. URL: `https://www.open-mpi.org/`.

8 Robert B. Ash. *Basic probability theory.* Dover Publications, Mineola, N.Y, dover ed edition, 2008. OCLC: ocn190785258 (pbk.).

9 Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* The MIT Press, Cambridge, Mass, 2008. OCLC: ocn171152628.

10 Guillem Bernat and Alan Burns. Combining (/sub m//sup n/)-hard deadlines and dual priority scheduling. In *Proceedings Real-Time Systems Symposium*, pages 46–57, San Francisco, CA, USA, 1997. IEEE Comput. Soc. `doi:10.1109/REAL.1997.641268`.

11 Guillem Bernat, Alan Burns, and Albert Liamosi. Weakly hard real-time systems. *IEEE Transactions on Computers*, 50(4):308–321, April 2001. `doi:10.1109/12.919277`.

12 Rainer Blind and Frank Allgower. Towards Networked Control Systems with guaranteed stability: Using weakly hard real-time constraints to model the loss process. In *2015 54th IEEE Conference on Decision and Control (CDC)*, pages 7510–7515, Osaka, December 2015. IEEE. `doi:10.1109/CDC.2015.7403405`.

13 Michael S. Branicky, Stephen M. Phillips, and Wei Zhang. Scheduling and feedback co-design for networked control systems. In *Proceedings of the 41st IEEE Conference on Decision and Control, 2002.*, volume 2, pages 1211–1217, Las Vegas, NV, USA, 2002. IEEE. `doi:10.1109/CDC.2002.1184679`.

14 Ian Broster, Alan Burns, and Guillermo Rodriguez-Navas. Timing Analysis of Real-Time Communication Under Electromagnetic Interference. *Real-Time Systems*, 30(1-2):55–81, May 2005. `doi:10.1007/s11241-005-0504-z`.

15 Marco Caccamo and Giorgio Buttazzo. Exploiting skips in periodic tasks for enhancing aperiodic responsiveness. In *Proceedings Real-Time Systems Symposium*, pages 330–339, San Francisco, CA, USA, 1997. IEEE Comput. Soc. `doi:10.1109/REAL.1997.641294`.

16 Hyunjong Choi, Hyoseung Kim, and Qi Zhu. Job-Class-Level Fixed Priority Scheduling of Weakly-Hard Real-Time Systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Montreal, Quebec, Canada, April 2019. IEEE. `doi:10.1109/RTAS.2019.00028`.

17 Hoon Sung Chwa, Kang G. Shin, and Jinkyu Lee. Closing the Gap Between Stability and Schedulability: A New Task Model for Cyber-Physical Systems. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 327–337, Porto, April 2018. IEEE. `doi:10.1109/RTAS.2018.00040`.

18 Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A Storm is Coming: A Modern Probabilistic Model Checker. In *Computer Aided Verification*, pages 592–600, Cham, 2017. Springer International Publishing.

19 Joanne Bechta Dugan and Randy Van Buren. Reliability evaluation of fly-by-wire computer systems. *Journal of Systems and Software*, 25(1):109–120, April 1994. doi:10.1016/0164-1212(94)90061-2.

20 Oliver Gettings, Sophie Quinton, and Robert I. Davis. Mixed criticality systems with weakly-hard constraints. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems - RTNS '15*, pages 237–246, Lille, France, 2015. ACM Press. doi:10.1145/2834848.2834850.

21 Arpan Gujarati, Mitra Nasri, and Björn B. Brandenburg. Lower-Bounding the MTTF for Systems with (m, k) Constraints and IID Iteration Failure Probabilities. Technical Report MPI-SWS-2018-004, Max Planck Insitute for Software Systems, April 2018. URL: https://www.mpi-sws.org/tr/2018-004.pdf.

22 Arpan Gujarati, Mitra Nasri, and Björn B. Brandenburg. Quantifying the Resiliency of Fail-Operational Real-Time Networked Control Systems. In *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, volume 106 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:24, Barcelona, Spain, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/lipics.ecrts.2018.16.

23 Arpan Gujarati, Mitra Nasri, Rupak Majumdar, and Björn B. Brandenburg. From Iteration to System Failure: Characterizing the FITness of Periodic Weakly-Hard Systems. Technical Report MPI-SWS-2019-001, Max Planck Insitute for Software Systems, Germany, May 2019. URL: https://www.mpi-sws.org/tr/2019-001.pdf.

24 Moncef Hamdaoui and Parameswaran Ramanathan. A dynamic priority assignment technique for streams with (m, k)-firm deadlines. *IEEE Transactions on Computers*, 44(12):1443–1451, December 1995. doi:10.1109/12.477249.

25 Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. System level performance analysis – the SymTA/S approach. *IEE Proceedings - Computers and Digital Techniques*, 152(2):148, 2005. doi:10.1049/ip-cdt:20045088.

26 Chao Huang, Wenchao Li, and Qi Zhu. Formal verification of weakly-hard systems. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems Computation and Control - HSCC '19*, pages 197–207, Montreal, Quebec, Canada, 2019. ACM Press. doi:10.1145/3302504.3311811.

27 Chao Huang, Kacper Wardega, Wenchao Li, and Qi Zhu. Exploring weakly-hard paradigm for networked systems. In *Proceedings of the Workshop on Design Automation for CPS and IoT - DESTION '19*, pages 51–59, Montreal, Quebec, Canada, 2019. ACM Press. doi:10.1145/3313151.3313165.

28 Anastasiia Izycheva and Eva Darulova. On Sound Relative Error Bounds for Floating-point Arithmetic. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, FMCAD '17, pages 15–22, Austin, TX, 2017. FMCAD Inc. URL: http://dl.acm.org/citation.cfm?id=3168451.3168462.

29 Matthias Kauer, Damoon Soudbakhsh, Dip Goswami, Samarjit Chakraborty, and Anuradha M. Annaswamy. Fault-tolerant Control Synthesis and Verification of Distributed Embedded Systems. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 56:1–56:6, 3001 Leuven, Belgium, Belgium, 2014. European Design and Automation Association. URL: http://dl.acm.org/citation.cfm?id=2616606.2616675.

30 Way Kuo and Ming J. Zuo. *Optimal reliability modeling: principles and applications.* John Wiley & Sons, Hoboken, N.J, 2003.

31 Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *Computer Aided Verification*, volume 6806, pages 585–591. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. doi:10.1007/978-3-642-22110-1_47.

32 Shuo-Yen Robert Li. A Martingale Approach to the Study of Occurrence of Sequence Patterns in Repeated Experiments. *The Annals of Probability*, 8(6):1171–1176, 1980. URL: https://www.jstor.org/stable/2243018.

33 Rupak Majumdar, Indranil Saha, and Majid Zamani. Performance-aware scheduler synthesis for control systems. In *Proceedings of the 9th ACM international conference on Embedded software - EMSOFT '11*, page 299, Taipei, Taiwan, 2011. ACM Press. `doi:10.1145/2038642.2038689`.

34 Paolo Pazzaglia, Luigi Pannocchi, Alessandro Biondi, and Marco Di Natale. Beyond the Weakly Hard Model: Measuring the Performance Cost of Deadline Misses. In *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, volume 106 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:22, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.ECRTS.2018.10`.

35 Hoang Pham. Optimal design of k-out-of-n redundant systems. *Microelectronics Reliability*, 32(1):119–126, January 1992. `doi:10.1016/0026-2714(92)90091-X`.

36 Gang Quan and Xiaobo Hu. Enhanced fixed-priority scheduling with (m,k)-firm guarantee. In *Proceedings 21st IEEE Real-Time Systems Symposium*, pages 79–88, November 2000. `doi:10.1109/REAL.2000.895998`.

37 Sophie Quinton and Rolf Ernst. Generalized Weakly-Hard Constraints. In *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*, Lecture Notes in Computer Science, pages 96–110. Springer Berlin Heidelberg, 2012.

38 Parameswaran Ramanathan. Overload management in real-time control applications using (m, k)-firm guarantee. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):549–559, June 1999. `doi:10.1109/71.774906`.

39 Enno Ruijters and Mariëlle Stoelinga. Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. *Computer Science Review*, 15-16:29–62, February 2015. `doi:10.1016/j.cosrev.2015.03.001`.

40 Raaj K. Sah. An explicit closed-form formula for profit-maximizing k-out-of-n systems subject to two kinds of failures. *Microelectronics Reliability*, 30(6):1123–1130, January 1990. `doi:10.1016/0026-2714(90)90291-T`.

41 Indranil Saha, Sanjoy Baruah, and Rupak Majumdar. Dynamic Scheduling for Networked Control Systems. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, HSCC '15, pages 98–107, New York, NY, USA, 2015. ACM. `doi:10.1145/2728606.2728636`.

42 Michael Sfakianakis, Stratis G. Kounias, and Alexander E. Hillaris. Reliability of a consecutive k-out-of-r-from-n:F system. *IEEE Transactions on Reliability*, 41(3):442–447, September 1992. `doi:10.1109/24.159817`.

43 Purnendu Sinha. Architectural design and reliability analysis of a fail-operational brake-by-wire system from ISO 26262 perspectives. *Reliability Engineering & System Safety*, 96(10):1349–1359, October 2011. `doi:10.1016/j.ress.2011.03.013`.

44 Fedor Smirnov, Michael Glaß, Felix Reimann, and Jürgen Teich. Formal reliability analysis of switched ethernet automotive networks under transient transmission errors. In *Proceedings of the 53rd Annual Design Automation Conference on - DAC '16*, pages 1–6, Austin, Texas, 2016. ACM Press. `doi:10.1145/2897937.2898026`.

45 Damoon Soudbakhsh, Linh T. X. Phan, Oleg Sokolsky, Insup Lee, and Anuradha Annaswamy. Co-design of Control and Platform with Dropped Signals. In *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*, ICCPS '13, pages 129–140, New York, NY, USA, 2013. ACM. `doi:10.1145/2502524.2502542`.

46 Susan Stanley. MTBF, MTTR, MTTF & FIT Explanation of Terms. URL: `http://www.bb-elec.com/Learning-Center/All-White-Papers/Fiber/MTBF,-MTTR,-MTTF,-FIT-Explanation-of-Terms/MTBF-MTTR-MTTF-FIT-10262012-pdf.pdf`.

End-To-End Deadlines over Dynamic Topologies

Victor Millnert
Lund University, Sweden

Johan Eker
Lund University, Sweden
Ericsson Research, Sweden

Enrico Bini
University of Turin, Italy

Abstract

Despite the creativity of the scientific community and the funding agencies, the underlying model of computation behind IoT, WSN, cloud, edge, fog, and mist is fundamentally the same; Computational nodes which are dynamically interconnected to form a system in where both processing capacity and connectivity may vary over time. On top of such a system, we consider applications that need packets to flow along a path and adhere to end-to-end deadlines. This application model is motivated by both control and automation systems, as well as telecom systems. The challenge is to guarantee end-to-end deadlines when allowing nodes and applications to join or leave.

The mainstream, and to some extent natural, approach to this is to relax the stringency of the constraint (e.g. use probabilistic guarantees, soft deadlines). In this paper we take a different approach and keep the end-to-end deadlines as hard constraints and instead partially limit the freedom of how nodes and applications are allowed to leave and join. We present a theoretical framework for modeling such systems along with proofs that deadlines are always honored.

2012 ACM Subject Classification Computer systems organization → Cloud computing; Networks → Cloud computing

Keywords and phrases Cloud, real-time, end-to-end latency guarantee, end-to-end response time guarantee, dynamic network

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.10

1 Introduction

Cloud computing plays a pivotal role in the ongoing digitalization of both the industry and the society at large. This transformation borrows many technologies and concepts from traditional cloud applications (mail services, ride sharing, media streaming, e-commerce etc.). The focus for these applications is commonly availability, scalability, and price/performance. While response time is of great concern for such applications, determinism is usually not. For the next generation of cloud-applications, such as industrial automation, collaborative traffic, and telecom systems, predictable timing is crucial. In order to secure a successful transformation and allow such timing-critical systems, the underlying cloud infrastructure must be able to guarantee predictable end-to-end response-times.

One example of such an application could be a dynamically reconfigurable production cell in a manufacturing plant, where the elements in the production cell are connected to the cloud and controlled centrally. The cloud provides large-scale compute and storage capacity. Cloud back-end systems are typically implemented as a service meshes consisting of networks of interconnected microservices. The production cell may be dynamically configured to adapt to changes, such as hardware failures, or respond to external events, etc. Elements may thus dynamically join or leave a cell. Figure 1 illustrates such a production cell with industrial robots connected to the cloud, which provide services for automation, analytics, artificial intelligence and machine learning algorithms. The topology of the service network may

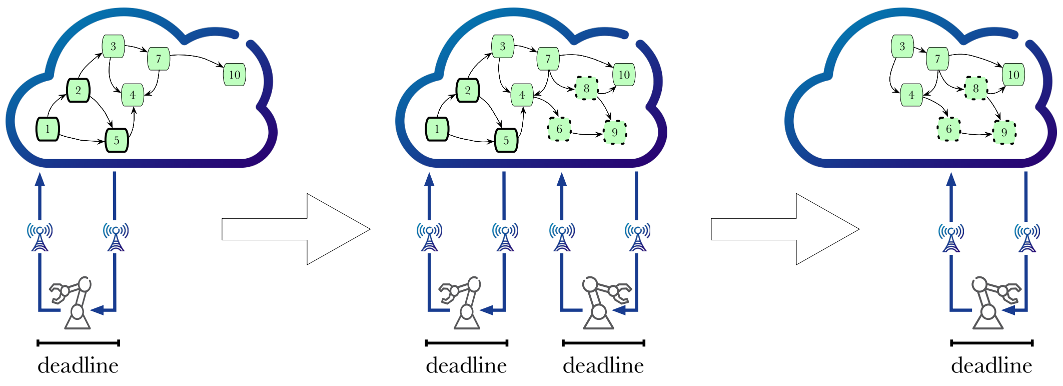


Figure 1 Illustration of configurable, and mobile, manufacturing cells connected to a set of smart services in the cloud. It highlights the changes of the network topology which arises when new robots, and cloud services, join and leave the system. With the upcoming 5G standard it becomes possible to establish a low-latency wireless connection between the robots and the cloud. However, in order to guarantee the required low end-to-end response-time for the robots, there still remains a challenge to ensure a low end-to-end response-time within the cloud, especially during the transitions of the network topology.

therefore change over time as robots join and leave the network. As mentioned earlier, the challenge is to still guarantee that the end-to-end deadline required by the robots are always met, despite dynamic changes of the network topology.

This paper addresses the challenge of allowing network churn and still ensures predictable end-to-end response times within the cloud. A framework that allows for transitions of the network topology is presented. Moreover, it is formally proved that the suggested framework guarantees that it will never violate any timing constraints.

2 The system model

The focus of this paper is to develop a framework where applications are implemented as flows of packets through a network of nodes. Each node offers a service to the incoming packets. The goal is to manage the on-line arrival/departure of flows and the on-line arrival/departure of nodes of the network in a way such that end-to-end deadlines of packets are honored, while allowing for topology changes.

Section 2.1 presents definitions and assumptions with respect to the services offered by *nodes*. Applications are then defined as a set of interconnected services and referred to as *flows*. Definitions and assumptions on flows are given in Section 2.2. Finally, in Section 2.3, we present some assumptions on how these flows and nodes interact with the resource manager.

2.1 Model of the nodes

A node represents an entity that offers a service to the incoming packets. The time taken by a node to provide the service to an incoming packet is given in Def. 1 below. The nodes in the system are denoted by $\mathcal{V}(t) \subset \mathbb{N}$, which is the set of indices of the nodes present at time t. The terms “node” and “vertex” are used interchangeably[1].

[1] We may use the term “node” when we refer to its capacity to process packets, while the term “vertex” is more often used when referring to the topological structure of the network.

▶ **Definition 1** (Response time). *We define the* response time $R_i(t)$*, as the time taken by a packet entering node i at time t to be processed.*

From the definition above, we remark the following facts:

- The response time $R_i(t)$ accounts for all sources of delay possibly occurring within node i (queuing, interference, processing, etc.).
- All incoming packets are treated in the same way regardless of the flow they belong to. Differentiantig packets depending on the application they belong to is feasible. This would require to attach the flow index j to the response time, which would become then $R_{i,j}(t)$. However, this choice poses a notational challenge only with no conceptual added value. For this reason we believe that letting the packet response time depend only on the node (and time) does not significantly impact the applicability of the results. We leave the case of per-flow response time to future works.
- Moreover, in contrast to a large body of the research on real-time systems, this paper does not address how the response time may be computed based on the amount of incoming workload (due to the arrival of packets) and the amount of processing capacity of a node, etc. The interested readers can refer to a vast relate literature addressing this aspects [12, 18, 7, 16, 17, 19, 25, 3].
  Rather, the focus is on the interactions between nodes aimed at guaranteeing end-to-end deadlines in the context of a dynamic network.

Naturally, the service provided to packets by a node might include some minimum requirements, which determines a lower bound to the response time of the service time of a node. The node is unable to process packets in less time than this value. Hence, to model the minimum time needed by a node to process a packet, we introduce the following definition.

▶ **Definition 2** (Response time lower bound). *We define the* response time lower bound $\underline{R}_i$*, as the minimum a packet may take to be processed by node i.*

In practice, $\underline{R}_i$ may represent the pure processing time of a packet, without any interference or delay of any kind.

Finally, we assume that the node is capable of *controlling its response-time*, as stated below in Assumption 1. This assumptions is backed by the vast body of research for different ways of controlling the response-time of cloud services. For instance, in [18] they use the concept of "brownout control" to ensure that the response-time of a server is within a desired limit. In [7] control theory is used to modify the processing capacity of the web servers to control response times. In [17] a combination of scaling the resources of the nodes with an admission control is used to ensure that the deadlines of the nodes in the network are met. More interesting work addressing this is found in [19, 25, 3].

▶ **Assumption 1** (Response-time control). *We assume that a node i is capable of controlling its response-time $R_i(t)$ such that it is always below a deadline $D_i(t) \geq \underline{R}_i$, i.e., such that*

$$\forall t \geq 0,\ \forall i \in \mathcal{V}(t), \quad R_i(t) \leq D_i(t). \tag{1}$$

2.2 Model of the flows

An application in the system is modeled as a *flow* indexed by some $j \in \mathcal{F}(t)$ and characterized by a *path* and an *end-to-end deadline*, properly defined next.

▶ **Definition 3** (Path). *The* path *of a flow $j \in \mathcal{F}(t)$ is defined by the sequence $p_j : \{1, \dots, \ell_j\} \to \mathcal{V}(t)$, with $\ell_j \geq 1$ being the* length of the path*, such that*

$$\forall i = 1, \dots, \ell_j - 1, \quad (p_j(i), p_j(i+1)) \in \mathcal{E}(t), \tag{2}$$

where $\mathcal{E}(t) \in \mathcal{V}(t) \times \mathcal{V}(t)$ are the current edges between the nodes of the network.

By Def. 3, it follows that $p_j(i)$ is the i-th node on the path of flow j. It should be noted that Eq. (2) enforces the existence of an edge of the graph between two consecutive nodes in a path. It should be noted that this paths may traverse a node more than once, which allows us to capture typical client-server sessions. With a slight abuse of notation, we may denote the image of the map p_j, which is the set of nodes touched by path j, by p_j only, rather than $p_j(\{1, \dots, \ell_j\})$.

We remark that the type of application addressed in this paper is borrowed from cloud microservices, where each service provides a unique value to the travelling packet. Hence, the route of packets belonging to an application is known and must not be decided at run time.

▶ **Definition 4** (End-to-end deadline). *We define the* end-to-end deadline $\mathcal{D}_j$ *of the flow $j \in \mathcal{F}(t)$ as the maximum time a packet may take to be processed by all nodes along the path p_j.*

Finally, a flow j is also characterized by an *end-to-end response-time* $\mathcal{R}_j(t)$, which is the time taken by a packet entering the first node $p_j(1)$ of the flow at time t to be processed by all nodes of the path p_j. However, before properly defining $\mathcal{R}_j(t)$, we need to introduce the *mid-path response-time* $\mathcal{R}_{j,i}(t)$, that is the time needed by a packet that entered flow j at time t to pass through the first i nodes of the path p_j. Formally, this quantity is defined recursively by

$$\mathcal{R}_{j,i}(t) = \begin{cases} R_{p_j(1)}(t) & \text{if } i = 1 \\ \mathcal{R}_{j,i-1}(t) + R_{p_j(i)}\big(t + \mathcal{R}_{j,i-1}(t)\big) & \text{otherwise.} \end{cases} \tag{3}$$

The intuition of (3) is quite straightforward: the time it takes a packet to traverse the first i nodes is equal to the time required to traverse the first $i-1$ nodes plus the response-time of the i-th node. The *end-to-end response-time* of a flow j is therefore given by $\mathcal{R}_{j,\ell_j}(t)$, which we compactly denote by $\mathcal{R}_j(t)$.

2.3 Model of the dynamic network

Flows and nodes may join and leave the network at run time. Hence, we define the network as follows.

▶ **Definition 5** (Network). *We define the* network $\mathcal{G}(t)$ *of the system at time t as the set of* nodes*,* directed edges*, and* flows *present at that time: $\mathcal{G}(t) = \{\mathcal{V}(t),\ \mathcal{E}(t),\ \mathcal{F}(t)\}$.*

Since we aim at guaranteeing end-to-end deadlines, the requests to join or leave must be properly handled by a *resource manager*, which manages the network (as illustrated in Figure 2). If not properly handled, the risk is that newly admitted flows may cause overload or the uncontrolled departure of nodes may disconnect the network.

The interactions between the resource manager and the flows are as follows:

- A flow $j \notin \mathcal{F}(t)$ may *request to join* the network. Such an instant is denoted by $f_j^{\mathsf{rq}+}$. When a new flow issues such a request, it also communicates to the resource manager the following information:
  1. its path p_j
  2. its end-to-end deadline $\mathcal{D}_j$.

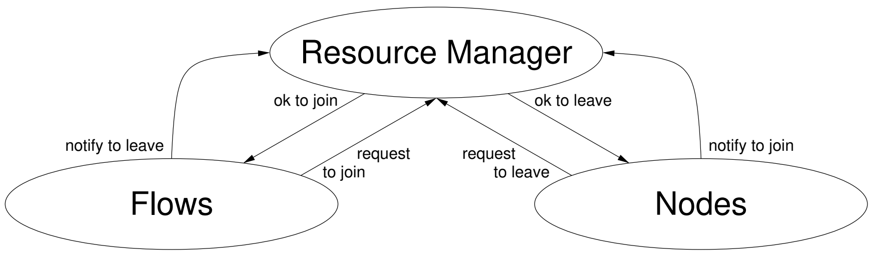


Figure 2 Scheme of interactions between the resource manager (which manages the network), the flows, and the nodes.

- After the request by a flow to join, the resource manager:
  1. accepts the flow j to the network at an instant $f_j^{\mathsf{ok}+}$ (which $\geq f_j^{\mathsf{rq}+}$), if feasible, or
  2. rejects the flow j immediately, if not feasible.

  Details on the admission of new flows based on its characteristics and the current state of the network are given in Section 4.
- A flow $j \in \mathcal{F}(t)$ may *notify and leave* the network at any time that we denote by f_j^-. In fact, it is only advantageous to let a flow (and its constraint) to leave.
- The resource manager may notify a flow $j \in \mathcal{F}(t)$ that it has to leave the network. This might, for instance, happen if a node along the path p_j requests to leave the network. In such a case, the resource manager is no longer able to provide the requested services of the flow j.

The interactions between the resource manager and the nodes (which are the vertices of the graph) are as follows:
- A node $i \notin \mathcal{V}(t)$ may *notify and join* the network at any time. We denote such an instant by v_i^+. Since a node is bringing a new service to the network, there is no admission control to its request to join.
- A node $i \in \mathcal{V}(t)$ may *request to leave* the network to the resource manager. The instant of such a request is denoted by $v_i^{\mathsf{rq}-}$.
- The resource manager lets a node leave only at time $v_i^{\mathsf{ok}-}$ (which is $\geq v_i^{\mathsf{rq}-}$). The time between $v_i^{\mathsf{rq}-}$ and $v_i^{\mathsf{ok}-}$ is needed by the resource manager to allow flows going through node i to properly exit.

Problem formulation

The problem formulation in this paper can now formally be summarized as follows;
- control when/how nodes are allowed to leave the network,
- control when/how flows are allowed to join the network,
- control the node deadlines $\mathbf{D}(t) = [D_i(t)]_{\forall i \in \mathcal{V}(t)}$,

such that the end-to-end deadlines of all the flows in the network are met:

$$\forall t \geq 0,\ \forall j \in \mathcal{F}(t), \quad \mathcal{R}_j(t) \leq \mathcal{D}_j. \tag{4}$$

3 Static networks

In this section, we introduce a protocol which allows for *dynamic deadlines* in static networks. By a static network we mean one where no nodes or flows join or leave the network, that is $\forall t \geq 0,\ \mathcal{G}(t) = \mathcal{G}$. The traffic rates and response times of nodes may change dynamically.

However, the topology is static: the nodes, the flows and their end-to-end deadlines do not change over time. Hence, we can drop the time dependency of the set $\mathcal{E}(t) = \mathcal{E}$ of edge, the set $\mathcal{V}(t) = \mathcal{V}$ of nodes, and the set $\mathcal{F}(t) = \mathcal{F}$ of flows. However, node deadlines may change to accommodate variations in the workload.

To give some intuition of the challenges of guaranteeing end-to-end deadlines in a system with dynamic node deadlines (even for a static network), we present a simple example in Section 3.1. In Section 3.2 we propose a solution, which allows for dynamic node deadlines and still provides guarantees on the end-to-end deadlines. Naturally, this is also proved in Theorem 6. Finally, in Section 3.3 we adapt the opening example of Section 3.1 such that it uses the method suggested in Section 3.2 and show that it is then able to guarantee that all the end-to-end deadlines are met for all times.

3.1 Example – issues with dynamic deadlines

If node deadlines were constant then a static assignment of node deadlines, equal to any vector of node deadlines $\mathbf{D} = \{D_i\}_{\forall i \in \mathcal{V}}$ satisfying

$$\mathbf{D} \in \overline{\mathbb{D}}(\mathcal{G}) = \left\{ D_i \in \mathbb{R}^+ : \forall i \in \mathcal{V},\ D_i \geq \underline{R}_i, \quad \forall j \in \mathcal{F},\ \sum_{i=1}^{\ell_j} D_{p_j(i)} \leq \mathcal{D}_j \right\}. \tag{5}$$

would guarantee no end-to-end deadlines to be missed. The intuition behind Eq. (5) is simple: the sum of the node deadlines along the path p_j of a flow j cannot exceed the end-to-end deadline $\mathcal{D}_j$ of the path j. We remark that the set $\overline{\mathbb{D}}(\mathcal{G})$ is convex, since it is the polytope built from the intersection among linear half-spaces. This also implies that every line between any two points in $\overline{\mathbb{D}}(\mathcal{G})$ belongs to $\overline{\mathbb{D}}(\mathcal{G})$.

Let us now consider the issues of performing a transition from some deadline assignment $\mathbf{D}(t_1) \in \overline{\mathbb{D}}(\mathcal{G})$ to another assignment $\mathbf{D}(t_2) \in \overline{\mathbb{D}}(\mathcal{G})$, hence with both the starting and ending node deadlines belonging to $\overline{\mathbb{D}}(\mathcal{G})$. Since $\overline{\mathbb{D}}(\mathcal{G})$ is convex then the linear transition of node deadlines

$$\mathbf{D}(t) = \mathbf{D}(t_1) \times \frac{t_2 - t}{t_2 - t_1} + \mathbf{D}(t_2) \times \frac{t - t_1}{t_2 - t_1}$$

always belongs to $\overline{\mathbb{D}}(\mathcal{G})$ for all $t \in [t_1, t_2]$.

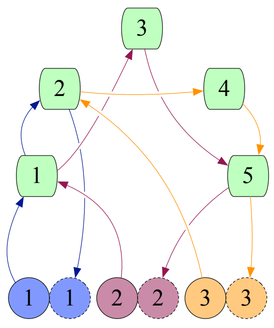


Figure 3 Example of network. Nodes are represented by light yellow boxes. Flows are represented by: a source of packets (a colored circle labelled by the flow index), a path (a sequence of arrows from the source, through the nodes), and a destination of the packets (a colored circle with dashed boundary labelled by the flow index). This example of network is used to illustrate an issue with dynamic deadlines in Section 3.1.

However, even if $\forall t \in [t_1, t_2]$, the linear combination $\mathbf{D}(t)$ always belongs to $\overline{\mathbb{D}}(\mathcal{G})$, the end-to-end deadline may be missed anyway. Suppose we have the network $\mathcal{G}$, illustrated in Figure 3 and focus on flow 1 (the blue flow), with path $p_1 = \{1, 2\}$. The end-to-end deadline for this flow is $\mathcal{D}_1 = 6$ milliseconds (ms). Suppose now that in response to an increase of the incoming packets rate, node 2 must change its deadline from $D_2(t_1) = 1$ to $D_2(t_2) = 5$. The node deadlines of the system are therefore changing from $\mathbf{D}(t_1) = [5, 1, \ldots]$ to $\mathbf{D}(t_2) = [1, 5, \ldots]$. Please, note that for the purpose of this example the particular choice of deadlines for nodes 3, 4, and 5 does not matter.

Figure 4 shows the deadlines node 1 and node 2 over time. At time $\tau_1 = 1$ a packet enters node 1, which has $D_1(\tau_1) = 5$. In the worst case, node 1 will finish processing that packet at time $\tau_1 + D_1(\tau_1) = 6$ms. Suppose now, that at time $t_1 = 2$ms, while this packet is still at node 1, the resource manager begins changing the node deadlines from $\mathbf{D}(t_1) = [5, 1, \ldots]$ towards $\mathbf{D}(t_2) = [1, 5, \ldots]$. When the packet exits node 1 at time $\tau_1 + D_1(\tau_1) = t_2 = 6$, then it enters node 2. Due to the change of node deadlines, now the value of $D_2(t)$ at $t = 6$ is $D_2(6) = 5$. This means that, in the worst case, the response time of the second node is also 5ms, since $R_2(t_2) \leq D_2(t_2) = 5$. The packet that entered flow 1 at time $\tau_1 = 1$ would therefore, in the worst case, may take up to 10ms to traverse its path p_1. Hence, the end-to-end deadline $\mathcal{D}_1 = 6$ of the packet is violated, despite the fact the sum of the node deadlines $D_1(t)$ and $D_2(t)$ along the path was never greater than $\mathcal{D}_1$ (that is $\forall t,\ D_1(t) + D_2(t) \leq \mathcal{D}_1$).

This simple example shows that when the network is allowed to change the node deadlines dynamically, the constraint $\mathbf{D}(t) \in \overline{\mathbb{D}}(\mathcal{G})$ is not a sufficient condition to guarantee end-to-end deadlines to be met. In fact, the violation of the end-to-end deadline illustrated by this example is related to the variation of the node deadlines. The node deadlines $\mathbf{D}(t)$ need to satisfy a stricter constraint, which is discussed next.

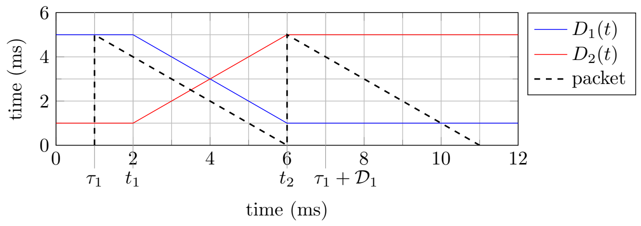


Figure 4 Example of how dynamic node deadlines may lead to end-to-end deadline violations. The node deadlines $D_1(t)$ (in blue) and $D_2(t)$ (in red) are changed from $\mathbf{D}(t_1) = [5, 1, \ldots]$ to $\mathbf{D}(t_2) = [1, 5, \ldots]$. This may lead a packet in flow 1 (with path $p_1 = \{1, 2\}$, as shown in Figure 3) to miss its end-to-end deadline $\mathcal{D}_1 = 6$.

3.2 Guaranteeing end-to-end deadlines

In this section, we provide the conditions that allow the network to dynamically change the node deadlines without incurring any end-to-end violation, as exemplified in Section 3.1. In fact, by letting node deadlines change over time, prediction of the end-to-end response time of packets becomes difficult. In Protocol 1 we present a solution to this problem. The intuition is that by limiting the rate-of-change of the node deadlines, it is possible to compute the end-to-end response time of a flow $j \in \mathcal{F}$ at time t. This allows us to compute the allowed node deadlines. This is formally proved in Theorem 6.

Protocol 1 Management of dynamic node deadlines in static networks.

- The node deadlines can never change with a rate larger than some fixed $\alpha \in [0,1]$:

$$\forall t \geq 0,\ \forall i \in \mathcal{V},\ |\dot{D}_i(t)| \leq \alpha.$$

- The node deadlines must always be within the set of *feasible node deadlines*:

$$\mathbf{D}(t) \in \mathbb{D}(\mathcal{G}) = \Big\{ D_i \in \mathbb{R}^+ : D_i \geq \underline{R}_i,\ i \in \mathcal{V},\ \forall j \in \mathcal{F},\ \sum_{i=1}^{\ell_j} (1+\alpha)^{\ell_j - i} D_{p_j(i)} \leq \mathcal{D}_j \Big\}.$$

▶ **Theorem 6** (Dynamic deadlines in static networks). *No end-to-end deadlines of any flow is violated, that is*

$$\forall t \geq 0,\ \forall j \in \mathcal{F} \quad \mathcal{R}_j(t) \leq \mathcal{D}_j, \tag{6}$$

as long as the node deadlines $\mathbf{D}(t)$ *never change with a rate faster than a given bound* $\alpha \in [0,1]$*:*

$$\forall t \geq 0,\ \forall i \in \mathcal{V}, \qquad |\dot{D}_i(t)| \leq \alpha, \tag{7}$$

and as long as the node deadlines remain within the space of feasible node deadlines:

$$\forall t \geq 0, \quad \mathbf{D}(t) \in \mathbb{D}(\mathcal{G}), \tag{8}$$

with $\mathbb{D}(\mathcal{G})$ *given by*

$$\mathbb{D}(\mathcal{G}) = \Big\{ D_i \in \mathbb{R}^+ : \forall i \in \mathcal{V},\ D_i \geq \underline{R}_i, \quad \forall j \in \mathcal{F},\ \sum_{i=1}^{\ell_j} (1+\alpha)^{\ell_j - i} D_{p_j(i)} \leq \mathcal{D}_j \Big\}. \tag{9}$$

Proof. We begin by recalling Assumption 1: the nodes in the network have a response-time controller which ensures that Eq. (1) always holds, that is $\forall t \geq 0,\ R_i(t) \leq D_i(t)$. It follows that for the first node $p_j(1)$ of any path $j \in \mathcal{F}$, it is always true that:

$$\forall t \geq 0,\ \forall j \in \mathcal{F}, \quad \mathcal{R}_{j,1}(t) = R_{p_j(1)}(t) \leq D_{p_j(1)}(t). \tag{10}$$

It then follows that for all subsequent nodes with index $i > 1$, from Eq. (3), we have that

$$\begin{aligned} \forall t \geq 0,\ \forall j \in \mathcal{F}, \quad \mathcal{R}_{j,i}(t) &= \mathcal{R}_{j,i-1}(t) + R_{p_j(i)}\big(t + \mathcal{R}_{j,i-1}(t)\big) \\ &\leq \mathcal{R}_{j,i-1}(t) + D_{p_j(i)}\big(t + \mathcal{R}_{j,i-1}(t)\big) \\ &\leq \mathcal{R}_{j,i-1}(t) + D_{p_j(i)}(t) + \alpha\, \mathcal{R}_{j,i-1}(t) \\ &= (1+\alpha)\mathcal{R}_{j,i-1}(t) + D_{p_j(i)}(t) \end{aligned} \tag{11}$$

where each step follows:
1. from the definition of end-to-end response-time (3),
2. from Eq. (1),
3. from Eq. (7), which implies Lipschitz-continuity of $D_i(t)$ with Lipschitz-constant α,
4. from basic math.

In the next step, we show

$$\forall t \geq 0,\ \forall j \in \mathcal{F}, \quad \forall x = 1, \ldots, \ell_j, \qquad \mathcal{R}_{j,x}(t) \leq \sum_{i=1}^{x} (1+\alpha)^{x-i} D_{p_j(i)}(t), \tag{12}$$

holds by proving by induction on the index x of nodes over the path j. When $x = 1$, Eq. (12) follows directly from (10). For any other $x > 1$, we have that

$$\begin{aligned}\mathcal{R}_{j,x}(t) &\leq (1+\alpha)\mathcal{R}_{j,x-1}(t) + D_{p_j(x)}(t)\\ &\leq (1+\alpha)\sum_{i=1}^{x-1}(1+\alpha)^{x-1-i}D_{p_j(i)}(t) + D_{p_j(x)}(t)\\ &= \sum_{i=1}^{x-1}(1+\alpha)^{x-i}D_{p_j(i)}(t) + D_{p_j(x)}(t)\\ &= \sum_{i=1}^{x}(1+\alpha)^{x-i}D_{p_j(i)}(t)\end{aligned}$$

where the different steps follow:
1. because the inequality is the same as Eq. (11)
2. because we exploit the inductive hypothesis of (12) for $x - 1$,
3. from basic math.

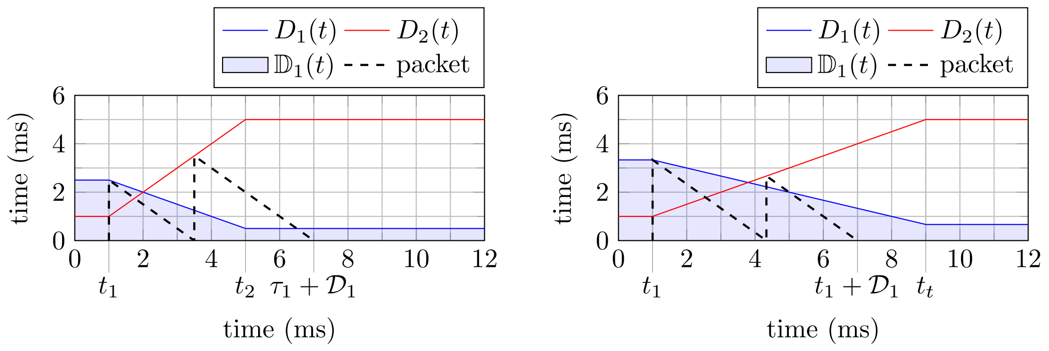
Hence, Eq. (12) is proved for all $x = 1, \ldots, \ell_j$.

Finally, we can conclude the proof by showing that

$$\forall t \geq 0,\ \forall j \in \mathcal{F} \qquad \mathcal{R}_j(t) = \mathcal{R}_{j,\ell_j}(t) \leq \sum_{i=1}^{\ell_j}(1+\alpha)^{\ell_j - i}D_{p_j(i)}(t) \leq \mathcal{D}_j$$

which implies that as long as the node deadlines are chosen such that $\mathbf{D}(t) \in \mathbf{D}(\mathcal{G})$ no end-to-end deadlines of any flow $j \in \mathcal{F}$ is violated, and the theorem is proved. ◀

3.3 Example – fixed by applying Theorem 6



(a) Maximum rate-of-change $\alpha = 1$. **(b)** Maximum rate-of-change $\alpha = 0.5$.

Figure 5 Modified example from Section 3.1. We illustrate how the system dynamically changes node deadlines over time, with different rates of change. In 5a and 5b the nodes deadlines are allowed to change with a rate of $\alpha = 1$ and $\alpha = 0.5$, respectively. Given the change of $D_2(t)$ (illustrated in red) from 1 to 5, the resource manager is only allowed to change $D_1(t)$ ensuring that it remains within the shaded blue region (denoted by $\mathbb{D}_1(t)$ and representing the space of feasible node deadlines for $D_1(t)$). Finally, as illustrated by the dashed black line, by ensuring that $D_1(t)$ remains within the blue region, the longest time it will take a packet to traverse flow 1 will be less than its end-to-end deadline.

This section illustrates that if the node deadlines are changed in accordance to Protocol 1, then the issue of end-to-end deadline misses shown in Section 3.1 cannot happen. To do so, we modify the example of Section 3.1. We consider again flow 1 of Figure 3, which has path $p_1 = \{1, 2\}$ and end-to-end deadline $\mathcal{D}_1 = 6$.

In this scenario, we assume that the resource manager must change $D_2(t)$ from $D_2(t_1) = 1$ to $D_2(t_2) = 5$. The reason is to allow node 2 to handle a sudden increase in incoming traffic. Recalling the example in Section 3.1, we can now verify that if the node deadlines begin in a state of $D_1(t_1) = 5$ and $D_2(t_1) = 1$ the resource manager is not able to change them at all. In fact, by writing explicitly the constraint of Eq. (9) in Protocol 1 for the path $j = 1$, we have

$$D_1(t_1) + (1 + \alpha)D_2(t_1) \leq \mathcal{D}_1$$
$$1 + (1 + \alpha)5 \leq 6 \qquad \Rightarrow \qquad \alpha \leq 0$$

which means that node deadlines are not allowed to change at all. If some change is needed (for example, to handle a burst of incoming traffic), then the node deadlines must be more constrained. In fact, given the change of $D_2(t)$, the choice of $D_1(t)$ has to satisfy the following constraint:

$$\forall t \geq 0, \quad (1 + \alpha)D_1(t) + D_2(t) \leq \mathcal{D}_1 = 6. \tag{13}$$

Next, we compare two cases of different values of feasible rate of change α for the node deadlines. The result are illustrated in Figure 5.

Maximum rate-of-change $\alpha = 1$

To allow the network to transition quickly from $D_2(t_1) = 1$ to $D_2(t_2) = 5$ we choose $\alpha = 1$. This means that the the choices of D_1 are constrained by the following condition:

$$\forall t \geq 0, \quad (1 + 1)D_1(t) + D_2(t) \leq \mathcal{D}_1 = 6.$$

This condition gives the resource manager the space of possible choices for $D_1(t)$ illustrated by the shaded blue region in Figure 5a. The largest possible choices for $D_1(t)$ therefore involves changing D_1 from $D_1(t_1) = 2.5$ to $D_1(t_2) = 0.5$, with $t_1 = 1$ and $t_2 = 5$.

Maximum rate-of-change $\alpha = 0.5$

The stringency of the constraint on the node deadlines when $\alpha = 1$ may be relaxed by requiring a slower transition between the node deadlines, for example $\alpha = 0.5$. By doing so, we get the following conditions on $D_1(t)$:

$$D_1(t_1) \leq \frac{10}{3} \approx 3.333, \qquad D_1(t_2) \leq \frac{2}{3} \approx 0.666,$$

in order to ensure that Eq. (13) holds, and assuming that $D_2(t)$ is again changed from $D_2(t_1) = 1$ to $D_2(t_2) = 5$. Not surprisingly, by requiring a smoother transition, the node deadlines may be larger. Similarly to the previous case, this is illustrated in Figure 5b, where we illustrate that when $D_1(t)$ is chosen to be as large as possible, a packet traversing flow 1 is always guaranteed to meet its end-to-end deadline of 6ms.

Trade-offs with alpha

This example illustrates some fundamental trade-offs that come by adopting Protocol 1. While it allows the system to change the node deadlines dynamically with a rate of α, it imposes some constraints on the possible choices of node deadlines. The quicker one wishes to change the node deadlines, the more restricted the choice of feasible node deadlines becomes, and vice versa. The design-parameter α should be chosen appropriately for a given application.

4 Dynamic networks

In this section, we generalize the method presented Section 3 to the case of a *dynamic network* $\mathcal{G}(t) = \{\mathcal{V}(t),\ \mathcal{E}(t),\ \mathcal{F}(t)\}$, where nodes and flows may join and leave the network at run time. We begin by showing, in Corollary 7, that the result of Theorem 6 transfers directly to a dynamic network. This means that the end-to-end deadline of any flow in the network will be met as long as the hypothesis of Theorem 6 hold for all states of the network $\mathcal{G}(t)$ at all times t. Conditions stated in Corollary 7 are fulfilled. By comparing Theorem 6 and Corollary 7 one can see that the only difference is that we now allow for a dynamic network, i.e., $\mathcal{G}(t)$ instead of a static network $\mathcal{G}$.

▶ **Corollary 7** (Dynamic deadlines in dynamic networks). *The end-to-end deadline of all flows are always met, at all times, that is:*

$$\forall t \geq 0,\ \forall j \in \mathcal{F}(t) \quad \mathcal{R}_j(t) \leq \mathcal{D}_j, \tag{14}$$

as long as the node deadlines $\mathbf{D}(t)$ *never change with a rate faster than some* $\alpha \in [0,1]$*:*

$$\forall t \geq 0,\ \forall i \in \mathcal{V}(t), \qquad |\dot{D}_i(t)| \leq \alpha, \tag{15}$$

and as long as the node deadlines belong to the space of feasible node deadlines:

$$\forall t \geq 0, \quad \mathbf{D}(t) \in \mathbb{D}(\mathcal{G}(t)), \tag{16}$$

with $\mathbb{D}(\mathcal{G}(t))$ *given by*

$$\mathbb{D}\big(\mathcal{G}(t)\big) = \Big\{ D_i \in \mathbb{R}^+ : \forall i \in \mathcal{V}(t),\ D_i \geq \underline{R}_i, \quad \forall j \in \mathcal{F}(t),\ \sum_{i=1}^{\ell_j} (1+\alpha)^{\ell_j - i} D_{p_j(i)} \leq \mathcal{D}_j \Big\}. \tag{17}$$

Proof. We begin the proof by observing that from Eq. (15), it follows directly from Theorem 6 that for a fixed time-instance t' it holds that

$$\forall j \in \mathcal{F}(t'), \quad \mathcal{R}_j(t') \leq \mathcal{D}_j, \tag{18}$$

as long as

$$\mathbf{D}(t') \in \mathbb{D}\big(\mathcal{G}(t')\big), \tag{19}$$

with $\mathbb{D}\big(\mathcal{G}(t')\big)$ given by Eq. (17). In fact, this is precisely what was stated and proved in Theorem 6, but with a fixed network topology $\mathcal{G}$, instead of an instantaneous "snapshot" $\mathcal{G}(t')$ of a dynamic topology.

Then, the hypothesis of Eq. (16) ensures that Eq. (19) holds $\forall t \geq 0$. Therefore, it follows that Eq. (18) holds $\forall t \geq 0$, and in turn that Eq. (14) does always hold, as required. ◀

As demonstrated by the short proof, Corollary 7 does not poses any deeper conceptual challenges compared to Theorem 6. However, the two hypothesis of (15) and (16) may be hard to hold simultaneously, if no special care is taken. This is illustrated in the next example.

Example – issues in acceptance a new flow

The blind admission of new flows as soon as they request to join, may cause the violation of one of the two hypothesis of the corollary (Equations (15) and (16)) making then Corollary 7 incapable to guarantee end-to-end deadlines. At the time when any new flow j' is admitted to the network, the set of flows $\mathcal{F}(t)$ includes the new flow j' which was not previously in the set. As a consequence of the acceptance of the new flow j' into $\mathcal{F}(t)$, the set of feasible node deadlines $\mathbb{D}\big(\mathcal{G}(t)\big)$ may suddenly shrink due to the newly added constraint. As illustrated in the next example, this might in turn cause the node deadlines to be in an infeasible state, such that the end-to-end deadline of the newly accepted flow will be violated. Notice that node deadlines **cannot** instantaneously adapt to the new constraint, otherwise the hypothesis of "bounded rate of change" of Eq. (15) is violated.

In Figure 6, we illustrate a system where a new flow j' registers to join the network at time $f_{j'}^{\mathsf{rq}+} = 3$. As soon as this flow is accepted into the network, at time $f_{j'}^{\mathsf{ok}+} = 4.5$, the set of feasible node deadlines (illustrated by the shaded green area) make a discrete jump. This means that the node deadlines $\mathbf{D}(t)$ (black thick line in Figure 6) will no longer remain within $\mathbb{D}\big(\mathcal{G}(t)\big)$. In other words this means that $\mathbf{D}(f_{j'}^{\mathsf{ok}+}) \notin \mathbb{D}\big(\mathcal{G}(f_{j'}^{\mathsf{ok}+})\big)$, which is a clear violation of the conditions of Corollary 7.

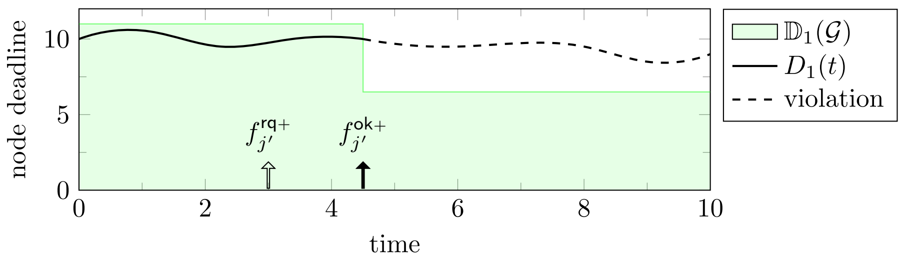


Figure 6 Illustration of why it might be difficult to ensure that $\forall t \geq 0,\ \mathbf{D}(t) \in \mathbb{D}\big(\mathcal{G}(t)\big)$ and $|\dot{D}_i(t)| \leq \alpha$. In this scenario, the space of feasible node deadline choices for the first node, $\mathbb{D}_1\big(\mathcal{G}(t)\big)$ (shaded green area), makes a discrete jump as soon as the new flow j' is accepted into the network. The reason is that the inclusion of the new flow adds a new end-to-end deadline constraints. This, in turn, may cause the node deadlines to instantaneouslt become infeasible."leave" the space of feasible node deadlines (illustrated by the dashed line).

The illustrated issue suggests that a special care must be taken when the network $\mathcal{G}(t)$ is modified, since the discrete variation of the network may not be compatible with the need of smooth node deadlines. Therefore, in Section 4.1, we present two protocols which address this issue:

- Protocol 2 explains how the requests of flows are managed, while
- Protocol 3 illustrates the management of nodes of the network.

4.1 Protocol allowing dynamic networks

In this section, we present a protocol for managing how flows may join and leave the network. We show that by following this protocol, the hypothesis of Corollary 7 do always hold, making then the corollary applicable. We then present a second protocol for how to manage nodes joining and leaving the network.

Management of flows

The intuition behind Protocol 2, which manages the flows, comes from the fact that as soon as a new flow j' is accepted into the network, at time $f_{j'}^{\mathsf{ok}+}$, the network changes from $\mathcal{G} = \{\mathcal{V}(t), \mathcal{E}(t), \mathcal{F}(t)\}$ to $\mathcal{G}^+ = \big\{\mathcal{V}(t), \mathcal{E}(t), \mathcal{F}(t) \cup \{j'\}\big\}$. The constraint corresponding to the new flow j' is thus added to $\mathbb{D}(\mathcal{G})$ leading to the new set of feasible node deadlines $\mathbb{D}(\mathcal{G}^+) \subseteq \mathbb{D}(\mathcal{G})$. Therefore, in order to ensure that the node deadlines remain within the set of feasible node deadlines once j' is accepted, i.e., that $\mathbf{D}(f_{j'}^{\mathsf{ok}+}) \in \mathbb{D}\big(\mathcal{G}(f_{j'}^{\mathsf{ok}+})\big)$, Protocol 2 will only accept j' if $\mathbf{D}(t) \in \mathbb{D}(\mathcal{G}^+)$. If this is the case when j' requests to join, i.e., that $\mathbf{D}(f_{j'}^{\mathsf{rq}+}) \in \mathbb{D}(\mathcal{G}^+)$, then the new flow will be accepted immediately, and $f_{j'}^{\mathsf{ok}+} = f_{j'}^{\mathsf{rq}+}$.

If on the other hand, $\mathbf{D}(f_{j'}^{\mathsf{rq}+}) \notin \mathbb{D}(\mathcal{G}^+)$, it is not possible to accept j' immediately. Therefore, in order to accept it, the resource manager changes the node deadlines towards a *goal point* $\mathbf{D}^* \in \mathbb{D}(\mathcal{G}^+)$. It will then accept the new flow j', once $\mathbf{D}(t) = \mathbf{D}^*$, which will occur at time $f_{j'}^{\mathsf{ok}+}$, given by Eq. (20).

Protocol 2 Management of flows.

- At time $f_{j'}^{\mathsf{rq}+}$, the new flow j' requests to join
- If $\mathbf{D}(f_{j'}^{\mathsf{rq}+}) \in \mathbb{D}(\mathcal{G}^+)$, then the flow j' is admitted immediately, that is $f_{j'}^{\mathsf{ok}+} = f_{j'}^{\mathsf{rq}+}$.
- If $\mathbf{D}(f_{j'}^{\mathsf{rq}+}) \notin \mathbb{D}(\mathcal{G}^+)$, then the admission of flow j' is delayed until the node deadlines have completed a linear transition to a *goal point* $\mathbf{D}^* \in \mathbb{D}(\mathcal{G}^+)$, that is

$$f_{j'}^{\mathsf{ok}+} = f_{j'}^{\mathsf{rq}+} + \frac{\max_{i\in\mathcal{V}(t)} \left|D_i^* - D_i(f_{j'}^{\mathsf{rq}+})\right|}{\alpha}. \tag{20}$$

  with α being the maximum feasible rate of change of node deadlines.
- If $\mathbf{D}\big(\mathcal{G}^+)\big) = \varnothing$, then the request of flow j' to join the network is rejected.
- A flow $j \in \mathcal{F}(t)$ may notify the resource manager and leave the network at any time. The time when it leaves is denoted by f_j^-.

Let us comment on the choice of the "goal point" $\mathbf{D}^*$. In general, any choice of $\mathbf{D}^* \in \mathbb{D}(\mathcal{G}^+)$ is valid. However, if the target is to minimize the transition-time fron the time $f_{j'}^{\mathsf{rq}+}$ flow j' request to join to the time $f_{j'}^{\mathsf{ok}+}$ it is admitted to the network (given by Eq. (20)), then it should be chosen as

$$\mathbf{D}^* = \operatorname*{arg\,min}_{\mathbf{D}\in\mathcal{G}^+} \max_i \left|D_i - D_i(f_{j'}^{\mathsf{rq}+})\right|.$$

We would like to point out three observations from Protocol 2. The first one is that during the transition to accept the new flow j' the node deadlines are changed according to the following linear function:

$$\forall i \in \mathcal{V}(f_{j'}^{\mathsf{rq}+}),\ \forall t \in [f_{j'}^{\mathsf{rq}+}, f_{j'}^{\mathsf{ok}+}], \quad D_i(t) = \frac{D_i(f_{j'}^{\mathsf{rq}+}) \times (t - f_{j'}^{\mathsf{rq}+})}{f_{j'}^{\mathsf{ok}+} - f_{j'}^{\mathsf{rq}+}} + \frac{D_i^* \times (f_{j'}^{\mathsf{ok}+} - t)}{f_{j'}^{\mathsf{ok}+} - f_{j'}^{\mathsf{rq}+}}. \tag{21}$$

This means that the node deadlines are changed along a line from $\mathbf{D}(f_{j'}^{\mathsf{rq}+})$ to $\mathbf{D}(f_{j'}^{\mathsf{ok}+})$. Since $\mathbb{D}(\mathcal{G})$ is a convex space, it follows that $\mathbf{D}(t) \in \mathbb{D}(\mathcal{G})$ during the transition. Hence, it also follows that the hypothesis of Eq. (16) in Corollary 7 holds during the transition.

The second observation is that by changing the node deadlines according to Eq. (21) we acquire the property that $D_i(f_{j'}^{\mathsf{ok}+}) = D_i^*$. This means that at the end of the transition, at time $t = f_{j'}^{\mathsf{ok}+}$, we have $\mathbf{D}(t) \in \mathbb{D}(\mathcal{G}^+)$. This implies that once the new flow j' is accepted condition (16) of Corollary 7 continues to hold.

The final observation is that by exploiting the value of $f_{j'}^{\mathsf{ok}+}$ of Eq. (20), we have

$$|\dot{D}_i(t)| = \frac{|D_i(f_{j'}^{\mathsf{rq}+}) - D_i^*|}{f_{j'}^{\mathsf{ok}+} - f_{j'}^{\mathsf{rq}+}} = \alpha \frac{|D_i(f_{j'}^{\mathsf{rq}+}) - D_i^*|}{\max_{i\in\mathcal{V}(t)} \left|D_i^* - D_i(f_{j'}^{\mathsf{rq}+})\right|} \le \alpha \tag{22}$$

This implies that Protocol 2 fulfills condition (15) of Corollary 7, since the maximum rate-of-change of any node is α during the transition.

By combining all three observations, we can conclude that by following Protocol 2, the hypotheses of Corollary 7 are always met when accepting new flows, and then no end-to-end deadline is violated.

Finally, we remark that if there is no possible choice of a goal point, i.e., if $\mathbb{D}(\mathcal{G}^+) = \varnothing$, then the request of j' to join is clearly rejected because the admission of the new flow j' may cause the violation of end-to-end deadline of some flow already admitted to the network.

Management of nodes

The basic idea behind Protocol 3 is that when a node $i \in \mathcal{V}(t)$ leaves the network, it does affect the flows with a path going through i. Therefore, at time $v_i^{\mathsf{rq}-}$, that is when node i requests to leave the network, the resource manager notifies any affected flow $j \in \{j \ : \ \forall j \in \mathcal{F}(t), \ i \in p_j\}$ that it will be pushed out from the network after a time T_i^{leave}. The rationale for this is simply that the resource manager will no longer be able to provide any guarantees for the end-to-end deadlines of the affected flows once node i has left the network.

However, should the affected flows still wish to use some of the services in the network, they may request to re-join the network as a new flow. In order to allow the affected flows adequate time to do this, and to also ensure that there is enough time for the packets in the affected flows, we require T_i^{leave} to be greater than the largest end-to-end deadline of the affected flows.

Moreover, it can be noticed that there is no condition on the nodes willing to join the network.

Protocol 3 Management of nodes.

- A node $i \notin \mathcal{V}(t)$ may notify the resource manager and join the network at any time v_i^+.
- When a node $i \in \mathcal{V}(t)$ requests to leave the network (we denote such an instant by $v_i^{\mathsf{rq}-}$), it is allowed to do so at:

  $$v_i^{\mathsf{ok}-} \geq v_i^{\mathsf{rq}-} + T_i^{\text{leave}}$$

  with $T_i^{\text{leave}} \geq \max\{\mathcal{D}_j \ : \ \forall j \in \mathcal{F}(t), \ i \in p_j\}$
- The resource manager will notify all the affected flows $j \in \{j \ : \ \forall j \in \mathcal{F}(t), \ i \in p_j\}$ that they will be kicked out at time $t = v_i^{\mathsf{ok}-}$ if they have not left the network by then.

Comments on handling multiple flows

It should be noted that while Protocol 2 only treats the case where a single flow j' request to join, it is possible to allow multiple flows to request. The management of this scenario can be achieved by introducing a *request queue*, and then applying Protocol 2 for the request at the head of the queue. Once the request is served, the resource manager can then repeat for the new head-of-the-queue request until there are no more pending requests. This methodology would only incur in a heavier notation which we prefer not to add to lighten the presentation.

Moreover, as we show in the experiments of Section 5.2 a new flow can be accepted in matter of milliseconds, or even micro seconds. Therefore, given the application of cloud robotics, it is fair to assume that there will not be multiple flows requesting to join simultaneously. And should there be, introducing a request queue will not incur any significant delay for accepting new flow.

4.2 Example – dynamic network topology

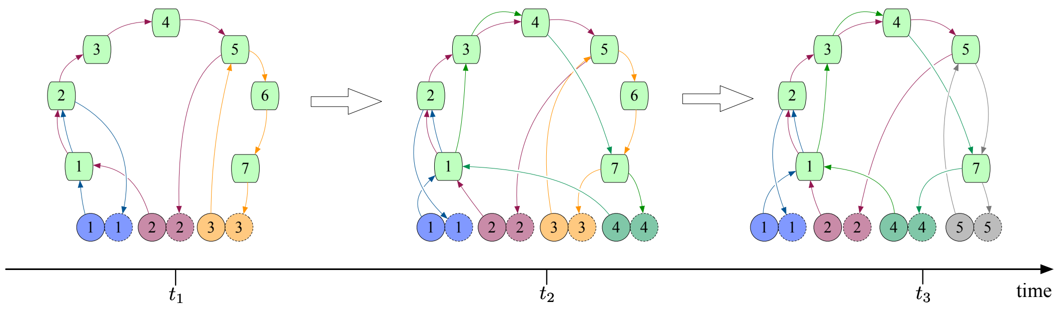


Figure 7 Illustration of how the network changes in the example of Section 4.2. In the first transition, from time t_1 to t_2, flow, 4 (green), joins the network, which already has flows 1, 2, and 3. Then in the second transition, from time t_2 to t_3, node 6 leaves the network. When node 6 leaves the network, it affects flow 3 (yellow), which then leaves, and re-join the network as a new flow 5 (gray).

In this section, by using some examples, we illustrate how the protocols between the flows, nodes, and the resource manager work.

The scenario is illustrated in Figure 7 and consists of the two transitions. In the first transition, from t_1 to t_2, flow 4 (green) requests to join the network. The second transition, from t_2 to t_3 illustrates how node 6 requests to leave the network. By doing so, it will affect flow 3, which has a path passing through node $i = 6$. The affected flow must therefore leave the network, and re-join as a new flow $j = 6$, illustrated by the gray arrows in Figure 7.

Next, we describe the first transition (of the new flow joining), followed by the second transition (node 6 leaving). The schedule for both transitions are depicted in Figure 8.

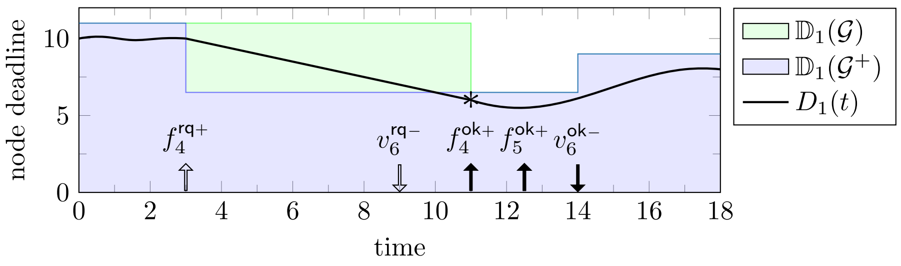


Figure 8 Illustration of how the space of feasible node deadlines changes when new flows are accepted into the network, as well as when nodes leave. It show a request from flow $j' = 4$ to join the network at time $f_4^{\mathsf{rq}+} = 3$ as well as a request from node $i = 6$ to leave the network at time $v_6^{\mathsf{rq}-} = 9$. It also illustrates how the resource manager changes the node deadlines towards $\mathbf{D}^*$ (given by $*$) before accepting the new flow $j' = 4$.

Request of a flow to join

When the new flow 4 requests to join the network, at time $f_4^{\mathsf{rq}+} = 3$, the node deadlines of the system are in a state such that it cannot be accepted immediately, i.e., $\mathbf{D}(f_4^{\mathsf{rq}+}) \notin \mathbb{D}(\mathcal{G}^+)$. According to Protocol 2, this requires the resource manager to change the node deadlines to a goal point $\mathbf{D}^* \in \mathbb{D}(\mathcal{G}^+)$. The node deadlines will therefore be changed linearly from $\mathbf{D}(f_4^{\mathsf{rq}+})$ to $\mathbf{D}^*$. At time $f_4^{\mathsf{ok}+}$, we have that $\mathbf{D}(t) = \mathbf{D}^*$, and then flow 4 is admitted into the network. This is illustrated in Figure 7 with $\mathbb{D}(\mathcal{G})$ shown as the shaded green region, $\mathbb{D}(\mathcal{G}^+)$ as the shaded blue region, and the goal point $\mathbf{D}^*$ as the $*$ symbol.

Request of a node to leave

At time $v_6^{\mathsf{rq}-} = 9$, node 6 requests to leave the network. This is illustrated in Figure 8 by the downward arrow. Since node 6 belongs to the path p_3 of flow 3 (see Figure 7), the departure of node 6 would affect flow 3 (yellow), with end-to-end deadline $\mathcal{D}_3 = 5$. By following Protocol 3, the resource manager will therefore notify flow 3 that it will no longer provide any end-to-end deadline guarantees after a time

$$v_6^{\mathsf{ok}-} = v_6^{\mathsf{rq}-} + T_6^{\mathrm{leave}} = 9 + 5 = 14.$$

The node will then be allowed to leave the network at this time $v_6^{\mathsf{ok}-}$, as illustrated in Figure 8. In the figure, it can also be noticed that the space of feasible node deadlines increases when node 6 leaves. The reason is that constraint of the end-to-end deadline $\mathcal{D}_3$ of flow 3 is removed.

In this example, we assume that flow 3 still wants to remain in the network. Therefore, it will request to re-join the network as a new flow 5 at time $f_5^{\mathsf{rq}+}$. At this time, the node deadlines already allow the requesting flow 5 to join (that is $\mathbf{D}(f_5^{\mathsf{rq}+}) \in \mathbb{D}(\mathcal{G}^+)$) and then flow 5 is admitted immediately ($f_5^{\mathsf{ok}+} = f_5^{\mathsf{rq}+}$), as shown in Figure 8.

5 Evaluation: trade-offs with alpha

In this section we evaluate some of the effects introducing Protocols 1, 2, and 3 might have on a system. We are particularly concerned with the trade-offs of choosing different values of the design-parameter α. The intuition is that by choosing a value for α we choose how quickly the resource manager is able to change the node deadlines in the network. A higher value of α will therefore allow for quicker changes. This will in allow the resource manager to accept new flows faster. However, as illustrated in Section 3, a higher value of α will require lower node deadlines. This means that the response-times of the nodes in the network have to be lower. In order to satisfy this, some response-time controllers might have to sacrifice the quality of service (QoS) provided by the nodes. For instance, in [17] the sacrifice is to sometimes discard packets, and in [18] the sacrifice is to decrease the amount of content provided by the nodes.

5.1 System used for evaluation

The system used to evaluate the trade-offs of α is presented in a previous work [17] by the authors. In short, it allows nodes in a network, such as the one illustrated in Figure 9, to ensure that the response-time is less than a specific node deadline. The way it does this is by combining admission control and service control in every node.

The goal of the *service controller* is to ensure there is adequate processing capacity in the nodes. It does so by dynamically scaling the processing capacity according to a control-law. However, since the system is targeting a cloud-environment, the incoming traffic may be very dynamic. Moreover, the amount of processing capacity provided to the nodes may vary over time (i.e., servers might crash, etc.).

The goal of the *admission controller* is to always ensure that the response-time of the node is less than the node deadline. If there is sufficient processing capacity to serve the incoming traffic, it will not have to discard any packets. However, should a node find itself in a situation where there is not sufficient processing capacity to meet the incoming traffic, then the admission controller will have discard packets in order to guarantee that the response-time of the node will be less than the node deadline.

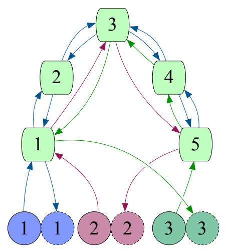


Figure 9 Illustration of the network used to evaluate the trade-offs with α and the time taken to reach the goal point $\mathbf{D}^*$ (in Section 5.2) as well as between α and the system performance (in Section 5.3).

5.2 Trade-off: alpha and time to accept a new flow

To evaluate the impact of α on the time required to accept a new flow, we will use the system presented in Section 5.1 and evaluate how long it takes the system to reach different goal points $\mathbf{D}^* \in \mathbb{D}(\mathcal{G}^+)$. Using a network with 5 nodes and 3 flows, we used the following set-up:

1. Choose the order-of-magnitude for the end-to-end deadlines $\overline{\mathcal{D}} \in \{0.1,\ 1,\ 10\}$ (milliseconds) as well as a value for $\alpha \in [10^{-3},\ 10^{0}]$.
2. Assign a randomly generated end-to-end deadline to each flow, $\mathcal{D}_j \in \mathcal{U}(0.7 \cdot \overline{\mathcal{D}},\ 1.3 \cdot \overline{\mathcal{D}})$. Note that $\mathcal{D}_j$ is drawn from a uniform distribution.
3. Chose the *goal point* $\mathbf{D}^*$ as the solution to the *optimal node-deadline problem*, presented in [17], but adapted with the constraints of Eq. (16):

$$\begin{aligned} &\text{minimize} && \sum_{i \in \mathcal{V}(t)} 1/D_i && \\ &\text{subject to} && \sum_{i=1}^{\ell_j} (1+\alpha)^{\ell_j - i} D_{p_j(i)} \le \mathcal{D}_j && \forall j \in \mathcal{F}(t) \\ & && D_i \ge 0 && \forall i \in \mathcal{V}(t) \end{aligned} \tag{23}$$

4. Simulate how long it takes the network to reach the desired goal point $\mathbf{D}^*$.
5. Repeated steps 2 thru 4 for 100 simulations.
6. From the 100 simulations, compute the *average time* to reach a goal point $\mathbf{D}^*$.
7. Repeat steps 1 thru 6 for a different choices of $\overline{\mathcal{D}}$ and α.

The result of the evaluation is shown in Figure 10. As expected, it shows a clear relationship between α and the time needed to reach $\mathbf{D}^*$. It is interesting to note that even when having end-to-end deadlines in the order of 10ms, and a very small α the resource manager is still able to reach $\mathbf{D}^*$ in less than a second. By allowing a higher α, it is possible to reach $\mathbf{D}^*$ in less than a microsecond.

5.3 Trade-off: alpha and quality of service

To evaluate how different choices of α affect the QoS provided by the nodes in the network we will again use the system briefly presented in Section 5.1. As mentioned there, the system used an *admission controller* and a *service controller* to ensure that the response-time of the nodes always remained less than their node deadlines.

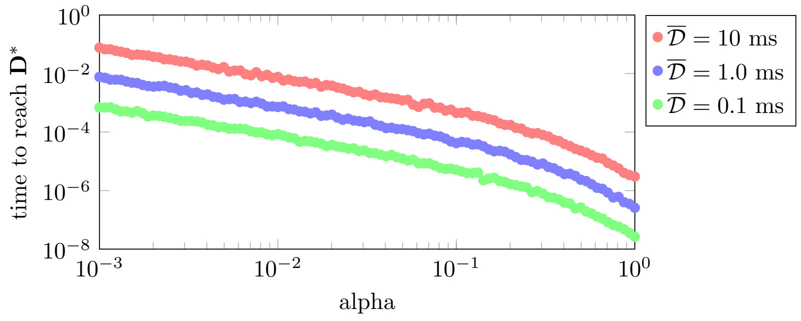


Figure 10 Simulation result of how long takes a system (with the network depicted in Figure 9) to reach a goal-point $\mathbf{D}^*$. It shows how this time depend on both the choice of α as well as how large the end-to-end deadlines of the systems are.

Due to the *uncertainties* of the available processing capacities, and since the amount of traffic going through the flows is highly varying, the admission controller sometimes have to discard packets. This is what we define as the QoS, in other words

$$\text{QoS} = 1 - \rho,$$

where ρ is the fraction of packets which are dropped.

The evaluation was performed using a network with 5 nodes and 3 flows, together with the following set-up:

1. Choose a *processing uncertainty* $\bar{\xi} \in \{0.05,\ 0.1,\ 0.2,\ 0.4\}$ and a value of $\alpha \in [10^{-3},\ 10^0]$..
2. Generate traffic based on data from the Swedish University Network (SUNET) and simulate the system for 20 seconds. A typical traffic pattern is illustrated in Figure 11a.
3. Compute the QoS for the simulation.
4. Repeat steps 2 and 3 for another 100 simulations.
5. Compute the *average* QoS for this choice of $\bar{\xi}$ and α.
6. Repeat steps 1 thru 5 for a new choice of $\bar{\xi}$ and α.

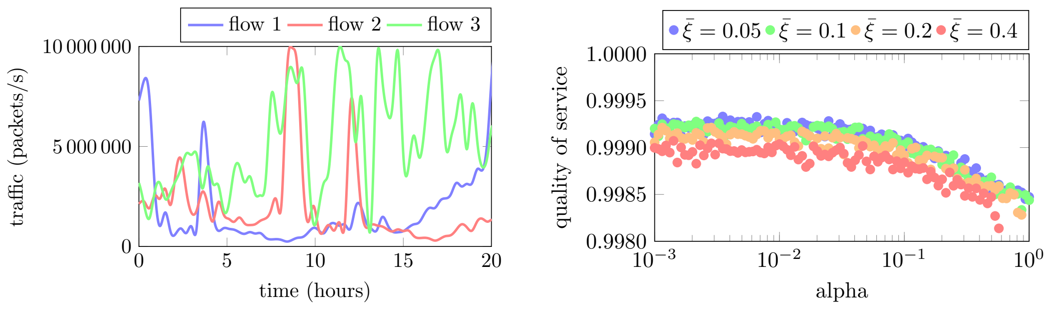
Some comments on the processing uncertainty above is that if $\bar{\xi} = 0.2$ a node in the network might believe it has a processing capacity of 1000 packets per second (pps), but in reality it could only handle 800 pps. Therefore, the higher $\bar{\xi}$, the higher the probability is that the node has a lack of available processing capacity.

The result of the evaluation, presented in Figure 11b, where the y-axis show the quality of service and the x-axis show the values of α. The different colors highlight the different bounds on the uncertainty for the processing capacity. An interesting observation is that the QoS does not depend so much on the uncertainty $\bar{\xi}$ as it does on the choice of α. As expected, when α increases, the QoS goes down. However, even for large values of α, the QoS remains fairly high, i.e., above 0.9980. This means that 99.8% of all the packets make it through the system on time.

6 Related work

Despite the fact that the addressed problem in this paper comes from very recent technology advancements (e.g. cloud computing and 5G), it is possible to abstract it in a way where related results can be found over quite vast a spectrum of older contributions. In an abstract way, the problem presented in this paper can be decomposed into the following sub-components:

- providing end-to-end deadline guarantees for flows in a network,
- splitting end-to-end deadlines into local deadlines.



(a) Traffic for one of the simulations. **(b)** Evaluation of α and the quality-of-service.

Figure 11 Simulation to evaluate how α affect QoS of the system. It highlights that despite a highly varying traffic going through the flows (as depicted in Figure 11a) the QoS remains high, even for a value of α close to 1. In fact, it shows that for $\alpha = 0.1$ about 99.8% of all the packets made it through the system and met their end-to-end deadlines.

However, to the best of our knowledge, no work has considered all of these sub-components together in the context of a dynamic network topology.

A considerable amount of previous works addresses the deadline guarantee of a sequence of jobs that needs to be processed at a given node of a network [24, 20, 21]. Within each node, jobs are scheduled by any single processor scheduling policy (FP, EDF, or else). The communication between nodes is modelled by propagating the jitter [24, 20] or the offset [21] of the task execution within a node. Gerber et al. [5] proposed an alternate method to translate end-to-end deadlines over a directed graph of nodes into constraints on the activation periods of the tasks running at the intermediate nodes.

In the context of compositional analysis, previous works have addressed the problem of isolating and composing a single flow over a network of nodes. Lorente et al. [14] extended the holistic analysis to the case with nodes running at a fraction of computing capacity (abstracted by a bounded-delay time partition with bandwidth and delay). Jayachandran and Abdelzaher [10] developed several transformations ("delay composition algebra") to reduce the analysis of a distributed system to the single processor case. Serreli et al. [22] proposed a component interface for chains of tasks activated sporadically and an intermediate deadline assignment, which minimises the requested computing capacity. Similarly, Ashjaei et al. [1] proposed resource reservation over each node along the path.

In the context of computation happening at "small" scale, it is worth mentioning the modular analysis by Hamann, Jersak, Richter, Ernst [6]. Such a modular analysis, which found an application in the automotive domain, may well be a source of inspiration to analyze the schedulability within each node and the interaction between nodes. It is, however, orthogonal to our method which focuses on the policies to allow new flows of packets ("event streams" in the terminology of [6]) to be admitted at run time. Also, network calculus [13] and real-time calculus [23] are excellent orthogonal methods to analyze the schedulability within nodes as well as theier interactions.

The idea of breaking end-to-end deadlines in local deadlines was also exploited by several authors. Di Natale and Stankovic [2] proposed to split the end-to-end deadline proportionally to the local computation time or to divide equally the slack time. Marinca et al. [15] proposed two methods to assign local deadlines ("Fair Laxity Distribution" and "Unfair Laxity Distribution") to balance the distribution of the slack among the flows. Later, Jiang [11] used time slices to decouple the schedulability analysis of each node, reducing the complexity of the analysis. More recently, Hong et al. [8] formulated the local deadline assignment problem as a Mixed-Integer Linear Program (MILP) with the goal of maximising the slack time. The number of local deadlines, however, is very high and makes the resulting optimisation problem hard to solve. Jabob et al. [9] proposed to split among local deadlines by using a *deadline ratio* $\rho \in (0, 1)$ configuration parameter chosen at design-time.

Related, but orthogonal to the presented research is the problem of mapping the flows of packets onto the available processing nodes. In the automotive context, Zhu et al. [26] formulated a MILP problem to find a task mapping that minimises the sum of a set of sensitive latencies. Garibay-Martínez et al. [4] used heuristics to partition tasks and assigned priorities to tasks sharing the same resource.

7 Conclusion and future works

In this work, we presented a framework, which allows applications and cloud-services to dynamically join and leave a system over time. The intuition is that by assigning and controlling how quickly local deadlines of cloud-services may change, it is possible to guarantee the end-to-end deadlines of the applications in presence of flows and nodes dynamically leaving and joining the network. Finally, with extensive simulations we are able to show that with the suggested protocols it is possible to accept new applications in matter of milliseconds. Moreover, we show that the constraints of the protocols does not affect the quality-of-service in a significant way.

This preliminary work opens for many research directions. Among them we mention:

impact of node policies How can the end-to-end response time benefit from per-flow packet scheduling policies within the nodes? For example, fixed priorities, EDF, etc.

decentralized protocol Can the framework be implemented in a decentralized way by exploiting per-node information rather than using the full knowledge, as in this paper?

References

1 Mohammad Ashjaei, Saad Mubeen, Moris Behnam, Luis Almeida, and Thomas Nolte. End-to-End Resource Reservations in Distributed Embedded Systems. In *22nd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–11, August 2016.

2 Marco Di Natale and John A. Stankovic. Dynamic End-to-end Guarantees in Distributed Real Time Systems. In *Proceedings of the 15th IEEE Real-Time Systems Symposium*, pages 215–227, December 1994.

3 Neha Gandhi, Dawn M Tilbury, Yixin Diao, J Hellerstein, and Sujay Parekh. Mimo control of an apache web server: Modeling and controller design. In *American Control Conference, 2002. Proceedings of the 2002*, volume 6, pages 4922–4927. IEEE, 2002.

4 Ricardo Garibay-Martínez, Geoffrey Nelissen, Luis Lino Ferreira, and Luis Miguel Pinho. Task partitioning and priority assignment for distributed hard real-time systems. *Journal of Computer and System Sciences*, 81(8):1542–1555, 2015.

5 Richard Gerber, Seongsoo Hong, and Manas Saksena. Guaranteeing Real-Time Requirements with Resource-based Calibration of Periodic Processes. *IEEE Transaction on Software Engineering*, 21(7):579–592, July 1995.

6 Arne Hamann, Marek Jersak, Kai Richter, and Rolf Ernst. A framework for modular analysis and exploration of heterogeneous embedded systems. *Real-Time Systems*, 33(1):101–137, July 2006. `doi:10.1007/s11241-006-6884-x`.

7 Dan Henriksson, Ying Lu, and Tarek Abdelzaher. Improved prediction for web server delay control. In *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 61–68. IEEE, 2004.

8 Shengyan Hong, Thidapat Chantem, and Xiaobo Sharon Hu. Local-deadline assignment for distributed real-time systems. *IEEE Transactions on Computers*, 64(7):1983–1997, 2015. doi:10.1109/TC.2014.2349494.

9 Romain Jacob, Marco Zimmerling, Pengcheng Huang, Jan Beutel, and Lothar Thiele. End-to-end real-time guarantees in wireless cyber-physical systems. In *Proceedings 2016 IEEE Real-Time Systems Symposium. RTSS 2016*, pages 167–178. IEEE, 2016.

10 Praveen Jayachandran and Tarek Abdelzaher. Delay Composition Algebra: A Reduction-Based Schedulability Algebra for Distributed Real-Time Systems. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 259–269, December 2008. doi:10.1109/RTSS.2008.38.

11 Shengbing Jiang. A decoupled scheduling approach for distributed real-time embedded automotive systems. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 191–198, 2006.

12 Cristian Klein, Martina Maggio, Karl-Erik Årzén, and Francisco Hernández-Rodriguez. Brownout: Building more robust cloud applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 700–711. ACM, 2014.

13 Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: a theory of deterministic queuing systems for the internet*, volume 2050 of *Lecture Notes in Computer Science*. Springer, 2001.

14 José L. Lorente, Giuseppe Lipari, and Enrico Bini. A hierarchical scheduling model for component-based real-time systems. In *Proc. of the 20th International Parallel and Distributed Processing Symp.*, April 2006. doi:10.1109/IPDPS.2006.1639405.

15 Dana Marinca, Pascale Minet, and Laurent George. Analysis of deadline assignment methods in distributed real-time systems. *Computer Communications*, 27(15):1412–1423, 2004.

16 Victor Millnert, Johan Eker, and Enrico Bini. Dynamic control of NFV forwarding graphs with end-to-end deadline constraints. In *IEEE International Conference on Communications*, pages 1–7. IEEE, 2017.

17 Victor Millnert, Johan Eker, and Enrico Bini. Achieving predictable and low end-to-end latency for a network of smart services. In *IEEE GLOBECOM 2018*, 2018.

18 Tommi Nylander, Marcus Thelander Andrén, Karl-Erik Årzén, and Martina Maggio. Cloud Application Predictability through Integrated Load-Balancing and Service Time Control. In *2018 IEEE International Conference on Autonomic Computing (ICAC)*, pages 51–60. IEEE, 2018.

19 Pradeep Padala, Kang G Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive control of virtualized resources in utility computing environments. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 289–302. ACM, 2007.

20 José Carlos Palencia and Michael González Harbour. Schedulability Analysis for Tasks with Static and Dynamic Offsets. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 26–37, December 1998.

21 Rodolfo Pellizzoni and Giuseppe Lipari. Holistic analysis of asynchronous real-time transactions with earliest deadline scheduling. *Journal of Computer and System Sciences*, 73(2):186–206, March 2007. doi:10.1016/j.jcss.2006.04.002.

22 Nicola Serreli, Giuseppe Lipari, and Enrico Bini. The Demand Bound Function Interface of Distributed Sporadic Pipelines of Tasks Scheduled by EDF. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, pages 187–196, July 2010. doi:10.1109/ECRTS.2010.17.

23 Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, volume 4, pages 101–104. IEEE, 2000.

24 Ken Tindell and John Clark. Holistic Schedulability Analysis for Distributed Hard Real-Time Systems. *Microprocessing and Microprogramming*, 50:117–134, April 1994.

25 Thiemo Voigt and Per Gunningberg. Adaptive resource-based Web server admission control. In *ISCC*, 2002.

26 Qi Zhu, Haibo Zeng, Wei Zheng, Marco Di Natale, and Alberto Sangiovanni-Vincentelli. Optimization of Task Allocation and Priority Assignment in Hard Real-time Distributed Systems. *ACM Transactions on Embedded Computing Systems*, 11(4):85:1–85:30, January 2013. doi:10.1145/2362336.2362352.

Reliable Dynamic Packet Scheduling over Lossy Real-Time Wireless Networks

Tao Gong[1]
University of Connecticut, Storrs, USA

Tianyu Zhang
University of Notre Dame, USA
Qingdao University, China

Xiaobo Sharon Hu
University of Notre Dame, USA

Qingxu Deng
Northeastern University, Shenyang, China

Michael Lemmon
University of Notre Dame, USA

Song Han
University of Connecticut, Storrs, USA

Abstract

Along with the rapid development and deployment of real-time wireless network (RTWN) technologies in a wide range of applications, effective packet scheduling algorithms have been playing a critical role in RTWNs for achieving desired Quality of Service (QoS) for real-time sensing and control, especially in the presence of unexpected disturbances. Most existing solutions in the literature focus either on static or dynamic schedule construction to meet the desired QoS requirements, but have a common assumption that all wireless links are reliable. Although this assumption simplifies the algorithm design and analysis, it is not realistic in real-life settings. To address this drawback, this paper introduces a novel reliable dynamic packet scheduling framework, called RD-PaS. RD-PaS can not only construct static schedules to meet both the timing and reliability requirements of end-to-end packet transmissions in RTWNs for a given periodic network traffic pattern, but also construct new schedules rapidly to handle abruptly increased network traffic induced by unexpected disturbances while minimizing the impact on existing network flows. The functional correctness of the RD-PaS framework has been validated through its implementation and deployment on a real-life RTWN testbed. Extensive simulation-based experiments have also been performed to evaluate the effectiveness of RD-PaS, especially in large-scale network settings.

2012 ACM Subject Classification Networks → Network resources allocation; Networks → Network dynamics; Networks → Network reliability

Keywords and phrases Real-time wireless networks, lossy links, dynamic packet scheduling, reliability

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.11

1 Introduction

In recent years, real-time wireless networks (RTWNs) have been making their way into a wide range of industrial applications [1, 5, 14, 19]. These applications commonly have stringent timing and reliability requirements to ensure timely data collection and control decision delivery. Thus packet scheduling in RTWNs plays an important role for achieving the desired Quality of Service (QoS) in such applications. QoS here is often measured by

[1] The first two authors have equal contribution to this work.

how well the network delivers the packets by their deadlines. Although packet scheduling in RTWNs has been studied for a long time, how to handle abruptly increased network traffic in the presence of unexpected disturbances (i.e., events causing more frequent sensing of the environment and processing of sensed data) remains a challenge. This challenge is further exacerbated by the lossy wireless links in typical industrial environments [7].

Most RTWNs adopt Time Division Multiple Access (TDMA) based data link layers to achieve deterministic real-time communication. Sensing and control tasks are abstracted as end-to-end (e2e) flows with specified timing and reliability requirements. Most earlier packet scheduling algorithm designs in RTWNs focus on schedulability analysis and employ centralized and static (or infrequently updated) management frameworks (e.g., [17, 18, 16, 8, 22]). Those solutions may fit well for small-scale static RTWNs. They however often lead to significantly degraded QoS when the system becomes large and/or when deployed for monitoring and controlling complex physical processes where disturbances are present.

To model and respond to disturbances in RTWNs, many dynamic scheduling approaches have been proposed. Both [4] and [27] support admission control in response to adding/removing tasks for handling disturbances in the network. They however do not consider scenarios when not all tasks can meet their deadlines. The protocol in [12] proposes to allocate reserved slots for occasionally occurring emergencies (i.e., disturbances), and allow regular tasks to steal slots from the emergency schedule when no emergency exists. However, how to satisfy the deadlines of regular tasks in the presence of emergencies is not considered. [21] proposes a MAC protocol with a centralized reschedule scheme allowing on-line changes of active streams and network topology. However, the scheduler and the data format of the schedule distribution are not specified in [21].

Another thread of research significantly advances the state of the art by providing dynamic packet scheduling functions in RTWNs. Among these approaches, OLS in [9] relies on a centralized gateway to construct and disseminate a dynamic schedule to all the nodes in the network; D^2-PaS in [23] offloads the schedule construction to individual nodes and only disseminates minimum information for the nodes to construct a dynamic schedule locally; and FD-PaS in [26] further eliminates the need of a centralized gateway by notifies and handles the disturbances in a local and distributed manner. They, however, all assume perfect wireless network links, which is not realistic especially in noisy and harsh industrial environments. To our best knowledge, none of the existing dynamic packet scheduling algorithms consider packet losses and thus can lead to poor QoS for real-life deployment.

On the other hand, a rich set of methods have been designed for RTWNs to improve the reliability of wireless packet transmission over lossy links. For instance, most RTWN solutions (e.g., WirelessHART [20], ISA 100.11a [10], and 6TiSCH [6]) employ multiple channels and some frequency hopping mechanisms to minimize potential interference. Further, [8] proposed a set of reliable graph routing algorithms in WirelessHART networks to explore path diversity to improve reliability. These works are complementary to the approach to be introduced in this paper since we focus on single channel with pre-defined routing. [3] proposed an algorithm to allocate a necessary number of retransmision links for individual nodes to guarantee a desired success ratio of packet delivery in a star network topology. [2] extended the network model in [3] to allow multi-hop flows and proposed both Link-Centric and Flow-Centric scheduling policies. However, the policies in [3, 2] tend to assign more retransmission slots than necessary, and thus require higher network bandwidth. Our approach in this work results in an optimal retransmission slot assignment. Furthermore, all aforementioned studies only focus on packet scheduling in static RTWN settings over lossy links, and cannot be easily extended to handle abruptly increased network traffic caused by unexpected disturbances. In a recently submitted work [25], we addressed disturbance handling in lossy RTWNs. However, the schedule used in the static setting is generated by directly applying the retransmission mechanism in [2] which can lead to higher network bandwidth usage than necessary. Further, [25] is based on the distributed framework FD-PaS to handle disturbances which can cause high QoS degradation on other uncritical tasks according to the results in [26].

In this work, we introduce a reliable dynamic packet scheduling framework, called RD-PaS, for meeting both *timing* and *reliability* requirements in packet scheduling in the presence of disturbances. When no disturbance occurs (i.e., in the static scenario), RD-PaS determines the *minimum* number of retransmission slots needed for each task to guarantee reliable e2e packet delivery, and construct a communication schedule locally in a hybrid manner at individual nodes. The hybrid approach needs a centralized controller and a local schedule generator to keep a good tradeoff between bandwidth usage and QoS. When a disturbance occurs, RD-PaS generates a dynamic schedule to guarantee desired reliability of critical task(s) while judiciously degrade the reliability of packet transmissions for other tasks. We formulate a reliable dynamic scheduling problem to minimize such degradation, prove that this problem is NP-hard, and present an effective heuristic to solve it. The functional correctness of the RD-PaS framework has been validated through its implementation and deployment on a real-life RTWN testbed. Extensive simulation-based experiments have also been performed to evaluate the effectiveness of RD-PaS, especially in large-scale network settings. Our results show that RD-PaS can reduce e2e packet deliver ratio degradation in dynamic schedule by 58% on average compared to the D^2-PaS approach.

The remainder of this paper is organized as follows. Section 2 describes the system model and problem definition, and gives an overview of the RD-PaS framework. Section 3 presents the details of RD-PaS for the Transmission-based Scheduling (TBS) model, including both static schedule construction and dynamic schedule adjustment in the presence of disturbances. These efforts are further extended to the Packet-based Scheduling (PBS) model in Section 4. In Section 5, we present the implementation and functional validation of RD-PaS on a real-life RTWN testbed. Performance evaluation from extensive simulation-based experiments is reported in Section 6. Finally, we conclude the paper and discuss future work in Section 7.

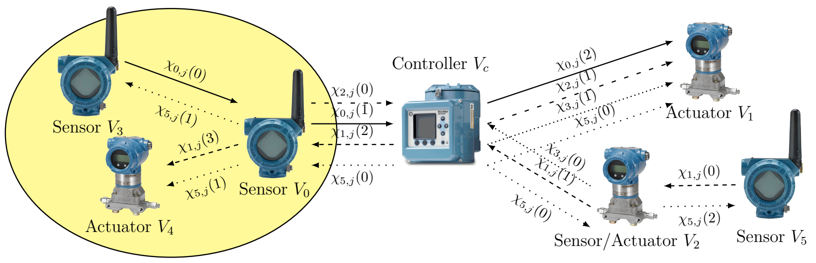
2 Preliminaries

In this section, we first discuss the system model and then give an overview of the proposed RD-PaS framework.

2.1 System Model and Problem Definition

The system architecture of an RTWN studied in this work is modeled after RTWNs often found in industrial process control applications. Such an RTWN consists of multiple sensor and actuator nodes wirelessly connected to a single controller node either directly or through relay nodes. The network is described by a directed graph $G = (V, E)$, where the node set $V = \{V_0, V_1, \ldots, V_c\}$. V_c is the controller node and the rest are referred to as the device nodes. A direct link $e = (V_i, V_j) \in E$ represents a wireless link from node V_i to V_j with a Packet Delivery Ratio (PDR), λ_e^L, which represents the probabilistic transmission success rate on link e[2]. V_c connects to all the nodes via some routes and is responsible for executing

[2] Link PDR λ_e^L is usually measured during the site survey and is stable during normal network operations. In case the value of λ_e^L changes significantly, the new value is assumed to be broadcast to all the nodes in the network.



Routing Paths

τ_0	τ_1	τ_2	τ_3	τ_5
$V_3 \downarrow V_0 \downarrow V_c \downarrow V_1$	$V_5 \downarrow V_2 \downarrow V_c \downarrow V_0 \downarrow V_4$	$V_0 \downarrow V_c \downarrow V_1$	$V_2 \downarrow V_c \downarrow V_1$	$V_c \downarrow V_0, V_1, V_2$ — $V_0 \downarrow V_3, V_4$ — $V_2 \downarrow V_5$

Figure 1 An example RTWN with 5 tasks running on 7 nodes. The sensor and actuator nodes are taken from a crude processing plant.

relevant control algorithms. V_c also contains a network manager which conducts network configuration and resource allocation. In this work, we focus on RTWNs with only one controller node. Networks with multiple controller nodes are left for future work.

We assume that the system executes a fixed set of control tasks $\mathcal{T} = \{\tau_0, \tau_1, \ldots \tau_n\}$ where τ_i $(0 \leq i < n)$ is a unicast task and τ_n is a broadcast task. Each task τ_i is associated with a period P_i and deadline D_i, and follows a designated single routing path with H_i hops. We use $\overrightarrow{L}_i = [L_i[0], L_i[1], \ldots, L_i[H_i - 1]]$ to represent the routing path of task τ_i. For a unicast task, $L_i[h] \in E$ $(0 \leq h < H_i)$. Each unicast task periodically generates a packet originated at a sensor node, passing through the controller node and delivering a control message to the designated actuator node. For the broadcast task τ_n, each hop involves multiple links, thus $L_n[h] = (L_n[h](0), L_n[h](1), \ldots)$, where $L_n[h](i) \in E$. The broadcast task runs periodically in V_c and only generates packets when necessary. These packets are broadcast to each node directly or though some intermediate nodes by the designed broadcast path L_n. The j-th released instance of τ_i is referred to as packet $\chi_{i,j}$, with its release time, deadline, and finish time denoted as $r_{i,j}$, $d_{i,j}$ and $f_{i,j}$, respectively. We denote the transmission of packet $\chi_{i,j}$ at the h-th hop as transmission $\chi_{i,j}(h)$, $(0 \leq h < H_i)$.

Fig. 1 shows an example RTWN running 4 unicast tasks (τ_0, τ_1, τ_2 and τ_3) and 1 broadcast task (τ_5) on 7 nodes ($V_0, V_1, \ldots, V_5$ and V_c) where V_0, V_3, V_5 are the sensor nodes, V_1, V_4 are the actuator nodes, and V_2 is a combined sensor and actuator node. The routing paths of individual tasks are summarized on the right side of Fig. 1.

In applications such as crude oil refining, a disturbance, e.g., a sudden change in temperature, may occur unexpectedly. When a disturbance occurs, the system usually requires the sensor nodes located within the range of the disturbance to monitor the environment more closely, and thus one or multiple tasks may demand more network bandwidth during the disturbance. To capture such abrupt increase in network resource demand upon the detection of a disturbance, we adopt the rhythmic task model [11] in this work[3]. In the rhythmic model, each task has two states: *nominal state* and *rhythmic state*. In the nominal state, τ_i releases packets following the nominal period P_i and each packet has a relative deadline $D_i \leq P_i$. In the rhythmic state, the period and relative deadline of τ_i adopt a series of new values specified by pre-designed vectors $\overrightarrow{P}_i$ and $\overrightarrow{D}_i$. Once τ_i returns to nominal state, it starts to use P_i and D_i again. When a disturbance occurs and the corresponding tasks (denoted as $\mathcal{T}_{Rhy}$) enter their rhythmic states, we say the system switches to the *rhythmic mode*. The system returns to the *nominal mode* after the disturbance has been completely handled, i.e.

[3] RD-PaS is not limited to the rhythmic task model and can be applied to any task models capturing unexpected network resource demand changes.

all the corresponding tasks return to their nominal states. In Fig. 1, when the disturbance (in the yellow region) occurs, τ_0 and τ_2 (installed on nodes V_3 and V_0, respectively) will enter their rhythmic states and the system switches to the rhythmic mode. In the following, we first assume that at any time during the system operation, at most one disturbance can occur and needs to be detected and handled. We will then generalize the system model to discuss concurrent disturbances at the end of Section 3.2.

Following the industrial practice for RTWNs, we consider a synchronized network adopting a time-slotted schedule. The length of a time slot is typically 10ms. Within each time slot, at most one packet can be transmitted over the air from a sender to a receiver. The acknowledgement (ACK) is then sent back from the receiver to the sender in the same slot to notify the successful reception.

Traditional RTWNs employ Link-based Scheduling (LBS) to allocate time slots. In LBS, each time slot is allocated to a link by specifying the sender and receiver. If packets from different tasks share a common link and are both buffered at the same sender, their transmission order is decided by a node-specified policy (e.g., FIFO). This approach introduces uncertainty in packet scheduling and may violate the e2e timing constraints on packet delivery. To tackle this problem, *Transmission-based Scheduling (TBS)* and *Packet-based Scheduling (PBS)* are proposed in [23] and [2], respectively, to construct deterministic schedules. Each of the two scheduling models has its own advantages and disadvantages and is preferred in different usage scenarios as discussed in [2]. Hence, we consider both models in our RD-PaS framework. Furthermore we focus on single-channel RTWNs in this work since it forms the basis for more advanced studies. Multichannel networks are left for future work.

In the TBS model, each time slot is allocated to the transmission of a specfic packet $\chi_{i,j}$ at a particular hop h or kept idle. Once the network schedule is constructed, packet transmission in each time slot is unique and fixed. In the PBS model, each time slot is allocated to a specific packet $\chi_{i,j}$ or kept idle. Within each time slot assigned to $\chi_{i,j}$, every node along $\chi_{i,j}$'s routing path decides the action to take (e.g., transmit, receive or idle), depending on whether the node has received $\chi_{i,j}$ or not. Table 1 gives a time slot allocation example for task τ_2 in Fig. 1. In TBS model, each time slot is allocated to a dedicated hop. In PBS model, slot 1 can be used to transmit both hops depending on whether the first transmission succeeds in slot 0.

Table 1 An example of time slot allocation in TBS model and PBS model.

	Slot 0	**Slot 1**	**Slot 2**
TBS model	$V_0 \to V_c$	$V_0 \to V_c$	$V_c \to V_1$
PBS model	$V_0 \to V_c$	$V_0 \to V_c$ $V_c \to V_1$	$V_c \to V_1$

Since each link e in the network may suffer packet losses, i.e., $\lambda_e^L < 1$, packet transmissions may fail, which can significantly affect the timely delivery of real-time packets. To handle such cases, a retransmission mechanism is commonly employed in RTWNs [20, 6]. Specifically, if a sender node does not receive the ACK from the receiver node of a packet, it automatically retransmits the packet in the next possible time slot.

To quantify the reliability requirement of the e2e packet delivery for each task, a *required* e2e PDR for τ_i, denoted as λ_i^R, is introduced. For example, a control application can tolerate 0.01% packet loss, so λ_i^R is 99.99%. Based on λ_i^R, the transmission of any packet of τ_i is reliable if and only if the achieved e2e PDR of τ_i is larger than or equal to λ_i^R, i.e., $\lambda_{i,j} \geq \lambda_i^R$. To simplify presentation, we assume that all tasks in the network share a common

Table 2 Table of Important Symbols and Notations.

$V_0, V_1, \ldots$	Device nodes: sensor, actuator or relay node
V_c	Controller node
$\mathcal{T}$, τ_i	Task set and task i
H_i, P_i, D_i	Number of hops, period and deadline of τ_i
$L_i[h]$	The h-th link on the routing path of τ_i $(0 \le h < H_i)$
$\chi_{i,j}$	The j-th released packet of τ_i
$r_{i,j}, d_{i,j}, f_{i,j}$	Absolute release time, deadline, finish time of $\chi_{i,j}$
$W_{i,j}$	Total number of slots assigned for $\chi_{i,j}$
λ^R	Required e2e packet delivery ratio (for all tasks)
$\lambda^L_{L_i[h]}$	Measured link packet delivery ratio of link $L_i[h]$
$\lambda_{i,j}$, $\overrightarrow{R}_{i,j}$	E2e PDR value and retry vector of $\chi_{i,j}$
$R_{i,j}[h]$	Number of trials for h-th hop assigned by $\overrightarrow{R}_{i,j}$
$\lambda^*_i(\cdot)$	Optimal PDR of τ_i as a function of number of assigned slots
$\overrightarrow{R}^*_i(\cdot)$	Optimal retry vector of τ_i as a function of numer of assigned slots
w^+_i	The smallest w achieving $\lambda^*_i(w) \ge \lambda^R$
$[t_{sp}, t_{ep})$	Time duration of system rhythmic mode (dynamic schedule)
$\delta_{i,j}$	PDR degradation of $\chi_{i,j}$
Γ	Active packet set containing all packets to be scheduled in the system rhythmic mode
ρ	Updated packet set

required e2e PDR value, denoted as λ^R. However, our proposed approach can be easily extended to support different λ^R's for different tasks. Table 2 summarizes the frequently used symbols in this paper.

Based on the above system model, the two key problems that we aim to solve in this work are as follows. **P1:** In the system nominal mode, construct a schedule such that both the e2e timing and reliability requirements of all tasks can be satisfied; **P2:** When disturbances occur and are detected, adjust the schedule in a dynamic and hybrid manner to still guarantee the reliable and timely transmissions of the rhythmic packets while achieving the minimum reliability degradation on other packets. More formally, we have the following.

P1: Given RTWN $G = (V, E)$ where each link $e \in E$ has an associated PDR, and task set $\mathcal{T}$ in which each task τ_i has a single routing path $\overrightarrow{L}_i$, determine the nominal-mode schedule under which the following constraints are satisfied.

▶ **Constraint 1.** $\forall i, j, \lambda_{i,j} \ge \lambda^R$. (e2e reliability requirements for all tasks)

▶ **Constraint 2.** $\forall i, j, f_{i,j} \le d_{i,j}$. (e2e timing requirements for all tasks)

P2*: Given the packet set, Γ, in the rhythmic mode under consideration, the PDR function of each task τ_i, and other network related constraints, determine the rhythmic-mode schedule such that $\sum_{\chi_{i,j} \in \Gamma} \max\{0, \lambda^R - \lambda_{i,j}\}$ is minimized, with the following constraints being satisfied.

▶ **Constraint 3.** $\forall \tau_i \in \mathcal{T}_{Rhy}, \lambda_{i,j} \ge \lambda^R$. (e2e reliability requirements for rhythmic tasks)

▶ **Constraint 4.** $\forall \tau_i \in \mathcal{T}_{Rhy}, f_{i,j} \le d_{i,j}$. (e2e timing requirements for rhythmic tasks)

Here we use **P2*** instead of **P2** as we have not discussed the network constraints. They will be elaborated in Section 3 and 4 where formal definitions of **P2** will be given.

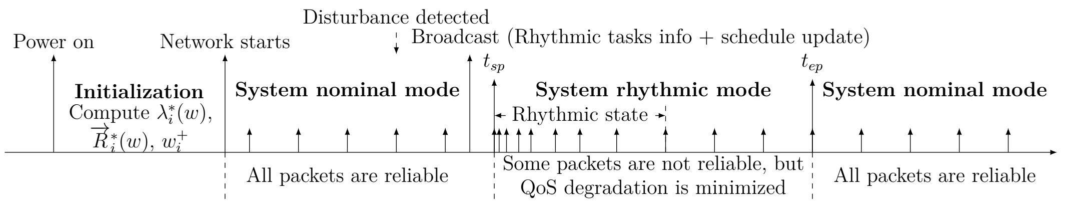


Figure 2 Overview of the execution model of RD-PaS in both nominal and rhythmic modes. Short upward arrows represent the releases of the rhythmic packets.

2.2 Overview of the RD-PaS Framework

We propose a reliable dynamic packet scheduling framework, referred to as RD-PaS, to address the questions raised above. An overview of the execution model of RD-PaS is shown in Fig. 2. Below we focus on a high-level discussion while leave the detailed explanation of the symbols in Section 3.

In the network initialization phase, each device node stores necessary specification information of all tasks (i.e., H_i, D_i, P_i and λ^R) locally after receiving it from the network manager through broadcast packets. Each device node then calculates the number of time slots to be allocated to each task (for both transmission and retransmission) in order to achieve the required e2e PDR value λ^R.

After the network starts, each device node generates a static schedule locally, following which all tasks can meet their timing and reliability requirements. By locally generating a static schedule, no unnecessary bandwidth is wasted on transmitting the schedule from the gateway. When a disturbance occurs, several sensor nodes within the range may detect it and send a report to the controller node via the corresponding tasks. After the controller node receives the disturbance information from any of the sensor nodes, V_c first determines a time duration, denoted as $[t_{sp}, t_{ep})$, during which the system runs in the rhythmic mode using a temporary dynamic schedule. As RD-PaS and D^2-PaS in [23] both require each node to generate schedule locally, RD-PaS adopts the same end point selection method in D^2-PaS to determine the system rhythmic mode duration $[t_{sp}, t_{ep})$. V_c, then, checks whether all tasks can still be reliably delivered after the rhythmic tasks entering their rhythmic states. If so, V_c only broadcasts the rhythmic tasks information (task IDs and the corresponding $\overrightarrow{P}_i$ and $\overrightarrow{D}_i$) to the network. Otherwise, V_c needs to generate a dynamic schedule in which the number of time slots assigned to certain periodic packets are updated in order to accommodate the increased workload from the rhythmic tasks. V_c then piggybacks the information of the updated packet set as well as the rhythmic tasks information to a broadcast packet and disseminates it to all nodes in the network. After all the nodes receive the updates, the system switches to the rhythmic mode to handle the disturbance.

In the rhythmic mode, individual device nodes generate their own dynamic schedules locally and these local schedules collaboratively guarantee the timing and reliability requirements of the rhythmic packets while minimizing the total reliability degradation suffered by other periodic tasks. After executing the dynamic schedules, all the device nodes return to the nominal mode and re-employ the static schedule.

In the following, we first present the details of the RD-PaS framework for the TBS model in Section 3. We then introduce required modifications to support the RD-PaS framework for the PBS model in Section 4.

3 Reliable Scheduling for TBS

This section focuses on reliable scheduling for the Transmission-based Scheduling (TBS) model. We first describe how RD-PaS constructs a reliable static schedule in the system nominal mode. We then introduce how RD-PaS handles disturbances in the rhythmic mode.

3.1 Reliable Static Scheduling

An RTWN starts at running in the nominal mode in which all tasks need to 1) be reliably scheduled to achieve the required e2e PDRs; and 2) meet the e2e timing constraints for all the packet transmissions. That is, we need to solve **P1** defined in Section 2.1. In the TBS model, each specific time slot is assigned to an individual packet transmission. Considering the lossy nature of wireless links, when a transmission is not successful, retransmissions are needed, which require extra time slots. To reduce the demand on network resources, we aim to minimize the number of extra slots for each task while satisfying the reliability requirement (i.e. Constraint 1 in **P1**). On the other hand, we observe that Constraint 2 can be handled separately from Constraint 1 since satisfying Constraint 2 can be treated as a standard transmission scheduling problem once the number of extra time slots is determined for each task. Thus, we intend to first tackle the following sub-problem.

P1.1: Given RTWN $G = (V, E)$ where each link $e \in E$ has an associated PDR, and task set $\mathcal{T}$ in which each task τ_i has a single routing path $\overrightarrow{L}_i$, determine the minimum number of extra slots needed by each task τ_i for satisfying Constraint 1.

To solve **P1.1**, we propose to first determine whether a given number of extra time slots for each task can satisfy Constraint 1 and then search for the optimal number of extra time slots for every task. We will prove later that this approach indeed leads to an exact solution for **P1.1**. We discuss our approach in detail below.

Let $\overrightarrow{R}_{i,j} = [R_{i,j}[0], R_{i,j}[1], \ldots, R_{i,j}[H_i - 1]]$ be the *retry vector* of packet $\chi_{i,j}$, where $R_{i,j}[h]$ denotes the number of time slots assigned to hop h of $\chi_{i,j}$. We use $W_{i,j}$ to denote the total number of time slots assigned to $\chi_{i,j}$, i.e., $W_{i,j} = \sum_{h=0}^{H_i-1} R_{i,j}[h]$. Given the PDRs of all the links along the routing path of τ_i and the retry vector of $\chi_{i,j}$, the e2e PDR of $\chi_{i,j}$, $\lambda_{i,j}$, can be derived as:

$$\lambda_{i,j} = \prod_{h=0}^{H_i-1} 1 - (1 - \lambda^{L}_{L_i[h]})^{R_{i,j}[h]}. \tag{1}$$

According to Constraints 1 and 2 in **P1**, all the packets released by τ_i must meet the same timing and reliability requirements in the system nominal mode. Thus, in the following discussion, we only consider parameter settings (including both the assigned number of slots and the retry vector) for each individual task τ_i instead of each packet $\chi_{i,j}$. For a given number of slots, say w, assigned to τ_i, the number of possible slot allocations, i.e. retry vectors, equals to $\binom{w-1}{H_i-1}$. We further introduce the following definitions.

▶ **Definition 1.** Optimal Retry Vector $\overrightarrow{R}^*_i(w)$: *An optimal retry vector of task τ_i for a given number of slots w is the retry vector that leads to the largest PDR value for the given w, denoted as $\lambda^*_i(w)$, among all the possible allocations.*

▶ **Definition 2.** Optimal Retry Vector Function $\overrightarrow{R}^*_i(\cdot)$: *The optimal retry vector function of task τ_i is the set of pairs $(w, \overrightarrow{R}^*_i(w))$ such that each $\overrightarrow{R}^*_i(w)$ is the optimal retry vector for the given number of slots w.*

Algorithm 1 PDR Table Computation under TBS for Task τ_i.

```
Input: $G = (V, E)$, $\tau_i$, $\lambda^R$
Output: PDR table of $\tau_i$ and $w_i^+$
1: $w \leftarrow H_i$;
2: $\overrightarrow{R}_i^*(w) \leftarrow [1, 1, 1, \ldots]$;
3: $\lambda_i^*(w) \leftarrow \prod_{h=0}^{H_i-1} \lambda_{L_i[h]}^L$;
4: while $\lambda_i^*(w) < \lambda^R$ do
5:     $w \leftarrow w + 1$;
6:     Select the hop index $h$ which yields the maximum PDR value (computed by Eq. (1));
7:     Update $\overrightarrow{R}_i^*(w)$ and $\lambda_i^*(w)$ in PDR table;
8: end while
9: $w_i^+ \leftarrow w$
```

▶ **Definition 3.** Optimal PDR Function $\lambda_i^*(\cdot)$: *The optimal PDR function of task τ_i is the set of pairs $(w, \lambda_i^*(w))$ such that each PDR value $\lambda_i^*(w)$ corresponds to the optimal retry vector with the given number of slots w.*

As the first step towards satisfying Constraint 1, we present our solution to calculate the optimal retry vector function $\overrightarrow{R}_i^*(\cdot)$ and the optimal PDR function $\lambda_i^*(\cdot)$ for each task τ_i. As both functions are only related to task τ_i itself, the computation for each task is independent. For the sake of clarity, we create a PDR table for each task τ_i to store both $\overrightarrow{R}_i^*(\cdot)$ and $\lambda_i^*(\cdot)$ for all (needed) values of w in each node, such overhead in our implementation is given in Sec. 5. (An example PDR table can be found in Table 4 in Section 5.) Below, we describe our optimal PDR table generation algorithm, Alg. 1, and prove its optimality.

Alg. 1 iteratively constructs the PDR table. At each iteration, we add one time slot to τ_i at the h-th hop that yields the maximum PDR value λ_i^* and store the resulting retry vector $\overrightarrow{R}_i^*$ into the PDR table (Lines 5-7). The retry vector is initially set to $[1, 1, 1, \ldots]$ and the corresponding PDR value equals to $\prod_{h=0}^{H_i-1} \lambda_{L_i[h]}^L$ (Lines 1-3). Since the required PDR value is λ^R, the iterative process stops when $\lambda_i^*(w) \geq \lambda^R$. We use w_i^+ to denote the minimum number of slots that guarantees the reliable delivery of τ_i.

Lemma 4 and Theorem 5 below affirm that Alg. 1 indeed results in the optimal retry vector function $\overrightarrow{R}_i^*(\cdot)$ and optimal PDR function $\lambda_i^*(\cdot)$.

▶ **Lemma 4.** *Let $\mathcal{G}(R^*(w)[h], \lambda_{L[h]}^L) = \frac{\lambda^*(w+1)}{\lambda^*(w)}$ be a function of $R^*(w)[h]$ and $\lambda_{L[h]}^L$. When $\lambda_{L[h]}^L$ is set to an arbitrary value λ_0, $\mathcal{G}_{\lambda_0} = \mathcal{G}(R^*(w)[h], \lambda_0)$ is a monotonically decreasing function of $R^*(w)[h]$.*

Proof of Lemma 4. If we update $\overrightarrow{R}^*(w)$ by allocating one slot at an arbitrary hop h-th, according to Eq. (1), we only need to update $\lambda^*(w)$ by replacing the term $1-(1-\lambda_{L[h]}^L)^{R^*(w)[h]}$ by $1-(1-\lambda_{L[h]}^L)^{R^*(w)[h]+1}$ to get $\lambda^*(w+1)$. That is,

$$\mathcal{G}(R^*(w)[h], \lambda_{L[h]}^L) = \frac{\lambda^*(w+1)}{\lambda^*(w)} = \frac{1-(1-\lambda_{L[h]}^L)^{R^*(w)[h]+1}}{1-(1-\lambda_{L[h]}^L)^{R^*(w)[h]}}$$

Thus, if $\lambda_{L[h]}^L$ is fixed to λ_0, we have:

$$\mathcal{G}'_{\lambda_0} = \frac{\partial \mathcal{G}(R^*(w)[h], \lambda_0)}{\partial R^*(w)[h]} = \frac{\lambda_0 \cdot (1-\lambda_0)^{R^*(w)[h]} \log(1-\lambda_0)}{\left((1-\lambda_0)^{R^*(w)[h]} - 1\right)^2}$$

Since $0 < \lambda_{L[h]}^L < 1$ and $(1-\lambda_{L[h]}^L)^{R^*(w)[h]} > 0$, we have $\mathcal{G}'_{\lambda_0} < 0$. Further, $\mathcal{G}_{\lambda_0}$ decreases monotonically as $R^*(w)[h]$ increases. ◀

▶ **Theorem 5.** *For any given number of time slots w, no other retry vector can yield a larger PDR value than $\overrightarrow{R}_i^*(w)$ as computed by Alg. 1.*

Proof of Theorem 5. We prove the theorem by mathematical induction, i.e., for any $w = H, H+1, \ldots, w^+$, the retry vector $\overrightarrow{R}^*(w)$ determined by Alg. 1 can achieve the largest PDR value $\lambda^*(w)$. (Here we omit the task index i since only one task is considered).

Base case: When $w = H$, the statement holds as only one possible retry vector exists, i.e., $\overrightarrow{R}^*(H) = [1, 1, \ldots, 1]$.

Inductive step: Suppose the PDR value of $\overrightarrow{R}^*(w)$ is largest among that of all possible retry vectors when $w = k$, we should prove that the PDR value of $\overrightarrow{R}^*(k+1)$ obtained by Alg. 1, i.e. $\lambda^*(k+1)$ is also the largest. We prove this by contradiction.

Suppose there exists another retry vector (denoted as $\overrightarrow{R}^o(k+1)$) leads a larger PDR value, i.e., $\lambda^*(k+1) < \lambda^o(k+1)$. Since the total number of slots assigned to the task (i.e., the sum of all elements in the retry vectors) both equal to $k+1$ and $\overrightarrow{R}^*(k+1) \neq \overrightarrow{R}^o(k+1)$, we can always find one hop at which the number of assigned slots in $\overrightarrow{R}^o(k+1)$ is larger than that in $\overrightarrow{R}^*(k+1)$. We use q to denote this hop index and $R^o(k)[q]$ to denote the number of slots assigned at the q-th hop in $\overrightarrow{R}^o(k)$. Then, $R^o(k+1)[q] > R^*(k+1)[q]$. Suppose $\overrightarrow{R}^*(k+1)$ is achieved by adding one slot at the p-th hop in $\overrightarrow{R}^*(k)$.

Case 1: $p = q$. In this case, $\overrightarrow{R}^*(k+1)$ and $\overrightarrow{R}^o(k+1)$ are both achieved by adding one slot at the p-th hop in $\overrightarrow{R}^*(k)$ and $\overrightarrow{R}^o(k)$, respectively. Then, according to Lemma 4, $\lambda^*(k+1)$ and $\lambda^o(k+1)$ can be rewritten with $\mathcal{G}(R^*(w)[h], \lambda^L_{L[h]})$ function as follows:

$$\lambda^*(k+1) = \lambda^*(k) \cdot \mathcal{G}(R^*(k)[p], \lambda^L_{L[p]}), \quad \lambda^o(k+1) = \lambda^o(k) \cdot \mathcal{G}(R^o(k)[p], \lambda^L_{L[p]}).$$

According to the assumption that the PDR value of $\overrightarrow{R}^*(k)$ is largest, we have $\lambda^*(k) \geq \lambda^o(k)$. Since $R^*(k)[p] < R^o(k)[p]$, according to Lemma 4, we have $\mathcal{G}(R^*(k)[p], \lambda^L_{L[p]}) > \mathcal{G}(R^o(k)[p], \lambda^L_{L[p]})$. Then, $\lambda^*(k+1) > \lambda^o(k+1)$. This contradicts our assumption.

Case 2: $p \neq q$. $\lambda^*(k+1)$ and $\lambda^o(k+1)$ can be rewritten as:

$$\lambda^*(k+1) = \lambda^*(k) \cdot \mathcal{G}(R^*(k)[p], \lambda^L_{L[p]}), \quad \lambda^o(k+1) = \lambda^o(k) \cdot \mathcal{G}(R^o(k)[q], \lambda^L_{L[q]}).$$

As $\lambda^*(k+1) < \lambda^o(k+1)$ and $\lambda^*(k) \geq \lambda^o(k)$, it must holds that

$$\mathcal{G}(R^*(k)[p], \lambda^L_{L[p]}) < \mathcal{G}(R^o(k)[q], \lambda^L_{L[q]}). \tag{2}$$

Since $R^*(k)[q] < R^o(k)[q]$ according to the assumption, the following inequality holds:

$$\mathcal{G}(R^*(k)[q], \lambda^L_{L[q]}) > \mathcal{G}(R^o(k)[q], \lambda^L_{L[q]}). \tag{3}$$

Combining Eq. (2) and Eq. (3), we have $\mathcal{G}(R^*(k)[p], \lambda^L_{L[p]}) < \mathcal{G}(R^*(k)[q], \lambda^L_{L[q]})$. Further,

$$\lambda^*(k) \cdot \mathcal{G}(R^*(k)[p], \lambda^L_{L[p]}) < \lambda^*(k) \cdot \mathcal{G}(R^*(k)[q], \lambda^L_{L[q]}).$$

This means that if we allocate one slot at the q-th hop in $\overrightarrow{R}^*(k)$ instead of at the p-th hop, we can have a larger PDR value. This contradicts with Alg. 1 which allocats one slot at the hop which yields the largest PDR value at each iteration.

Since both cases lead to contradiction, the inductive step is proved. Thus, Theorem 5 holds for all values of w. ◀

Now with the functions $\overrightarrow{R}_i^*(\cdot)$ and $\lambda_i^*(\cdot)$ being determined, we have successfully solved **P1.1**. To satisfy Constraint 2 in **P1**, we need to create a static schedule, i.e., specifying when a packet uses a slot, to ensure real-time constraints are met. We introduce an observation that helps map the reliable static schedule generation problem, i.e., **P1**, to a conventional real-time scheduling problem.

▶ **Observation 1.** *Given task set $\mathcal{T}$ to be reliably scheduled, if we set the number of slots for τ_i to w_i^+ according to $\lambda_i^*(\cdot)$*[4]*, w_i^+ is then equivalent to the execution time of τ_i. Then, each task $\tau_i \in \mathcal{T}$ with P_i, D_i and w_i^+ can be mapped to a task in a conventional real-time task set with the same period, deadline and execution time. Thus, a feasible schedule for the corresponding conventional real-time task set is also a feasible schedule under which all tasks in $\mathcal{T}$ can be reliably delivered.*

Given the schedule specifying the slot assignment for each task, each node can further allocate specific slots to the transmission at each hop according to the retry vector function $\overrightarrow{R}_i^*(\cdot)$. Thus, given a task set to be reliably scheduled in an RTWN, the network can adopt any conventional real-time scheduling algorithm to generate a static schedule that guarantees to meet all the constraints in **P1**. Since we allow at most one transmission within each timeslot, determining the nominal-mode schedule (i.e., **P1**) can be mapped to a uni-processor scheduling problem. Here, we adopt Earliest-Deadline-First (EDF) [13] to generate optimal schedule for tasks and assign time slots to transmissions according to retry vector, consistently at each node.

Note that regarding the broadcast task, two more issues need to be considered. First, the transmission of a broadcast packet at each hop involves one sender node but multiple receiver nodes. Second, no acknowledgement is sent back from the receiver nodes in a broadcast slot. The first issue mainly affects the number of slots assigned at each hop since multiple links with different link PDRs are involved. To tackle this, we directly adopt the lowest link PDR to determine the number of retries assigned at the hop. Due to the second issue, the sender node does not have any knowledge about whether the current transmission succeeds. Thus, we just let the sender node to keep transmitting at all the slots assigned to the current hop to maximize the success probability.

3.2 Reliable Dynamic Scheduling

Our proposed solution for **P1** ensures that both timing and reliability requirements are met in the system nominal mode. However, upon the detection of any disturbance, the corresponding tasks enter their rhythmic states and follow new release patterns and deadlines as shown in Fig. 2. The static schedule may no longer be able to meet both requirements especially for all the critical rhythmic packets. Therefore, a well-designed reliable dynamic packet scheduling mechanism is needed to enable the system to be adaptive to any workload change after the detection of a disturbance.

In our RD-PaS framework, the network generates the static schedule by assigning w_i^+ slots to each task τ_i according to the retry vector function. When a disturbance is detected and reported to the control node, the system follows the execution model outlined in Section 2.2

[4] All the retry vectors for other w values stored in $\overrightarrow{R}_i^*(\cdot)$ are used in the dynamic schedule generation, which will be discussed in Section 3.2.

to switch to the rhythmic mode. The main challenge here is to generate a temporary dynamic schedule when tasks cannot be reliably delivered after the rhythmic tasks (in $\mathcal{T}_{Rhy}$) enter their rhythmic states. That is, problem $\mathbf{P2}^*$ needs to be solved. The dynamic schedule must be able to accommodate the increased rhythmic workload and minimizes the degradation on both timing and reliability of other periodic tasks. Specifically, all the rhythmic packets must meet their timing and reliability requirements. That is, Constraints 3 and 4 are satisfied.

To ensure this, we may have to sacrifice the reliability requirements, i.e. lowering the e2e PDR values of some periodic packets, or even sacrifice their timing requirements, i.e. dropping some periodic packets. That is, the number of slots assigned to each packet may need to be updated. Since the PDR table for each task containing both the retry vector function $\overrightarrow{R}_i^*(\cdot)$ and PDR function $\lambda_i^*(\cdot)$ is pre-calculated and stored at each node, V_c only needs to piggyback on a broadcast packet the information of the updated total number of slots ($W_{i,j}$) assigned to each periodic packet, and then each node can decode the updated retry vector accordingly, once it receives this information.[5]

To formally define the dynamic schedule generation problem, we introduce some concepts/notation. Let Γ denote the *active packet set* containing all the packets to be scheduled within the rhythmic mode duration $[t_{sp}, t_{ep})$. Since the payload size of a broadcast packet is bounded, we set an upper bound on the number of periodic packets whose $W_{i,j}$ can be changed, and denote it as α. To capture the reliability degradation for periodic packet $\chi_{i,j}$, let $\delta_{i,j}$ represent the difference between the required PDR λ^R and the updated PDR value $\lambda_{i,j} = \lambda_i^*(W_{i,j})$ in the dynamic schedule, i.e., $\delta_{i,j} = \max\{0, \lambda^R - \lambda_{i,j}\}$. Note that the timing degradation of each packet can also be captured by $\delta_{i,j}$ where $\delta_{i,j} = \lambda^R$ if $\chi_{i,j}$ is dropped. Now the dynamic schedule generation problem, which is defined formally below, becomes finding $W_{i,j}$ for each periodic packet in Γ to satisfy Constraint 3 and 4.

P2: Given the active packet set Γ, the PDR function $\lambda_i^*(\cdot)$ of each task τ_i, the maximum allowed number of updated packets α, determine the updated packet set $\rho = \{W_{i,j} | \chi_{i,j} \in \Gamma\}$ such that i) the size of ρ is not larger than α, i.e., $|\rho| \leq \alpha$, and ii) the total reliability degradation is minimized, i.e., $\forall \chi_{i,j} \in \rho, \min \sum \delta_{i,j}$.

The theorem below states that determining the updated packet set, i.e. solving $\mathbf{P2}$, is non-trivial.

▶ **Theorem 6.** *The updated packet set generation problem* ***P2****, i.e., the dynamic schedule generation problem, is NP-hard.*

Proof of Theorem 6. We prove the theorem by reducing the 0-1 knapsack problem [15] to a special case of the updated packet set generation problem.

The 0-1 knapsack problem is defined as follows: Given a set of n items numbered from 1 up to n, each with a weight w_i and a value v_i, along with a maximum weight capacity W. Each item can either be included in the knapsack, denoted as $x_i = 1$, or not which is denoted by $x_i = 0$. The 0-1 knapsack problem is to maximize the sum of the values of the items in the knapsack, i.e. $\max \sum_{i=1}^n v_i x_i$, so that the sum of the weights is less than or equal to the knapsack's capacity W, i.e. $\sum_{i=1}^n w_i x_i \leq W$ and $x_i \in \{0, 1\}$.

Given a knapsack problem, we construct a special case of the updated packet set generation problem in polynomial time: Suppose the active packet set $\Gamma = \{\chi_1, \chi_2, ..., \chi_n\}$ such that $\forall \chi_i \in \Gamma, r_i = 0, D_i = W, H_i = w_i$. Each packet χ_i can either be scheduled, i.e. $\lambda_i = v_i$ or

[5] In the system rhythmic mode, we adjust the assigned number of slots for each packet instead of each task for more flexibility.

Algorithm 2 Updated Packet Set Generation.

```
Input: $\Gamma, \alpha, \lambda_i^*(w)$
Output: $\rho$
 1: Schedule the rhythmic packets in $\Gamma$ using $w_0^+$;
    //Suppose $n$ is the number of periodic packets in $\Gamma$
 2: if all periodic packets in $\Gamma$ can be reliably scheduled then
 3:    No packet needs to be updated;
 4: else
 5:    Find the first $n - \alpha$ schedulable periodic packets with the minimum $w_i^+$ using the packet-
       dropping heuristic in [23];
 6:    if Such $n - \alpha$ periodic packets can be found then
 7:       if the $\alpha$ packets can be scheduled using $H_i$ then
 8:          Assign extra slots to the $\alpha$ packets by Alg. 3;
 9:       else
10:          Determine the dropped packet set (suppose $m$ packets) using the dropping heuristic in
             [23];
11:          Assign extra slots to the $\alpha - m$ packets by Alg. 3;
12:       end if
13:    else
14:       Drop all the periodic packets;
15:    end if
16: end if
```

dropped, i.e. $\lambda_i = 0$. Let the required PDR value λ^R for all packets equals to $\max\{v_i\}$. Then, the PDR degradation $\delta_i = \lambda^R - v_i$ if χ_i is scheduled. Otherwise, $\delta_i = \lambda^R$.

As minimizing the total PDR degradation for all packets equals to maximizing the total PDR value, the updated packet set with the minimum total PDR degradation can be determined if and only if a knapsack with the maximum value can be identified. ◀

Next we propose a heuristic to solve **P2** and the high-level idea is as follows. Since dropping any packet $\chi_{i,j}$ leads to a significant decrease in the PDR value of $\chi_{i,j}$, i.e., $\delta_{i,j} = \lambda^R$, we always prefer to allocate at least the basic number of slots (i.e., H_i) to each packet. If the network bandwidth is sufficient, we assign extra slots to periodic packets in a greedy manner according to their PDR degradation. Alg. 2 summarizes the updated packet set generation algorithm which uses the greedy extra slots assignment heuristic described in Alg. 3. Specifically, at each iteration, Alg. 3 adds one slot to the packet resulting in the minimum PDR degradation after an extra slot has been assigned. Using Alg. 2 and Alg. 3, the updated packet set can be determined in $\mathcal{O}(\alpha \cdot W_{max})$ time where W_{max} is the maximum w_i^+ among all the tasks.

Note that the proposed RD-PaS framework can be readily extended to handle concurrent disturbances in RTWNs, following the similar way as elaborated in [24]. Specifically, we need to handle two cases depending on the relative positions of any two consecutive disturbances [24]. The first case is when both disturbances occur before an upcoming broadcast slot. Then, V_c simply generates a dynamic schedule considering all rhythmic tasks triggered by the two disturbances to handle them together. The second case is when a subsequent disturbance arrives at V_c after the dynamic schedule information for handling the first disturbance has been broadcast. In this case, V_c must update the dynamic schedule starting from the next broadcast slot. The readers are referred to [24] for the details.

Algorithm 3 Extra Slots Assignment.

```
1:  $\mathcal{S}_{extra} \leftarrow$ {Packets to be assigned extra slots};
2:  while $\mathcal{S}_{extra} \neq \emptyset$ do
3:      Add one slot to the packet $\chi_s$ if doing so leads to the minimum PDR degradation;
4:      if the system is schedulable then
5:          if $\chi_s$ is already reliable then
6:              Remove $\chi_s$ from $\mathcal{S}_{extra}$;
7:          end if
8:      else
9:          Reduce one slot from $\chi_s$;
10:         Remove $\chi_s$ from $\mathcal{S}_{extra}$;
11:     end if
12: end while
```

4 Reliable Scheduling for PBS

In this section, we discuss how to support the RD-PaS framework for the packet-based scheduling (PBS) model. At the highest level, reliable scheduling for PBS has three main differences from that for TBS. First, since each time slot is assigned to a specific packet instead of a dedicated hop, retry vector $\overrightarrow{R}_{i,j}$ and its function $\overrightarrow{R}_i^*(\cdot)$ are no longer needed. Second, the computation for PDR function $\lambda_i^*(\cdot)$ is different because the time slot allocation mechanism has changed. Third, the retransmission mechanism of the broadcast task for TBS, i.e., keep transmitting using all assigned slots at each hop, does not work for PBS since each slot allocation is not dedicated to a hop but a packet.

Since PDR function is a key parameter in checking reliability, we first describe how to compute the PDR value for a task with a given number of slots in PBS. Let $Pr_i(0,w)$ denote the probability of a packet of τ_i staying in the source node within w slots; $Pr_i(h,w)$ denote the probability of a packet of τ_i being transmitted to the receiver of the h-th hop along the routing path ($1 \leq h \leq H_i$), and have not been successfully forwarded, within w slots. $Pr_i(h,w)$ can be computed by:

$$Pr_i(h,w) = \begin{cases} 1 & h=0, w=0 \\ \lambda^L_{L_i[h-1]} Pr_i(h-1,w-1) & h \neq 0, w=h \\ (1-\lambda^L_{L_i[h]}) Pr_i(h,w-1) & h=0, w\neq 0 \\ Pr_i(h,w-1) + \lambda^L_{L_i[h-1]} Pr_i(h-1,w-1) & h=H_i, w\neq h \\ (1-\lambda^L_{L_i[h]}) Pr_i(h,w-1) + \lambda^L_{L_i[h-1]} Pr_i(h-1,w-1) & \text{otherwise.} \end{cases} \quad (4)$$

In Fig. 3, we use an example task with 2 hops (links a and b with PDR λ_a^L and λ_b^L, respectively) and 4 slots to describe the computation of $Pr_i(h,w)$. As shown in the figure, $Pr_i(h,w)$ can be either reached by $Pr_i(h-1,w-1)$, followed by a successful transmission ($\lambda^L_{L_i[h-1]}$), or $Pr_i(h,w-1)$, followed by a failed transmission $(1-\lambda^L_{L_i[h]})$, except for boundary conditions. These boundary conditions include the following:

Case 1: When $h=0, w=0$, the source node generates a packet ($Pr_i(0,0)=1$).

Case 2: When $h \neq 0, w=h$, it is not possible for $Pr_i(h,w)$ to be reached by $Pr_i(h,w-1)$ ($Pr_i(1,1)$, $Pr_i(2,2)$ in the figure). Thus only $Pr_i(h-1,w-1)$ is considered.

Case 3: When $h=0, w\neq 0$, it is not possible for $Pr_i(h,w)$ to be reached by $Pr_i(h-1,w-1)$ ($Pr_i(0,1)$, $Pr_i(0,2)$ in the figure). Thus only $Pr_i(h,w-1)$ is considered.

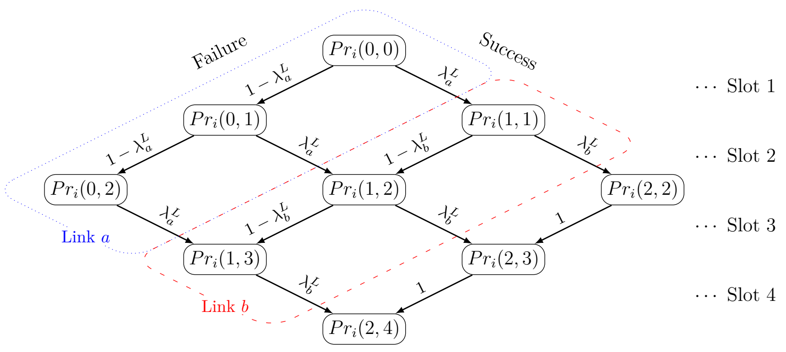


Figure 3 PDR computation for a task with two hops under the PBS model.

Algorithm 4 PDR Table Computation under PBS for Task τ_i.

Input: $G = (V, E)$, τ_i, λ^R
Output: The PDR function of τ_i and w_i^+

```
1: $w \leftarrow 0$;
2: while $\lambda_i(w) < \lambda^R$ or $w < H_i$ do
3:     $w \leftarrow w + 1$;
4:     for $h = 0$ to $H_i$ do
5:         Compute $Pr_i(h, w)$ following Eq.(4);
6:     end for
7:     if $w >= H_i$ then
8:         $\lambda_i^*(w) \leftarrow Pr_i(H_i, w)$;
9:     end if
10: end while
11: $w_i^+ \leftarrow w$
```

Case 4: When $h = H_i, w \neq h$, $Pr_i(h, w-1)$ always reaches $Pr_i(h, w)$ ($Pr_i(2,3)$, $Pr_i(2,4)$ in the figure).

Different from TBS, which finds the optimal PDR values by using retry vectors for a given w, the PDR values in PBS is solely determined by w, i.e., $\lambda_i^*(w) = Pr_i(H_i, w)$. Based on Eq.(4), we propose a dynamic programming algorithm (Alg. 4) to compute $Pr_i(h, w)$ and finally $\lambda_i^*(w)$. In Alg. 4, the iteration starts from $w = 1$, and stops when λ^R is reached. In each iteration, it computes all $Pr_i(h, w)$ for $0 \leq h \leq H_i$, and stores them to $\lambda_i^*(\cdot)$ if $w \geq H_i$.

After the PDR function is computed, we can apply the same method proposed in Section 2.2 and 3 to generate reliable static and dynamic schedule, respectively. More specifically, we use Observation 1 with computed PDR function to generate a reliable static schedule, and use Alg. 2 and Alg. 3 to determine the updated W in the rhythmic mode.

Now let us consider the broadcast task. Because the link layer multicast does not have ACK and in PBS each slot is allocated to a packet instead of a hop, it is not possible for the broadcast task to track its progress. Thus the broadcast task still needs to follow the TBS model. That is, for the broadcast task, we adopt the lowest link PDR for each hop among all the receivers, and use Alg. 1 to compute $\overrightarrow{R}_i^*(\cdot)$ and $\lambda_i^*(\cdot)$.

5 Testbed Implementation and Validation

To validate the functionality of the proposed RD-PaS framework in real-life RTWNs, we implemented RD-PaS on a 7-node RTWN testbed (see Fig. 4) running the 6TiSCH protocol.

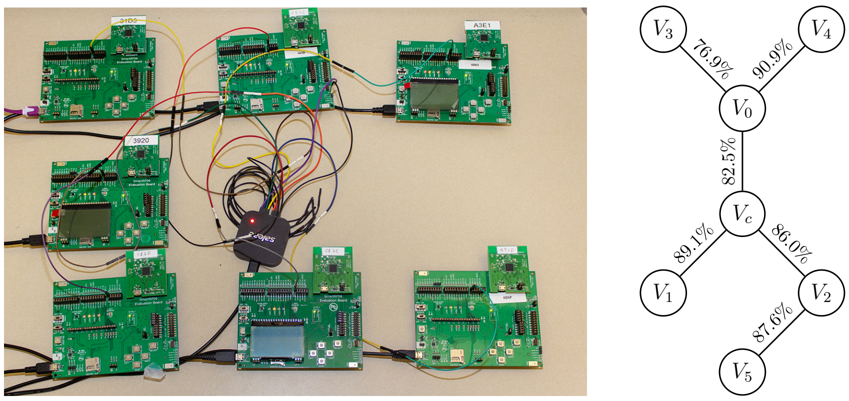


Figure 4 Left: the RTWN testbed with 7 CC2538 evaluation boards; Right: the testing topology with emulated link PDR values.

The testbed consists of seven CC2538 evaluation boards. One of these boards is configured as the controller node, while the others are configured as device nodes. A 16-channel 802.15.4 sniffer and an 8-channel logic analyzer are used to capture and analyze the activities of each device node. Our modified 6TiSCH stack utilizes 5KB more ROM and 2KB more RAM space for implementing RD-PaS (in TBS and PBS). These are relatively small compared to the original 6TiSCH stack which needs 69KB ROM and 6KB RAM. Due to the page limit, the implementation details of the RD-PaS framework is omitted. Below, we focus on discussing the functional validation of RD-PaS on the testbed.

The testing topology is shown on the right side of Fig. 4. To attain the link PDRs as specified in the topology, we implemented a random packet dropper at the MAC layer of each device node. Six tasks are installed in the testbed and the task specifications are summarized in Table 3. The desired e2e PDR for all the tasks, λ^R, is set to 99%. τ_0, τ_1, τ_2 and τ_3 are unicast tasks, τ_5 is a broadcast task, and τ_4 is a task that handles all network management packets. Since we always allocate two *shared* slots at the beginning of τ_4's period, we set $D_4 = 2$. For simplicity, only τ_0 enters the rhythmic state when a rhythmic event occurs.

5.1 Validation of reliable static scheduling

To validate the static schedule construction in RD-PaS, we run the specified task set on the testing topology in the nominal mode under both TBS and PBS models. The PDR tables computed by the testbed are exactly the same as those obtained from simulation. The PDR table for task τ_1 is given in Table 4 (while others are not shown due to the page limit). The highlighted rows indicate the corresponding w_i^+'s for TBS ($w_i^+ = 13$) and PBS ($w_i^+ = 7$) when λ^R is reached.

Table 3 Parameters of the task set deployed on the testbed.

Task	Routing Path	$P_i(D_i)$	$\overrightarrow{P}_i = \overrightarrow{D}_i$
τ_0	$V_3 \to V_0 \to V_c \to V_1$	30 (30)	$[20, 20, 20, 20, 20, 20]$
τ_1	$V_5 \to V_2 \to V_c \to V_0 \to V_4$	45 (45)	-
τ_2	$V_0 \to V_c \to V_1$	40 (40)	-
τ_3	$V_2 \to V_c \to V_1$	60 (60)	-
τ_4	-	60 (2)	-
τ_5	$V_c \to (V_0, V_1, V_2), V_0 \to (V_3, V_4), V_2 \to (V_5)$	120 (120)	-

Table 4 PDR table for task τ_1 in TBS and PBS models.

w	PDR Table in TBS Model		PDR Table in PBS Model
	$\lambda_i^*(w)$	$\overrightarrow{R}_i^*(w)$	$\lambda_i^*(w)$
4	0.564963	1,1,1,1	0.564963
5	0.663832	1,1,2,1	0.864394
6	0.756769	1,2,2,1	0.964613
7	0.850608	2,2,2,1	0.991720 ($\lambda_i^*(w_i^+)$)
8	0.928013	2,2,2,2	
9	0.952201	2,2,3,2	
10	0.968572	2,3,3,2	
11	0.981822	3,3,3,2	
12	0.989274	3,3,3,3	
13	0.993672 ($\lambda_i^*(w_i^+)$)	3,3,4,3	

Table 5 Reliable static schedule validation in TBS and PBS models on the testbed.

Task	TBS Model			PBS Model		
	$\overrightarrow{R}_i^*$	$\lambda_i^*(w_i^+)$	Measured PDR	w_i^+ or $\overrightarrow{R}_i^*$	$\lambda_i^*(w_i^+)$	Measured PDR
τ_0	[4,3,3]	99.01%	99.21%	7	99.68%	99.22%
τ_1	[3,3,4,3]	99.37%	99.61%	7	99.17%	99.65%
τ_2	[3,3]	99.34%	99.41%	5	99.80%	99.34%
τ_3	[3,3]	99.60%	99.71%	4	99.29%	99.65%
τ_5	[4,4,3]	99.38%	100%	[4,4,3]	99.38%	100%

We further test 5000 packets for each unicast and broadcast task under both models, and compare the actual e2e PDR values collected from the testbed with the simulated values from Alg. 1 and Alg. 4. These results are summarized in Table 5. τ_4 is omitted in the table since it is a task dedicated for network management packets. It can be concluded from the table that the reliable static scheduling function in RD-PaS executes correctly as the actual e2e PDRs are improved to the desired values ($\geq 99\%$) in both models in the presence of specified packet loss. The slight differences between the measured and predicted e2e PDR values are expected due to the limited sample size.

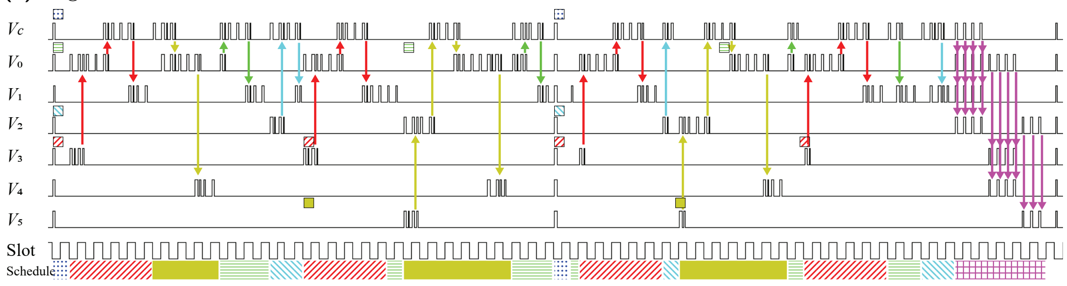
5.2 Validation of reliable dynamic scheduling

To validate the functional correctness of reliable dynamic scheduling in RD-PaS on our testbed, we let the network trigger rhythmic events, and use the logic analyzer to capture the radio activities through a physical pin on each device node and plot the waveforms. We configure the network to enter the rhythmic mode at slot 720. The hyperperiod of the task set is 360 according to Table 3. (Rhythmic events can happen at any time. We chose this integer multiple of the hyperperiod to simplify the waveform demo.) Fig. 5 illustrates a sample waveform for 240 consecutive slots (slot 600-840) in the TBS model. (Both TBS and PBS models are validated. We present the results in the TBS model here for ease of explanation.) The network runs in the nominal mode for the first 120 time slots (Fig. 5b) and then switches to the rhythmic mode in the next 120 slots (Fig. 5c). Seven waveforms represent the radio activities, either transmitting, receiving, or listening, for all the 7 nodes, as labeled on the left side of the figures. Each rising and falling edge in the *Slot* row (lower part of the figures) mark the start of a new time slot. In the *schedule* row (lower part of the figures), slot assignments are indicated using different colors.

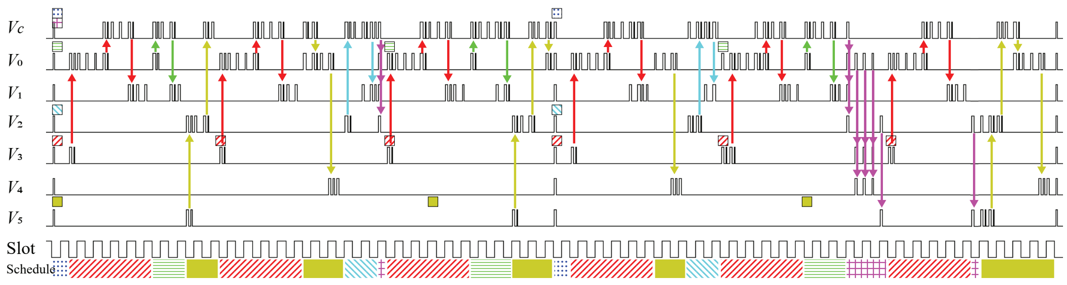
From Fig. 5b, we observe that each task τ_i releases its packets according to P_i, and

Release (τ0/τ1/τ2/τ3/τ4/τ5)
Schedule (τ0/τ1/τ2/τ3/τ4/τ5)
Transmission (τ0/τ1/τ2/τ3/τ5)

(a) Legend.



(b) Radio activities in slots 600 to 720 (nominal mode).



(c) Radio activities in slots 720 to 840 (rhythmic mode). Task τ_0 is in the rhythmic state and releases packets following $\overrightarrow{P}_0$ given in Table 3.

Figure 5 Slot information and radio activities in the reliable dynamic scheduling test case captured by the logic analyzer.

w_i^+ number of slots are allocated to each packet before its deadline (shown in the *schedule* row). In each scheduled slot, the sender attempts to transmit the packet and may succeed (marked by the arrows). Although some attempts fail, all the packets are still delivered to the destination node because of the right amount of retransmission slots as determined by the reliable static scheduling function. In Fig. 5c, τ_0 enters the rhythmic state, and its period is reduced according to $\overrightarrow{P}_0$ given in Table 3. Also as shown in the *schedule* row, the $W_{i,j}$ values for τ_0 do not change, while those for τ_1, τ_2, τ_3, τ_5 are reduced to $[9,9,9]$, $[4,5,5]$, $[4,4]$, $[7]$, respectively. The $\overrightarrow{R}_{i,j}$ vectors are also selected correctly by the updated $W_{i,j}$ values in the rhythmic mode, and all the packets from the rhythmic task (τ_0) are successfully delivered to the destination. The captured results match the results from the simulation, and this validates the correctness of the reliable dynamic scheduling function in RD-PaS.

6 Simulation-based Performance Evaluation

In this section, we evaluate the performance of RD-PaS through extensive simulations and compare RD-PaS with a state-of-the-art dynamic approach, D^2-PaS.[6] The first three sets of simulations compare packet delivery ratio, network bandwidth usage and number of extra slots produced by RD-PaS with those by D^2-PaS. The last set of simulations studies

[6] [26] shows that D^2-PaS has a clear advantage in packet dropping performance compared to the fully distributed scheduling framework FD-PaS, so we omit the comparison between RD-PaS and FD-PaS. Also, since we have proved the optimality of our retransmission slots assignment in Sec. 3.1, we omit to compare with the retransmission mechanism in [2] in the static setting.

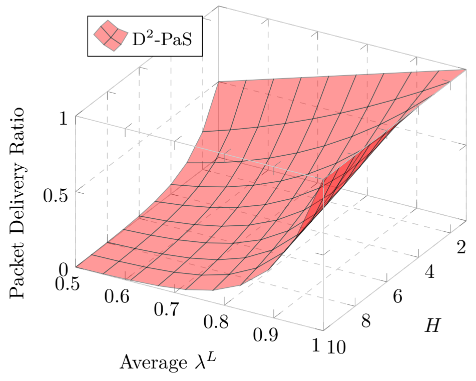


Figure 6 PDR in D^2-PaS framework.

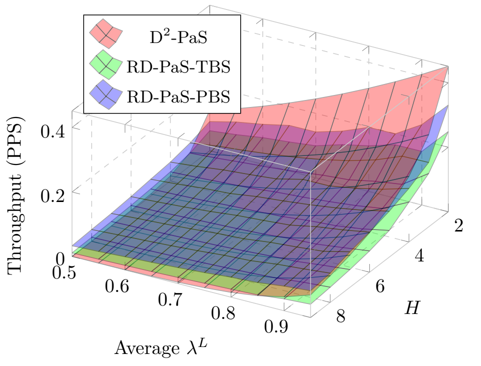


Figure 7 Throughput comparison among different scheduling frameworks.

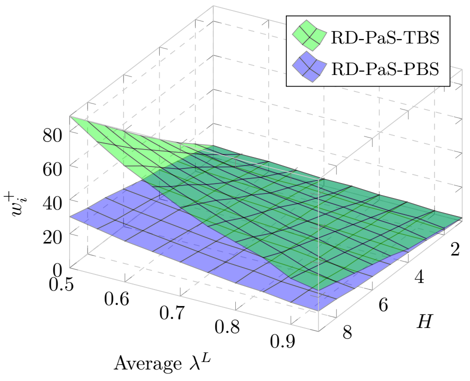


Figure 8 Comparison of w_i^+ in TBS and PBS.

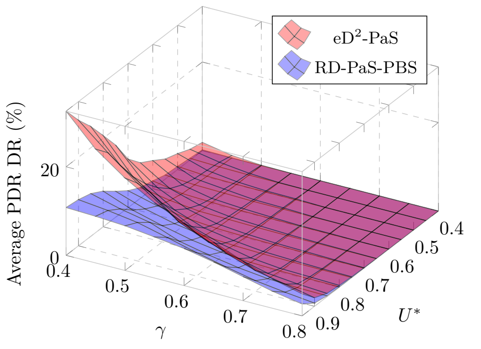


Figure 9 Comparison of the PDR degradation rate.

the behavior of the rhythmic mode. We evaluate the reliability degradation by comparing RD-PaS with D^2-PaS on handling disturbances in RTWNs.

6.1 Comparison of Packet Delivery Ratio

As RD-PaS utilizes retransmission slots to guarantee the required e2e PDR value for each task, there is no doubt that the system reliability will be improved compared with a traditional scheduling framework not considering reliablity. To quantify such improvements, we calculate the e2e PDR resulted from applying D^2-PaS in lossy links with randomly generated link PDRs. Since the e2e PDR for each task is independent, we use different settings to randomly generate tasks and compute the PDR value for each task. The number of hops for a task, H, is drawn from the uniform distribution over $\{1, 2, ..., 10\}$ and the PDR value of each link on the routing path is randomly generated by controlling the average value of link PDR, λ^L, following a uniform distribution in $\{0.5, 0.55, ..., 1\}$. As periods and deadlines do not affect the packet delivery ratio, we only study PDR's dependcy on H and λ^L. Fig. 6 shows the e2e PDR of a task as a function of λ^L and H. Because RD-PaS can always guarantee the required PDR value, its results are always at the ceiling (above 99%) of the figure and are thus omitted. From Fig. 6, we can observe the large gap between RD-PaS and D^2-PaS (60.6% on average) in guaranteeing the e2e PDR of the task.

6.2 Comparison of Network Bandwidth Usage

Allocating extra retransmission slots can significantly improve the reliability of packet delivery. However, higher network bandwidth is required which may affect system schedulability. In this set of experiments, we study the efficiency of using time slots to deliver packets, in different scheduling frameworks, according to the performance metric *throughput*. Throughput is defined as the number of packets delivered per slot (PPS) and is the ratio between the e2e PDR value and the number of allocated slots assigned to the task, i.e. $\frac{\lambda_i^*(w)}{w}$. The parameter settings of this set of experiments are the same as that in Section 6.1.

Fig. 7 summarizes throughputs for different scheduling frameworks with varied average link PDR λ^L and the number of hops, H, for the generated task. From the results, we can observe that D^2-PaS has a higher throughput when H is small and when λ^L is close to 1. However when the link PDR drops and H increases, RD-PaS (in both TBS and PBS models) gains better throughput. This is mainly due to the fact that using a time slot for retransmission can gain more throughput than transmitting a new packet in these cases. The simulation results also show that RD-PaS in the PBS model can always achieve a better throughput than in the TBS model. The reason is that the PBS model can always achieve same PDR with less number of slots, compared to the TBS model due to the PBS's ability in sharing slots among transmissions of a packet.

6.3 Comparison of Required Numbers of Slots

In this set of experiments, we make further evaluation on RD-PaS in TBS and PBS models. As discussed in Section 4, the PBS model provides more flexibility on the retransmission slot assignment, and a less number of slots, w_i^+, is required to achieve the same λ^R as compared to the TBS model. Fig. 8 gives the comparison on the required number of slots under different settings of average λ^L and H, and the required end-to-end PDR value λ^R is set to 99%. As can be observed, tasks in PBS model require less number of slots than in TBS model, when $H > 1$. The required number of slots in the PBS model is 55.0% less on average compared to that in TBS model. This is consistent with the observation that one packet requires less number of slots to achieve the same λ^R in the PBS model.

6.4 Effectiveness in Handling Rhythmic Events

To evaluate the performance of RD-PaS in handling rhythmic events, we compare the *degradation rate (DR)* between RD-PaS and D^2-PaS. DR is defined as the ratio between the sum of reliability degradation (i.e., $\delta_{i,j}$) from all periodic packets and the total number of generated periodic packets in the rhythmic mode. As D^2-PaS does not consider unreliable wireless links, we first extend D^2-PaS to support reliable transmission, denoted as eD^2-PaS. Specifically, all packets in eD^2-PaS are reliably transmitted using w_i^+ slots in the static schedule. In the dynamic schedule, transmission and retransmission slots assigned for each packet are not differentiated, i.e., each packet can either be reliably scheduled or dropped.

To better control the system workload, we vary the nominal utilization of the task set. Specifically, we use a random periodic task set generated according to a target nominal utilization U^*. The generation of each random task τ_i is controlled by the following parameter settings: i) the number of hops H_i is drawn from the uniform distribution over $\{2, 3, ..., 16\}$, ii) the nominal period P_i is equal to deadline D_i and follows a uniform distribution in $\{50, 51, ...100\}$. As the simulation results in the last sub-section have shown, the PBS model requires less total number of slots to achieve the same transmission reliability. Thus, here we use the PBS model to generate the PDR function $\lambda_i^*(\cdot)$ for each task τ_i.

After a task set is generated, we randomly select two tasks to be the rhythmic tasks. To better control the workload of the rhythmic event, we assume that all the rhythmic periods (deadlines) are the same in $\overrightarrow{P}_i(\overrightarrow{D}_i)$ and the number of elements in $\overrightarrow{P}_i$ equals to 10. The value of each element $P_{i,R}$ is thus controlled by the rhythmic period ratio, $\gamma = \frac{P_{i,R}}{P_i}$.

Fig. 9 shows the results of DR as a function of both the nominal task set utilization U^* and the rhythmic period ratio γ. Each data point is the average value of $1,000$ trials. From Fig. 9, we can observe that RD-PaS has a lower PDR degradation rate (58.4% on average) over eD^2-PaS. The main reason is that eD^2-PaS either schedules or drops any packet $\chi_{i,j}$, i.e. $W_{i,j} \in \{0, w_i^+\}$. However, RD-PaS has more flexibility on tuning the number of slots assigned to $\chi_{i,j}$, i.e. $W_{i,j} \in \{0, H_i, \ldots, w_i^+\}$.

7 Conclusion and Future Work

In this paper, we present RD-PaS, a reliable dynamic packet scheduling framework for RTWNs. RD-PaS provides guaranteed reliability of packet delivery in RTWNs for both transmission-based scheduling model and packet-based scheduling model in a hybrid manner. In the presence of unexpected disturbances, RD-PaS makes dynamic schedule adjustment judiciously to guarantee timely and reliable delivery of the critical rhythmic packets while minimizes reliability degradation for noncritical packets. A provably optimal algorithm (for the static case) as well as a heuristic (for the dynamic case) are introduced for realizing RD-PaS. Extensive testbed and simulation based experiments are conducted to validate the correctness and effectiveness of RD-PaS. Our experimental results show that RD-PaS can significantly improve the QoS (in terms of reliability) compared with the state-of-the-art approaches. As future work, we will extend RD-PaS to further support RTWNs with multi-channel scheduling and multi-path routing capabilities, and evaluate its performance in large-scale RTWN testbeds.

References

1 Johan Åkerberg, Mikael Gidlund, and Mats Björkman. Future research challenges in wireless sensor and actuator networks targeting industrial automation. In *2011 9th IEEE International Conference on Industrial Informatics*, pages 410–415, July 2011. `doi:10.1109/INDIN.2011.6034912`.

2 Ryan Brummet, Dolvara Gunatilaka, Dhruv Vyas, Octav Chipara, and Chenyang Lu. A Flexible Retransmission Policy for Industrial Wireless Sensor Actuator Networks. In *2018 IEEE International Conference on Industrial Internet (ICII)*, pages 79–88, October 2018. `doi:10.1109/ICII.2018.00017`.

3 Yu Chen, Hongwei Zhang, Nathan Fisher, Le Yi Wang, and George Yin. Probabilistic Per-Packet Real-Time Guarantees for Wireless Networked Sensing and Control. *IEEE Transactions on Industrial Informatics*, 14(5):2133–2145, May 2018. `doi:10.1109/TII.2018.2795567`.

4 Octav Chipara, Chengjie Wu, Chenyang Lu, and William Griswold. Interference-Aware Real-Time Flow Scheduling for Wireless Sensor Networks. In *2011 23rd Euromicro Conference on Real-Time Systems*, pages 67–77, July 2011. `doi:10.1109/ECRTS.2011.15`.

5 Li Da Xu, Wu He, and Shancang Li. Internet of Things in Industries: A Survey. *IEEE Transactions on Industrial Informatics*, 10(4):2233–2243, November 2014. `doi:10.1109/TII.2014.2300753`.

6 Diego Dujovne, Thomas Watteyne, Xavier Vilajosana, and Pascal Thubert. 6TiSCH: deterministic ip-enabled industrial internet (of things). *IEEE Communications Magazine*, 52(12):36–41, December 2014. `doi:10.1109/MCOM.2014.6979984`.

7 Vehbi C Gungor, Gerhard P Hancke, et al. Industrial Wireless Sensor Networks: Challenges, Design Principles, and Technical Approaches. *IEEE Transactions on Industrial Electronics*, 56(10):4258–4265, October 2009. `doi:10.1109/TIE.2009.2015754`.

8 Song Han, Xiuming Zhu, Aloysius K Mok, Deji Chen, and Mark Nixon. Reliable and Real-Time Communication in Industrial Wireless Mesh Networks. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 3–12, April 2011. `doi:10.1109/RTAS.2011.9`.

9 Shengyan Hong, Xiaobo Sharon Hu, Tao Gong, and Song Han. On-Line Data Link Layer Scheduling in Wireless Networked Control Systems. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 57–66, July 2015. `doi:10.1109/ECRTS.2015.13`.

10 ISA Standard. Wireless systems for industrial automation: process control and related applications. *ISA-100.11 a-2009*, 2009.

11 Junsung Kim, Karthik Lakshmanan, and Ragunathan Raj Rajkumar. Rhythmic Tasks: A New Task Model with Continually Varying Periods for Cyber-Physical Systems. In *2012 IEEE/ACM Third International Conference on Cyber-Physical Systems*, pages 55–64, April 2012. `doi:10.1109/ICCPS.2012.14`.

12 Bo Li, Lanshun Nie, Chengjie Wu, Humberto Gonzalez, and Chenyang Lu. Incorporating Emergency Alarms in Reliable Wireless Process Control. In *Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems*, ICCPS '15, pages 218–227, New York, NY, USA, 2015. ACM. `doi:10.1145/2735960.2735983`.

13 Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 1973.

14 Chenyang Lu, Abusayeed Saifullah, Bo Li, Mo Sha, Humberto Gonzalez, Dolvara Gunatilaka, Chengjie Wu, Lanshun Nie, and Yixin Chen. Real-Time Wireless Sensor-Actuator Networks for Industrial Cyber-Physical Systems. *Proceedings of the IEEE*, 104(5):1013–1024, May 2016. `doi:10.1109/JPROC.2015.2497161`.

15 Silvano Martello, David Pisinger, and Paolo Toth. New trends in exact algorithms for the 0–1 knapsack problem. *European Journal of Operational Research*, 123(2):325–332, 2000. `doi:10.1016/S0377-2217(99)00260-X`.

16 Abusayeed Saifullah, Dolvara Gunatilaka, Paras Tiwari, Mo Sha, Chenyang Lu, Bo Li, Chengjie Wu, and Yixin Chen. Schedulability Analysis under Graph Routing in WirelessHART Networks. In *2015 IEEE Real-Time Systems Symposium*, pages 165–174, December 2015. `doi:10.1109/RTSS.2015.23`.

17 Abusayeed Saifullah, You Xu, Chenyang Lu, and Yixin Chen. Real-Time Scheduling for WirelessHART Networks. In *2010 31st IEEE Real-Time Systems Symposium*, pages 150–159, November 2010. `doi:10.1109/RTSS.2010.41`.

18 Abusayeed Saifullah, You Xu, Chenyang Lu, and Yixin Chen. End-to-End Communication Delay Analysis in Industrial Wireless Networks. *IEEE Transactions on Computers*, 64(5):1361–1374, May 2015. `doi:10.1109/TC.2014.2322609`.

19 Emiliano Sisinni, Abusayeed Saifullah, Song Han, Ulf Jennehag, and Mikael Gidlund. Industrial Internet of Things: Challenges, Opportunities, and Directions. *IEEE Transactions on Industrial Informatics*, 14(11):4724–4734, November 2018. `doi:10.1109/TII.2018.2852491`.

20 Jianping Song, Song Han, Al Mok, Deji Chen, Mike Lucas, Mark Nixon, and Wally Pratt. WirelessHART: Applying wireless technology in real-time industrial process control. In *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 377–386, April 2008. `doi:10.1109/RTAS.2008.15`.

21 Federico Terraneo, Paolo Polidori, Alberto Leva, and William Fornaciari. TDMH-MAC: Real-time and multi-hop in the same wireless mac. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 277–287, December 2018. `doi:10.1109/RTSS.2018.00044`.

22 Haibo Zhang, Pablo Soldati, and Mikael Johansson. Performance Bounds and Latency-Optimal Scheduling for Convergecast in WirelessHART Networks. *IEEE Transactions on Wireless Communications*, 12(6):2688–2696, June 2013. `doi:10.1109/TWC.2013.050313.120543`.

23 Tianyu Zhang, Tao Gong, Chuancai Gu, Huayi Ji, Song Han, Qingxu Deng, and Xiaobo Sharon Hu. Distributed Dynamic Packet Scheduling for Handling Disturbances in Real-Time Wireless Networks. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 261–272, April 2017. `doi:10.1109/RTAS.2017.11`.

24 Tianyu Zhang, Tao Gong, Song Han, Qingxu Deng, and X Sharon Hu. Distributed Dynamic Packet Scheduling Framework for Handling Disturbances in Real-Time Wireless Networks. *IEEE Transactions on Mobile Computing*, pages 1–1, 2018. `doi:10.1109/TMC.2018.2877681`.

25 Tianyu Zhang, Tao Gong, Song Han, Qingxu Deng, and Xiaobo Sharon Hu. Fully Distributed Packet Scheduling Framework for Handling Disturbances in Lossy Real-Time Wireless Networks, 2019. `arXiv:1902.02023`.

26 Tianyu Zhang, Tao Gong, Zelin Yun, Song Han, Qingxu Deng, and Xiaobo Sharon Hu. FD-PaS: A fully distributed packet scheduling framework for handling disturbances in real-time wireless networks. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, April 2018. `doi:10.1109/RTAS.2018.00007`.

27 Marco Zimmerling, Luca Mottola, Pratyush Kumar, Federico Ferrari, and Lothar Thiele. Adaptive Real-Time Communication for Wireless Cyber-Physical Systems. *ACM Transactions on Cyber-Physical Systems*, 1(2):8:1–8:29, February 2017. `doi:10.1145/3012005`.

Isolation-Aware Timing Analysis and Design Space Exploration for Predictable and Composable Many-Core Systems

Behnaz Pourmohseni
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
behnaz.pourmohseni@fau.de

Fedor Smirnov
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
fedor.smirnov@fau.de

Stefan Wildermann
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
stefan.wildermann@fau.de

Jürgen Teich
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
juergen.teich@fau.de

Abstract

Composable many-core systems enable the independent development and analysis of applications which will be executed on a shared platform where the mix of concurrently executed applications may change dynamically at run time. For each individual application, an off-line Design Space Exploration (DSE) is performed to compute several mapping alternatives on the platform, offering Pareto-optimal trade-offs in terms of real-time guarantees, resource usage, etc. At run time, one mapping is then chosen to launch the application on demand. In this context, to enable an independent analysis of each individual application at design time, so-called *inter-application isolation schemes* are applied which specify temporal/spatial isolation policies between applications. State-of-the-art composable many-core systems are developed based on a *fixed* isolation scheme that is exclusively applied to every resource in every mapping of every application and use a timing analysis *tailored* to that isolation scheme to derive timing guarantees for each mapping. A *fixed* isolation scheme, however, heavily restricts the explored space of solutions and can, therefore, lead to suboptimality. Lifting this restriction necessitates a timing analysis that is applicable to mappings with an arbitrary mix of isolation schemes on different resources. To address this issue, in this paper, we (a) present an *isolation-aware timing analysis* that – unlike existing analyses – can handle multiple isolation schemes in combination within one mapping and delivers safe yet tight timing bounds by identifying and excluding interference scenarios that can never happen under the given combination of isolation schemes. Based on the timing analysis, we (b) present a DSE which explores the choices of isolation scheme per resource within each mapping and uses the proposed timing analysis for timing verification. Experimental results demonstrate that, for a variety of real-time applications and many-core platforms, the proposed approach achieves an improvement of up to 67% in the quality of delivered mappings compared to approaches based on a fixed isolation scheme.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Embedded and cyber-physical systems; Computer systems organization → Multicore architectures

Keywords and phrases Many-core systems, timing analysis, design space exploration (DSE), isolation scheme, predictability, composability.

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.12

Acknowledgements This work is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - Project Number 146371743 - TRR 89 Invasive Computing.

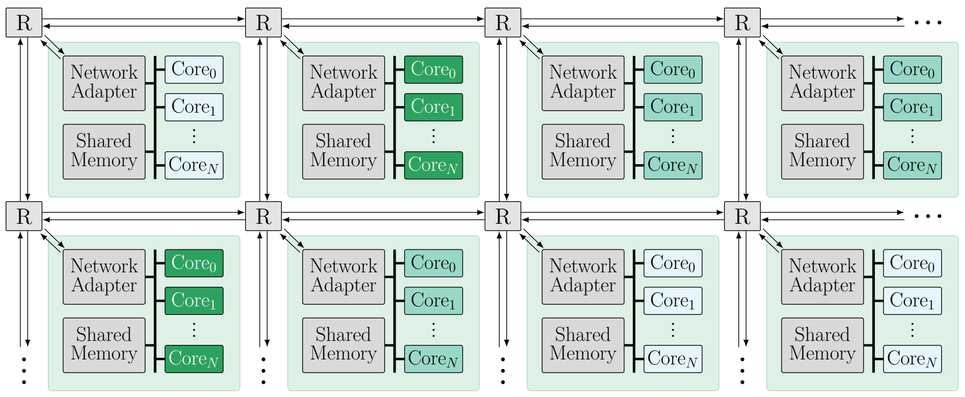


Figure 1 An example of a tiled many-core architecture composed of heterogeneous compute tiles.

1 Introduction

The ever-growing number of applications hosted in modern embedded systems introduces a high compute power demand which has given rise to the recent shift towards many-core architectures, e.g., Tilera TILE-Gx [4], Kalray MPPA-256 [9], and Intel SCC [23]. For scalability, a many-core architecture is often organized as a set of compute tiles interconnected via a Network-on-Chip (NoC), see Figure 1. Each compute tile comprises a Network Adapter (NA), a set of processing cores and peripherals such as memories which are interconnected via a set of buses. From a designer's perspective, the large number and diversity of applications and their non-functional requirements, e.g., real-time constraints, in modern embedded systems introduces an immense system design complexity. This renders the *integrated* system design approach, in which the whole system is designed at once, impractical. Over the past decade, *composable* systems, e.g. [21], have emerged to address this issue. In a composable system, applications are temporally and/or spatially isolated from each other, enabling an *incremental* system design approach where each application is first developed and analyzed individually and then added to the system on demand [1].

Composability is particularly crucial for the development of dynamic many-core systems with variable workloads and hard real-time requirements. In such systems, several independent applications are executed simultaneously, each being launched and terminated on demand and independently from others, resulting in a dynamic mix of active applications and, thus, a dynamic availability of platform resources. Each running application may be exposed to a variable workload which corresponds to a dynamic compute power demand to, e.g., meet real-time constraints. Moreover, unforeseeable conditions, e.g., thermal hot spots and hardware faults, affect the execution of applications that are currently running on the affected regions of the platform. Reserving resources according to the worst-possible workload scenario often leads to an immense underutilization of system resources which is typically not acceptable due to cost concerns. To cope with such dynamics in both resource availability of the system and resource demand of applications, *Hybrid Application Mapping (HAM)* strategies have emerged recently [45, 50]. In HAM, each application is developed and analyzed individually using an off-line Design Space Exploration (DSE) which computes several deployment options, so-called *mappings*, of the application on the platform. The computed mappings are ensured to offer diverse resource demand and performance guarantees to address various run-time resource-availability and workload scenarios, respectively. The mappings computed for each application are then provided to a so-called *run-time platform manager* which launches each application on demand using a precomputed mapping that satisfies the on-line performance requirements of the application and the resource constraints of the platform.

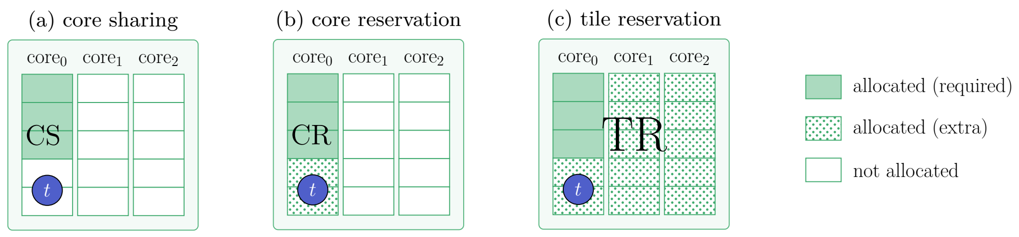


Figure 2 An example of a task t mapped to $core_0$ of a 3-core tile under (a) core-sharing, (b) core-reservation, and (c) tile-reservation isolation schemes. For a timely completion, t requires a core budget of 3 (shaded cells). Dotted cells are allocated in addition due to the isolation scheme in use.

Moreover, if the mapping in use by a running application fails, e.g., due to thermal hot spots, resource faults, or a drastic workload change, the run-time manager switches the application to another precomputed mapping which conforms to the new conditions [37].

The off-line DSE used in HAM strategies employs compute-intensive optimization and verification techniques to find high-quality mappings that offer Pareto-optimal trade-offs w.r.t. multiple – oftentimes non-linear and conflicting – design objectives, e.g., resource usage, latency, energy, etc. To achieve *predictability*, the worst-case timing properties of each mapping are bounded based on the choice of allocated resources by deriving the worst-case timing interferences that may be imposed by other (concurrent) applications which share resources with the mapping under analysis [48]. During the DSE, however, an application is developed individually, and the characteristics of the potential concurrent applications are not available to be considered in the timing analysis. In order to regulate the maximum degree of timing interferences that may be imposed by other applications, temporal and spatial isolation techniques are employed which apply certain restrictions on the accessibility of resources used by the mapping under analysis to other applications that run concurrently. These restrictions are referred to as so-called *inter-application isolation schemes*. Figure 2 illustrates three major isolation schemes that are predominantly used in many-core systems to establish composability. In all three cases, an exemplary task t is mapped to $core_0$ of a 3-core tile. For the sake of brevity, only cores are depicted in the illustration of the tile. The compute power of each core is divided into 5 budgets of equal size. For a timely completion, t requires a budget of 3 on $core_0$. In the following, each isolation scheme is explained.

- **Core Sharing (CS).** In the isolation scheme referred to as core sharing, illustrated in Figure 2a, only the 3 core budgets required for t are allocated for the mapping. Hence, concurrent applications are *temporally isolated* from the current mapping on $core_0$ and can use the 2 not-allocated budgets of $core_0$, the whole budget of $core_1$ and $core_2$, and any other on-tile resource, e.g., memories and the NA. Core sharing is the least restrictive isolation scheme which merely depends on temporal isolation on all resources.
- **Core Reservation (CR).** In the isolation scheme referred to as core reservation, illustrated in Figure 2b, $core_0$ is allocated as a whole, regardless of the budget demand of t. This realizes a *spatial isolation* from other applications on $core_0$, eliminating interferences from them on $core_0$ and resulting in an alleviated worst-case latency compared to core sharing. Note that interferences may still arise on other on-tile resources, e.g., memory buses used by t, as applications are temporally isolated from each other on those resources.
- **Tile Reservation (TR).** In the isolation scheme referred to as tile reservation, illustrated in Figure 2c, the compute tile is allocated as a whole, eliminating any interference from concurrent applications on any on-tile resource. This is the most restrictive isolation scheme which realizes a *spatial isolation* from concurrent applications at tile level and enables the largest reduction in the worst-case timing interferences imposed on t.

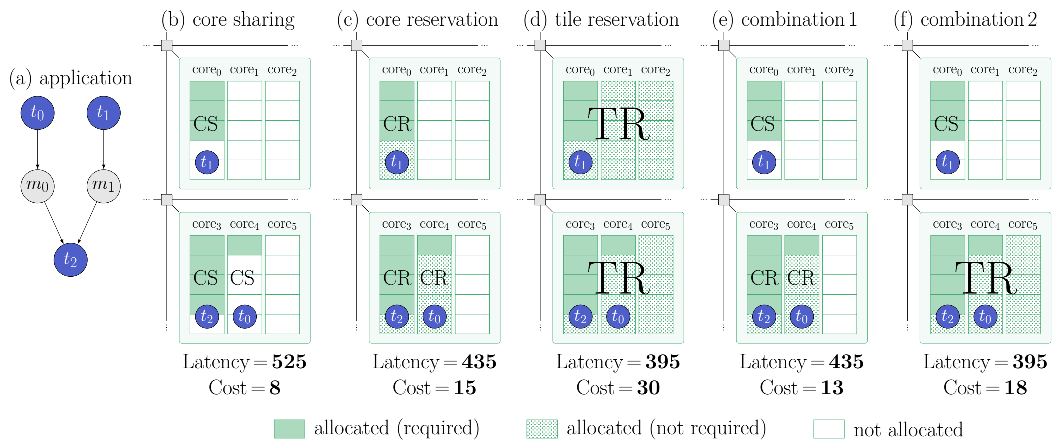

(a) application
t_0
t_1
m_0
m_1
t_2
(b) core sharing
(c) core reservation
(d) tile reservation
(e) combination 1
(f) combination 2
core$_0$ core$_1$ core$_2$
core$_3$ core$_4$ core$_5$
CS
CR
TR
Latency = **525** Cost = **8**
Latency = **435** Cost = **15**
Latency = **395** Cost = **30**
Latency = **435** Cost = **13**
Latency = **395** Cost = **18**
allocated (required)
allocated (not required)
not allocated

Figure 3 (a) an example of an application. A mapping of the application on two adjacent tiles is illustrated for the following isolation scenarios: (b) exclusive core sharing, (c) exclusive core reservation, (d) exclusive tile reservation, (e) core sharing on core$_0$ and core reservation on core$_3$ and core$_4$, (f) core sharing on core$_0$ and tile reservation on the lower tile. The core budget required for each task is shaded on the respective core. For each scenario, the worst-case latency of the mapping (derived based on Table 1a–b) and its allocation cost are given below the respective sub-figure.

Noteworthy, sharing the NoC which interconnects the tiles can hardly be avoided [34]. Hence, applications are always *temporally isolated* from each other on the NoC. State-of-the-art composable many-core systems are designed based on a *fixed isolation scheme* that is uniformly applied to *every* core/tile of *every* mapping of *every* application in the system and is coupled with a timing analysis *tailored* to that specific isolation scheme to derive tight timing guarantees. The choice of isolation scheme highly impacts the resource usage and worst-case timing characteristics of the mappings. As an example, consider the application shown in Figure 3a and a mapping of it on two adjacent 3-core tiles illustrated in Figure 3b–f. For the sake of this motivational example, assume that the timing analysis of the tasks yields the qualitative Worst-Case Response Time (WCRT) values given in Table 1a under different isolation schemes. Likewise, the qualitative Worst-Case Traversal Time (WCTT) of message m_1 (communicated between the two tiles, from t_1 to t_2) is given in Table 1b for various combinations of isolation scheme on m_1's source and destination cores/tiles. Message m_0 is implicitly communicated between t_0 and t_2 via the tile's shared memories. In this example, applying core sharing exclusively, as illustrated in Figure 3b, offers the minimum allocation cost, i.e., 8 core budgets, at the expense of considerably high worst-case

Table 1 Qualitative timing properties assumed for tasks and messages in the example illustrated in Figure 3: (a) WCRT of tasks t_0, t_1, and t_2 under different isolation schemes, (b) WCTT of message m_1 for different combinations of isolation scheme on m_1's source and destination cores/tiles.

(a) Worst-Case Response Time assumed for t_0–t_2

isolation scheme	WCRT(t_0)	WCRT(t_1)	WCRT(t_2)
TR	290	75	105
CR	315	115	120
CS	380	165	145

(b) Worst-Case Traversal Time assumed for m_1

source isolation scheme	destination isolation scheme	WCTT(m_1)
TR	TR	12
TR	CR/CS	15
CR/CS	TR	17
CR/CS	CR/CS	20

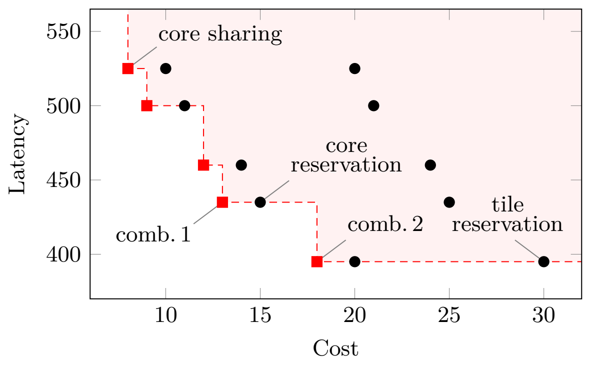


Figure 4 The latency-cost trade-offs offered by the 15 possible isolation scheme scenarios for the application mapping from Figure 3. Pareto-optimal trade-offs are designated by red squares, dominating the highlighted space. Labeled trade-offs correspond to the scenarios illustrated in Figure 3.

interferences on cores and other on-tile resources, resulting in an end-to-end application latency of 525 time units in the worst case. Applying core reservation exclusively, as depicted in Figure 3c, results in an elevated allocation cost of 15 budgets. It, however, eliminates core interferences which alleviates the worst-case application latency to 435 time units. Applying tile reservation exclusively, as depicted in Figure 3d, minimizes the worst-case application latency to 395 time units at the expense of a maximized allocation cost of 30 budgets.

Motivation. A *fixed isolation scheme* that is always applied exclusively – which is the common practice in state-of-the-art composable many-core systems – heavily restricts the space of explored mappings and excludes numerous promising solutions where multiple isolation schemes are applied *in combination*. For instance, in the example above, considering arbitrary combinations of isolation schemes within one mapping enables 12 additional isolation scenarios, two of which are depicted in Figure 3e–f. In combination 1, core sharing is applied on core$_0$ and core reservation is applied on core$_3$ and core$_4$, offering the same latency as the exclusive core-reservation scenario from Figure 3c, but at a lower cost. Likewise, combination 2 applies core sharing on core$_0$ and tile reservation on the lower tile, outperforming the exclusive tile-reservation scenario from Figure 3d by offering the same latency at a considerably lower cost. Having evaluated all possible combinations of isolation scenarios, Figure 4 illustrates the obtained cost-latency trade-offs where Pareto-optimal trade-offs are designated by red squares, dominating the highlighted space above the red dashed line. The trade-offs corresponding to the scenarios in Figure 3 are labeled accordingly in Figure 4. As shown, nearly always, Pareto-optimal trade-offs are obtained only when multiple isolation schemes are applied in combination within one mapping. In Section 6, we experimentally verify this observation for realistic use cases as well.

Contribution. The example above demonstrates that using multiple isolation schemes in combination yields mappings of a better quality trade-off. Deriving safe bounds on the worst-case timing characteristics of such mappings, however, requires an *isolation-aware timing analysis* that can capture the impact of the isolation scheme of each core/tile on the worst-case timing behavior of tasks/messages affected by it and, hence, on the worst-case timing behavior of the application. On the one hand, existing timing analyses applicable to composable many-core systems are *tailored* to one fixed isolation scheme and cannot handle multiple isolation schemes in combination within one mapping. On the other hand, from an architectural point of view, many-core platforms are undergoing a transition from single-core tiles, e.g., in [4], to multi-core tiles, e.g., in [9]. This transition introduces new sources of timing interference within each tile, e.g., on shared memory buses, which must be accounted for to obtain safe timing guarantees. In the context of composable systems, there exist no timing analysis known to us which accounts for all sources of timing interference that may arise in a many-core platform with multi-core tiles (without applying restrictive spatial isolation schemes that eliminate on-tile inter-application interferences altogether). Provided with an isolation-aware timing analysis, the off-line DSE in HAM strategies can be extended to also explore the choices of isolation scheme for each core/tile within each investigated mapping to obtain solutions with a better quality trade-off. In this line, the paper at hand makes the following contributions in the context of composable many-core systems:

1. We present an *isolation-aware timing analysis* which can handle arbitrary combinations of isolation schemes within one mapping and accounts for all timing interferences that may arise within a shared multi-core tile and on the NoC interconnecting tiles. It derives tight timing bounds by identifying and excluding interference scenarios that can never happen under the current combination of isolation schemes.
2. We present an *isolation-aware DSE* which explores the choices of isolation scheme for each core/tile in each investigated mapping of the application under analysis and uses the proposed timing analysis for the scheduling and timing verification of the mappings.

For a variety of hard real-time applications and many-core platforms, we experimentally verify the advantage of the proposed isolation-aware exploration and timing analysis approach over existing fixed-isolation-scheme approaches in terms of the quality of obtained mappings.

Organization. The remainder of this paper is organized as follows. In Section 2, related work on timing analysis and DSE of multi-/many-core systems is reviewed. Section 3 presents the preliminaries and the system model for this work. The proposed isolation-aware DSE and timing analysis are presented in Sections 4 and 5, respectively. Experimental results are discussed in Section 6 before the paper is concluded in Section 7.

2 Related Work

Worst-case timing analysis of applications in multi-/many-core systems has long been conducted in two steps: First, a context-independent analysis is applied to bound the Worst-Case Execution Time (WCET) of each task in isolation, i.e., in absence of interferences. An overview of tools and methods for context-independent WCET analysis is provided in [51]. Following the context-independent analysis, an interference analysis is performed to bound the additional latencies that may be imposed on shared resources due to external interferences.

Multi-/many-core timing analyses predominantly focus on worst-case interference analysis based on the context-independent characteristics of each task and message. For instance, the analyses presented in [2, 6, 8, 15, 17, 40, 41, 46] bound the WCRT of tasks in a multi-core setup where several cores are connected to one or more memories over shared buses. In [2, 8], a framework for multi-core response time analysis is presented for a preemptive Fixed-Priority (FP) core scheduling policy coupled with multiple memory bus arbitration policies, e.g., Time-Division Multiplexing (TDM), Round-Robin (RR), and FP. Targeting mixed-criticality systems, [15, 17] analyze WCRT under a non-preemptive FP core scheduling policy and a RR memory bus arbitration. The authors of [40, 46] present response time analyses for non-preemptive FP core scheduling policy and a multi-level bus arbitration scheme which combines RR and FP policies. In [41], memory bus interference is bounded using an ILP-based timing analysis under a RR bus arbitration policy. The framework presented in [6], analyzes memory bus interference under a variety of arbitration policies, e.g., TDM and FP. Concerning the NoC, in [43] and [54] NoC transfer delays are analyzed under FP and RR link arbitration policies, respectively.

The works listed above consider – to some extent – non-preemptive and/or *contention-oriented* arbitration/scheduling policies, e.g., priority-based schemes, for which safe timing guarantees can be derived only if the whole set of applications accessing shared resources is known at the time of analysis. In a composable system, however, the mix of concurrently running applications that share some resources may not be known at design time. Hence, contention-oriented policies can be applied in a composable system only if each and every resource with such an arbitration scheme is exclusively used by one application. To realize this on state-of-the-art many-core platforms, each mapping computed for each application must (a) have a tile-reservation isolation scheme only, (b) be restricted within one tile only, and (c) not rely on inter-tile communications, e.g., I/O transfers. In practice, however, the small number of cores comprised within one tile [4, 23], on the one hand, and the high compute power demand of each application, on the other hand, necessitate the deployment of some applications over several compute tiles to meet their real-time requirements. For such applications, NoC links could – in theory – be exclusively reserved per application to enable the timing analysis of multi-tile mappings using NoC delay analyses, e.g. [54, 43]. In practice, however, sharing NoC links among applications can hardly be avoided [34].

Contention-free arbitration policies based on time slicing, e.g., TDM and Weighted Round-Robin (WRR), offer a practical approach for predictable inter-application resource sharing without violating composability. For instance, the authors of [48, 49] consider a NoC [22] with WRR link arbitration policy and analyze worst-case NoC delays based on the reserved link budget for each communication, independent of the other communication flows that may share the same links. For WCRT analysis, [48] considers a preemptive RR core scheduling policy which, however, restricts its scope of coverage to core interferences originating from within the application under analysis only. In [49], on the other hand, a WRR core scheduling policy is considered, and based on that, a response time analysis is presented which accounts also for core interferences that may be imposed by other (currently unknown) applications. Both [48, 49], however, consider single-core tiles and, hence, cannot capture NA and memory bus interferences that may arise in a *multi-core tile*, e.g., in [9, 23]. Contrarily, we present a timing analysis that captures also the NA and memory bus interferences in multi-core tiles.

From an isolation scheme point of view, existing multi-/many-core timing analyses are tailored to a fixed isolation scheme. For instance, the analyzes in [2, 6, 8, 15, 17, 40, 41, 46] are applicable only under tile-reservation isolation scheme. The analysis presented in [48] is applicable to systems with single-core tiles only and assumes tile-reservation isolation scheme. Authors in [49] consider single-core tiles and core-sharing isolation scheme. The timing analysis presented in this paper is the first timing analysis known to us which can handle arbitrary mixes of isolation schemes in combination within one mapping.

Application mapping in multi-/many-core systems is typically viewed as a multi-objective optimization problem and is known to be NP-hard [14]. Due to its immensely large space of possible mapping solutions, using exact optimization approaches, e.g., enumeration or (integer-)linear programming, to solve this NP-hard problem demands an extremely high computational effort and is prohibitively time-consuming, except for very small/simple problems. In this context, *meta-heuristic optimization* approaches, e.g., evolutionary/genetic algorithms [7, 13, 12], simulated annealing [27], and particle swarm optimization [25], enable a scalable optimization approach that can deliver high-quality solutions at a reasonable time- and computational effort. As a result, they have become the de facto standard approach for multi-objective application mapping optimization in multi-/many-core systems. For instance, in [33, 36, 48, 49], evolutionary/genetic algorithms are used for mapping optimization. In [16, 17], mapping optimization is realized using simulated annealing algorithms, while [42] adopts particle swarm optimization. The majority of existing works on mapping optimization in HAM methodologies employ *population-based* meta-heuristics, e.g., particle swarm optimization or evolutionary/genetic algorithms, to collect a so-called *population* of best mappings. During the Design Space Exploration (DSE), the optimizer follows the course of several iterations to generate new mappings and evaluate them w.r.t. the given set of design objectives, e.g., resource cost, timing, or energy. It collects a population of explored mappings which is updated per iteration to retain the best solutions found so far and is used for the generation of mappings in the next iteration. Like [33, 36, 48, 49], we employ a multi-objective evolutionary algorithm for mapping optimization.

To the best of our knowledge, existing DSE proposals on mapping optimization in the context of HAM strategies, e.g., [26, 33, 36, 47, 44, 48, 49, 53], consider a fixed isolation scheme that is applied exclusively to *every* resource (core or tile) of *every* explored mapping of *every* application. For instance, the DSE approaches in [36, 33, 53] assume a core-reservation isolation scheme. In [26, 44, 48], a tile-reservation isolation scheme is assumed. In [49], a core-sharing isolation scheme is assumed. The DSE proposed in this paper does not assume a globally-fixed isolation scheme. Instead, it explores the choices of isolation scheme per allocated resource within each explored mapping. This provides a fine-grained control over the degree of admitted inter-application interferences (which we analyze using the proposed timing analysis) and renders many mappings with promising quality trade-offs reachable to the DSE which are not reachable under a fixed isolation scheme.

3 Preliminaries

3.1 Mapping Optimization Problem Specification

Similar to any other optimization problem, the multi-/many-core mapping optimization problem requires a problem model which describes the space of possible solutions and the conditions that must be satisfied by a mapping to be regarded a valid solution. In this paper, we use the graph-based system model from [3]. This model represents the mapping optimization problem by a so-called *specification* which describes the entire design space and is used to generate different mapping solutions. The specification and each mapping generated based on that consist of an *application graph*, an *architecture graph*, and *mapping edges* connecting them, which will be introduced in Sections 3.1.1 to 3.1.3.

3.1.1 Application Model

We consider periodic hard real-time applications. Each application is specified by an application graph $G_P(T \cup M, E)$ where T denotes the set of (processing) tasks, M denotes the set of messages exchanged between tasks, and edges $e \in E$ define data dependencies among tasks and messages, e.g., see Figure 5a. For each task $t \in T$, the execution period $\mathrm{PRD}(t)$, the context-independent worst-case execution time $\mathrm{WCET}(t, c)$ on each mappable core c, and the memory demand $\mathrm{MD}(t)$ (maximum number of memory accesses) per execution iteration

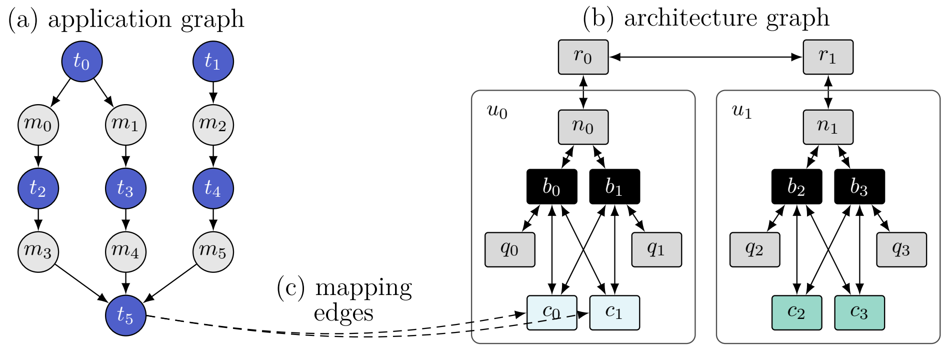


Figure 5 Specification of an exemplary mapping optimization problem, composed of (a) application graph, (b) architecture graph, and (c) mapping edges connecting them (depicted only for task t_5).

are known. For each message $m \in M$, the transfer period $\mathrm{PRD}(m)$ and the maximum payload size $\mathrm{PLD}(m)$ together with its corresponding memory demand $\mathrm{MD}(m)$ (number of memory accesses for reading/writing m from/to memory) are given.

3.1.2 Architecture Model

We consider heterogeneous tiled many-core architectures composed of multi-core tiles interconnected by a NoC. The platform architecture is specified by an architecture graph $G_A = (R \cup N \cup B \cup C \cup Q \cup U, L)$, e.g., see Figure 5b. Here, $r \in R$ denotes a NoC router, $n \in N$ a NA, $b \in B$ a memory bus, $c \in C$ a processing core, and $q \in Q$ a shared memory (or memory bank) with an own memory bus, respectively. Edges $l \in L$ represent bidirectional connections between these resources. Each $u \in U$ represents a compute tile which comprises a set of cores and memories and a NA, thus, $u \subseteq N \cup B \cup C \cup Q$. Each memory (or memory bank) is accessible to cores and the NA on the same tile via a shared bus. Each NA consists of separate Transmitter (TX) and Receiver (RX) units with own ports to each memory bus. We assume many-core architectures without timing anomalies [39]. This enables a compositional analysis of execution, communication, and memory latency contributions which can be combined to derive globally safe timing guarantees.

Memory Model. We consider a No Remote Memory Access (NORMA) scheme which is commonly practiced in many-core systems to achieve scalability [32]. Under a NORMA scheme, the memories located on one tile are not accessible to resources located on other tiles, thus, data exchanges among different tiles are realized exclusively by means of explicit message passing between them. To achieve storage composability, (a) the on-tile memory space is partitioned, such that each core is given a dedicated storage space. Likewise, (b) shared caches are partitioned or disabled. In the same line, (c) the dedicated storage space of each core is dynamically partitioned among the tasks hosted by it. Finally, (d) each message is provided with a dedicated memory space in each tile where it is produced and/or consumed. To perform a memory access, the requestor must first attain the ownership of the shared bus associated with the target memory. We assume blocking and indivisible memory accesses, so that the processing progress on the requestor side is stalled during the memory operation. Each memory access is a single-word operation with a known maximum service time, eliminating burst/block operations. A requestor may perform several consecutive but non-overlapping single-word memory operations during its bus ownership interval.

NoC Model. We consider wormhole-switched NoCs with a credit-based virtual-channel flow control, for instance [22], to allow per-link bandwidth reservation and, thereby, to enable link sharing without violating composability. Under a *wormhole-switched* flow control, data packets are decomposed into control flow digits (flits) of fixed size which are then routed over the NoC independent from each other in a pipeline fashion [35]. The *virtual-channel* flow control [5] provides multiple buffers per physical link to enable transmission preemption and composable link sharing. The transmission progress of flits in each buffer of a router is controlled using so-called *credits* which reflect the availability of buffer space at the next router. This realizes a backpressure mechanism inherently.

Communication Model. Intra-tile communications, i.e., data exchanges *within* one tile, are realized through a dedicated space in the tile's shared memories. Inter-tile communications, i.e., data exchanges *between* tiles, are realized by explicit message passing over the NoC. For the latter, after the data is written by the sender into the dedicated memory space, the TX reads the data, decomposes it into flits, and injects the flits into the NoC. The flits are then routed over the NoC toward the destination tile where the RX reconstructs the data from the flits and writes it into a dedicated memory space to be read by the receiver thereafter.

Resource Arbitration. Any hardware resource, e.g., cores, memory buses or the NoC, that can be shared among multiple applications is assumed to have an arbitration policy that is both predictable and composable. While (a) *predictability* enables formal worst-case timing analysis of each application, (b) *composability* ensures that worst-case timing bounds can be derived for each application merely based on the resource budgets reserved for it, without any knowledge about other applications that may run concurrently to it and share resources with it. This definition necessitates a *contention-free* arbitration policy for each and every resource that may be shared among applications. Time-Division Multiplexing (TDM) and Weighted Round-Robin (WRR) are well-established predictable and composable arbitration policies which serve as primary candidates for composable many-core systems [19, 21, 22]. Both TDM and WRR establish temporal isolation using time-triggered preemption, dividing the access to a resource into time slots of equal length that are periodically assigned to requestors which have reserved one or more slots on that resource. TDM is not work-conserving[1] which leads to a poor average-case performance, making it an unattractive candidate for, e.g., systems hosting both real-time and best-effort applications [24]. Contrarily, WRR provides predictability and composability in a work-conserving fashion by skipping over idle slots. This enables a notable average-case performance improvement in favor of best-effort applications while allowing worst-case timing guarantees to be derived for real-time applications. We assume each shared resource has a preemptive time-triggered arbitration policy that follows similar principles as TDM and WRR.

3.1.3 Mapping Edges

In the specification of the mapping optimization problem, the application graph and the architecture graph are connected by mapping edges $V \subseteq T \times C$, e.g., see Figure 5c. Each mapping edge $v = (t, c) \in V$ indicates that task $t \in T$ can be executed on core $c \in C$.

[1] In a work-conserving arbitration policy, time slots are assigned only to requestors with pending requests while idle slots, i.e., those without an access request, are skipped. This scheme results in a varying arbitration period and varying position of assigned slots for each requestor within one arbitration period.

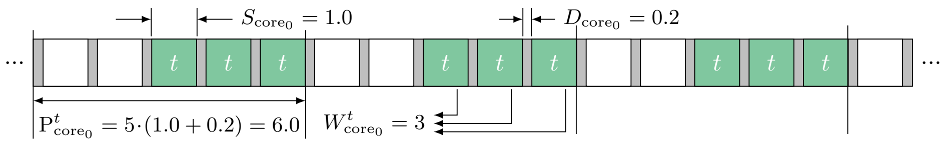

$S_{\text{core}_0} = 1.0$
$D_{\text{core}_0} = 0.2$
t
$\mathrm{P}^t_{\text{core}_0} = 5\cdot(1.0 + 0.2) = 6.0$
$W^t_{\text{core}_0} = 3$

Figure 6 Arbitration tuple of task t on core$_0$ from Figure 2a, calculated as $(1.0, 3, 6.0)$.

3.2 Arbitration Tuple

In this work, all parameters relevant for the worst-case timing analysis of a requestor on a resource, i.e., the budget reserved for it and the worst-case interference that can be imposed by other requestors, are compactly reflected by a so-called *arbitration tuple*: For each requestor x of resource r, the arbitration tuple is represented as (S_r, W^x_r, P^x_r) where S_r denotes the length of one arbitration slot of r, W^x_r denotes the number of periodic arbitration slots (weight) reserved for x on r, and P^x_r denotes the worst-case arbitration period perceived by x on r. Given the arbitration tuple, one can deduce that x has a periodic reserved time budget of $(W^x_r \cdot S_r)$ on r and, thus, experiences a worst-case wait time of $(P^x_r - W^x_r \cdot S_r)$ per arbitration period P^x_r. We exemplify the calculation of the arbitration tuple for task t in Figure 2a under an arbitration policy with a slot length of $S_{\text{core}_0} = 1.0$, an arbitration delay of $D_{\text{core}_0} = 0.2$ between consecutive slots[2], and an arbitration capacity of $K_{\text{core}_0} = 5$ slots per period. Under such an arbitration policy, task t with 3 reserved periodic slots perceives an arbitration weight of $W^t_{\text{core}_0} = 3$ and a worst-case arbitration period of $P^t_{\text{core}_0} = 5 \times (1.0 + 0.2) = 6.0$ based on which the arbitration tuple is created as $(1.0, 3, 6.0)$. This calculation is also illustrated in Figure 6. Note that, the arbiter delay is reflected in the calculation of arbitration period, rendering the arbitration tuple expressive of realistic resource arbiters in practice.

It may happen that the isolation scheme of a resource leads to one or more arbitration slots to be allocated by the mapping under analysis but not utilized. Given a work-conserving arbitration policy, e.g. WRR, in such cases we reduce the arbitration capacity of that resource to exclude those slots that are never utilized and, hence, are always skipped by the arbiter. For instance, in Figure 2b, core$_0$ is allocated exclusively where only 3 slots are utilized by t while the remaining 2 slots are never utilized. Thus, given a work-conserving arbitration policy, the arbitration period is guaranteed not to exceed $P^t_{\text{core}_0} = (5 - 2) \times (1.0 + 0.2) = 3.6$, resulting in an adapted arbitration tuple of $(1.0, 3, 3.6)$ for t on core$_0$.

The proposed timing analysis presented in Section 5 takes the arbitration tuples as input and derives safe bounds on the worst-case timing characteristics of each task and message under the combination of applied isolation schemes. For the calculation of the arbitration tuples, the arbitration policy of each resource r and its parameters, namely, arbitration slot length S_r, arbitration delay D_r, and arbitration capacity K_r, are provided by the architecture, while both the isolation scheme of each resource r and the number W^x_r of slots reserved for each analyzed requestor x are decided by the mapping optimizer during the DSE. We calculate arbitration tuples for the following requestors:

- For each task $t \in T$, the arbitration tuple on core $c \in C$ executing t is calculated and denoted as (S_c, W^t_c, P^t_c).
- For each inter-tile message $m \in M$, the arbitration tuple (a) on the TX tx injecting m into the NoC, (b) on the NoC route ρ (sequence of links) over which m is routed, and (c) on the RX rx receiving m from the NoC are calculated and denoted as $(S_{tx}, W^m_{tx}, P^m_{tx})$, $(S_\rho, W^m_\rho, P^m_\rho)$, and $(S_{rx}, W^m_{rx}, P^m_{rx})$, respectively.

[2] The arbitration delay denotes the latency of the arbiter for switching between consecutive arbitration slots. For instance, on a core, it corresponds to the context-switch overhead of the operating system.

- For each core $c \in C$ that has at least one task mapped to it, the arbitration tuple on each on-tile memory bus $b \in B$ is calculated and denoted as (S_b, W_b^c, P_b^c).
- For each TX tx that routes at least one outbound message out of the tile, the arbitration tuple on each on-tile memory bus $b \in B$ is calculated and denoted as $(S_b, W_b^{tx}, P_b^{tx})$.
- For each RX rx that routes at least one incoming message into the tile, the arbitration tuple on each on-tile memory bus $b \in B$ is calculated and denoted as $(S_b, W_b^{rx}, P_b^{rx})$.

4 Isolation-Aware Design Space Exploration (DSE)

This section presents our isolation-aware DSE approach, illustrated in Figure 7. The DSE takes as input the specification of the mapping optimization problem comprising the application graph, the architecture graph, and mapping edges, and delivers a set of mappings that offer Pareto-optimal trade-offs w.r.t. a given set of design objectives, e.g., latency, throughput, energy, and resource usage. The DSE employs a *mapping optimizer* to explore the space of possible mappings of the application on the target architecture. The optimizer creates each mapping by conducting a series of binding, isolation, routing, allocation, and scheduling design decisions, elaborated later in Section 4.1. Once a mapping is generated, it is provided to a set of *evaluators* to assess the quality of the mapping w.r.t. the given design objectives. The proposed isolation-aware timing analysis, which will be presented in Section 5, is used here as an evaluator to derive safe bounds for timing-related design objectives, namely, latency and throughput. The optimizer uses an evolutionary algorithm which conducts several iterations to generate new mappings and collect a set of Pareto-optimal mappings.

The Pareto-optimal mappings will be used at run time to launch the application on demand by selecting a mapping that complies with the current real-time constraints of the application and resource availability of the system (restricted due to, e.g., other running applications). Since our timing analysis accounts for the worst-case interferences that may be imposed by any mix of (statically unknown) concurrent applications, the worst-case timing characteristics it provides for each mapping are guaranteed to hold regardless of the mix and deployment of other applications at run time. This allows the DSE to be performed for each application *individually* without any knowledge about the other applications in the system.

4.1 Mapping Creation

The creation of each mapping starts by making the *binding* and the *isolation* design decisions:

- **Binding.** In this step, each task of the application is bound to a core on the architecture. We use the SAT-Decoding approach [29] to explore the binding of tasks to cores. By encoding the constraints from [31], we ensure that each task is bound exactly to one core.
- **Isolation.** In this step, an isolation scheme is selected for each core and each tile. To this end, each core and each tile is decided to be either *shared* or *reserved.* We implement the exploration of isolation schemes using the SAT-Decoding approach [29], see also [18].

The creation of the mapping is completed by deriving the implications of binding and isolation decisions on message *routing*, resource *allocation*, and *scheduling* of tasks and message:

- **Routing.** In this step, for each inter-tile message (i.e., a message communicated between two tasks which, according to the binding decisions, are bound to different tiles), a NoC route (sequence of links) is determined over which the message is transferred between the two tiles. Without loss of generality, we use the XY-routing algorithm [35].
- **Allocation.** In this step, the processor budget required for the mapping is allocated according to the previously made binding and isolation decisions: If a task is bound onto any core of a reserved tile, the whole tile is allocated completely and, thus, none of the

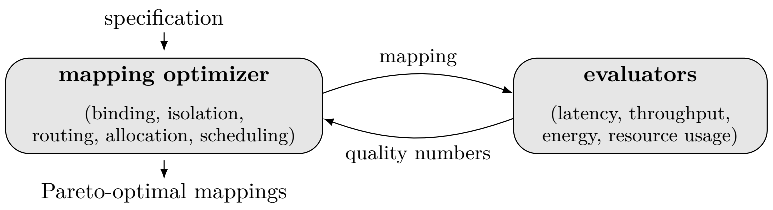

specification
mapping optimizer
(binding, isolation, routing, allocation, scheduling)
Pareto-optimal mappings
mapping
quality numbers
evaluators
(latency, throughput, energy, resource usage)

Figure 7 Overview of the proposed isolation-aware design space exploration (DSE) approach.

resources on that tile can be used by other applications (tile reservation). If a task is bound to a reserved core on a shared tile, that particular core is allocated completely while other resources on the tile can be used by other applications (core reservation). Finally, if a task is bound to a shared core on a shared tile, only the minimum budget required by that task will be allocated on that core while the remaining core budget can be used by other applications (core sharing). Any core and tile that does not correspond to one of the cases above is not allocated and, therefore, can be used by other applications. We also allocate the minimum budget required for each inter-tile message on NoC links that are part of the message's route specified in the routing step.

- **Scheduling.** We elaborate on the scheduling step in Section 4.2.

4.2 Isolation-Aware Scheduling

In the scheduling step, the arbitration tuples listed in Section 3.2 which are required for the timing analysis of the mapping are calculated. In what follows, we first calculate in Section 4.2.1 the resource budgets (arbitration weight) required for tasks and messages. Based on these weights, the arbitration tuples are then calculated in Section 4.2.2.

4.2.1 Resource Budget Calculation

To calculate the arbitration weight of each task/message $x \in T \cup M$ on each respective resource r, first the worst-case arbitration period P_r^x for x on r is calculated (as presented in Section 3.2) based on r's arbitration capacity K_r and arbitration slot length S_r which are provided by the architecture. Then, for each task t bound to core c, we start with an arbitration weight of $W_c^t = 1$ and iteratively (I) construct the arbitration tuple (S_c, W_c^t, P_c^t) and (II) use Equations (1) to (4) from Section 5 to determine t's WCRT for the current arbitration weight W_c^t. If the WCRT of t exceeds its deadline given by the application, i.e., $\text{WCRT}(t) > \text{PRD}(t)$, we (III) increment W_c^t by one and go back to step (I). Otherwise, the iterations are terminated and the current arbitration weight is considered for t. Likewise, for each inter-tile message m routed via TX unit tx, RX unit rx, and NoC route ρ, we use Equations (5) to (8) from Section 5 to calculate the minimum arbitration weight $W_{tx}^m = W_{rx}^m = W_\rho^m$ such that m's WCTT does not exceed its production period, i.e., $\text{WCTT}(m) \leq \text{PRD}(m)$.

After the weights are derived, we evaluate the overall weight demanded on each resource. If the weight demanded by all tasks mapped to a core exceed the core's arbitration capacity, the mapping is considered as infeasible and is discarded. Likewise, if the weight demanded on a NoC link, a TX or a RX by messages routed over it exceeds its arbitration capacity, the mapping is considered as infeasible and discarded.

4.2.2 Arbitration Tuple Calculation

Given the arbitration weights of tasks and messages on their respective resources, calculated in Section 4.2.1, and the isolation scheme of each core/tile, the arbitration tuples for tasks and messages are calculated as follows. For each exclusively allocated core c, first the arbitration capacity is reduced as $K_c = \sum_{t \in T} W_c^t$ to discard scheduling slots that are not utilized by the tasks mapped to it, and the arbitration period P_c^t of each task $t \in T$ mapped to it is refined accordingly as presented in Section 3.2. Then, the arbitration tuple (S_c, W_c^t, P_c^t) is created for each task $t \in T$ mapped to c where the scheduling slot length S_c is provided by the architecture. Likewise, for each exclusively allocated tile, first the arbitration capacity of the TX unit tx is reduced as $K_{tx} = \sum_{m \in M} W_{tx}^m$ to reflect only the arbitration slots reserved for the outbound messages $m \in M$ of that tile within the current mapping. Then, the arbitration period P_{tx}^m of each outbound message m on tx is refined, and the arbitration tuple $(S_{tx}, W_{tx}^m, P_{tx}^m)$ is constructed. A similar procedure is followed for the incoming messages of the tile. Since the NoC is assumed to be shared, no reduction will be applied to the capacity of NoC links, and thus, no refinement is required for the calculation of the arbitration tuple $(S_\rho, W_\rho^m, P_\rho^m)$ of an inter-tile message $m \in M$ on the links of its NoC route ρ.

For each core $c \in C$, the arbitration tuple (S_b, W_b^c, P_b^c) on each memory bus $b \in B$ connected to it is calculated as follows. The bus arbitration slot length S_b, the bus arbitration capacity K_b, and the core arbitration weight W_b^c on the bus are provided by the architecture. The arbitration period P_b^c perceived by c on bus b is calculated according to the decided isolation scheme: If a tile-reservation isolation scheme is selected, the bus arbitration capacity K_b is reduced to exclude the arbitration slots corresponding to idle cores (cores that do not host any tasks). Accordingly, the refined arbitration period P_b^c is calculated as presented in Section 3.2. For each TX and RX, the arbitration tuples $(S_b, W_b^{tx}, P_b^{tx})$ and $(S_b, W_b^{rx}, P_b^{rx})$ on each memory bus $b \in B$ connected to them is calculated similarly. Note that all arbitration capacity reductions discussed above are applied only in case of a work-conserving arbitration policy, e.g., WRR. Otherwise, the arbitration capacities remain unaffected.

5 Isolation-Aware Timing Analysis

This section presents the proposed timing analysis which formally bounds the WCRT of each task $t \in T$ and the WCTT of each inter-tile message $m \in M$ using the arbitration tuples calculated in the scheduling step. The timing analysis is performed compositionally [20], so that the worst-case timing behavior of each task/message is decomposed into several timing contributions on different resources which are analyzed separately and, then, are combined to obtain globally safe timing guarantees. The obtained task/message latency bounds are then used to bound the worst-case latency (makespan) and throughput of the mapping.

5.1 Worst-Case Response Time

The WCRT of a task denotes the worst-case time interval between its start time and completion time, subject to the worst-case timing interferences that may arise on shared resources due to the presence of interfering requests. In each iteration of its execution, a task reads its required data and input messages from the memory, performs its processing, and writes its output messages into the memory. In general, two sources of interference may occur here: (a) bus interferences when accessing memory and (b) preemption delays on the cores. Therefore, the WCRT of each task $t \in T$ can be bounded as:

$$\mathrm{WCRT}(t, c, q, b) = \mathrm{WCET}(t, c) + \mathrm{MD}(t) \cdot \mathrm{ST}(q, b) + I^{\mathrm{bus}}(t, c, q, b) + I^{\mathrm{core}}(t, c, q, b) \qquad (1)$$

where $\mathrm{WCET}(t, c)$ denotes the worst-case execution time of t on core c in isolation, assuming a zero delay for bus and memory. $\mathrm{MD}(t)$ denotes the memory demand (number of single-word memory accesses) of t, and $\mathrm{ST}(q, b)$ denotes the service time of memory q over bus b, i.e., the turnaround delay of a single-word memory access in absence of bus interferences. $I^{\mathrm{bus}}(t, c, q, b)$ denotes the worst-case delay introduced due to interferences on bus b over which memory q is accessed. $I^{\mathrm{core}}(t, c, q, b)$ denotes the worst-case preemption delay imposed on t's execution on c. Here, $\mathrm{WCET}(t, c)$ and $\mathrm{MD}(t)$ are provided by the application, $\mathrm{ST}(q, b)$ is provided by the architecture, and $I^{\mathrm{bus}}(t, c, q, b)$ and $I^{\mathrm{core}}(t, c, q, b)$ are derived as follows.

5.1.1 Memory Bus Interference

Given that task t executed on core c accesses memory q over bus b, we bound the worst-case memory bus interference $I^{\mathrm{bus}}(t, c, q, b)$ based on the memory demand $\mathrm{MD}(t)$ of t, the service time $\mathrm{ST}(q, b)$ of memory q accessed over bus b, and the arbitration tuple (S_b, W_b^c, P_b^c) for core c on bus b. To this end, we first derive the maximum number of dedicated bus arbitration slots, denoted by $N(t, c, q, b)$, that t may require for performing $\mathrm{MD}(t)$ memory operations: On the one hand, $N(t, c, q, b)$ can never exceed $\mathrm{MD}(t)$, since each bus arbitration slot is necessarily long enough to allow performing at least one single-word memory access, i.e., $S_b \geq \mathrm{ST}(q, b)$. On the other hand, $N(t, c, q, b)$ can never exceed the maximum number of bus arbitration slots that may pass during t's execution when performing its memory accesses in absence of bus interferences. Thus, $N(t, c, q, b)$ can be bounded as:

$$N(t, c, q, b) = \min\left\{\mathrm{MD}(t), \left\lceil \frac{\mathrm{WCET}(t, c) + \mathrm{MD}(t) \cdot \mathrm{ST}(q, b)}{S_b} \right\rceil\right\} \tag{2}$$

In the worst case, the memory accesses of t are distributed such that each of the $N(t, c, q, b)$ required bus slots falls into a separate bus arbitration period. In each bus arbitration period, a worst-case wait time of $(P_b^c - W_b^c \cdot S_b)$ may be imposed on t's execution before t acquires the bus ownership. Therefore, the overall bus interference can be bounded as:

$$I^{\mathrm{bus}}(t, c, q, b) = N(t, c, q, b) \cdot (P_b^c - W_b^c \cdot S_b) \tag{3}$$

In some cases, a memory access may be initiated so late that it cannot be completed before the end of the respective bus arbitration slot. To compensate for this misalignment, the arbitration delay D_b of each bus $b \in B$ is extended by the service time $\mathrm{ST}(q, b)$ of its memory q. This practice (a) allows late memory accesses to be completed before the next bus arbitration slot is assigned and, thereby, (b) eliminates memory interferences due to overlapping memory accesses from different bus masters in consecutive bus slots. Note that, if t accesses multiple memories, the analysis above must be applied for each memory bus b_i over which t performs $\mathrm{MD}_i(t)$ accesses to memory q_i with a service time of $\mathrm{ST}(q_i, b_i)$.

5.1.2 Core Preemption Delay

We bound the worst-case preemption delay of task t on core c based on its arbitration tuple on c, i.e., (S_c, W_c^t, P_c^t), and the arbitration tuple of c on the bus b over which c accesses memory q, i.e., (S_b, W_b^c, P_b^c). In each iteration of its execution, t requires an overall dedicated processor time of $(\mathrm{WCET}(t, c) + \mathrm{MD}(t) \cdot \mathrm{ST}(q) + I^{\mathrm{bus}}(t, c, q, b))$ on core c to complete its processing and perform its memory accesses over the shared bus. In each scheduling period of c, an overall processor time of $(W_c^t \cdot S_c)$ is dedicated to t and, hence, a worst-case periodic wait time of $(P_c^t - W_c^t \cdot S_c)$ is imposed on t's execution due to preemption. Thus, the overall preemption delay imposed on t's execution can be bounded as:

$$I^{\text{core}}(t, c, q, b) = \left\lceil \frac{\text{WCET}(t, c) + \text{MD}(t) \cdot \text{ST}(q) + I^{\text{bus}}(t, c, q, b)}{W_c^t \cdot S_c} \right\rceil \cdot \left(P_c^t - W_c^t \cdot S_c\right) \quad (4)$$

where the first factor derives the maximum number of scheduling periods over which the execution of t may span, and the second factor reflects the worst-case wait time imposed per scheduling period. Note that, also here, a memory access may be initiated so late that it cannot be completed before the end of the scheduling slot. To compensate for this misalignment, the context-switch latency D_c of each core $c \in C$ is extended by the maximum service time among all memories accessible to c to allow an initiated memory access to be completed before the following context switch on c.

5.2 Worst-Case Traversal Time

The WCTT of an inter-tile message m denotes its worst-case transfer latency from the memory in which m is stored on the source tile to the respective target memory on m's destination tile. This transfer is realized in three steps: (I) the TX tx on the source tile reads m from memory q over bus b, decomposes it into flits, and injects the flits into the NoC. (II) the flits are routed over m's NoC route ρ to the destination tile. Once arrived at the destination tile, (III) the RX rx reconstructs m and writes it into m's dedicated space in memory $\hat{q}$ over bus $\hat{b}$. Therefore, the WCTT of message m can be bounded as:

$$\text{WCTT}(m, tx, q, b, \rho, rx, \hat{q}, \hat{b}) = D^{\text{tx}}(m, tx, q, b) + D^{\text{noc}}(m, \rho) + D^{\text{rx}}(m, rx, \hat{q}, \hat{b}) \quad (5)$$

where $D^{\text{tx}}(m, tx, q, b)$, $D^{\text{noc}}(m, \rho)$, and $D^{\text{rx}}(m, rx, \hat{q}, \hat{b})$ respectively denote the worst-case latency of the transfer steps (I)–(III) above which we analyze in the following.

5.2.1 TX/RX Latency

Reading message m from memory q over bus b for injection into the NoC is subject to two sources of interference: (a) TX interference (as multiple outbound messages may be transmitted concurrently from the tile) and (b) bus interference (when TX accesses memory for reading m). We assume that, in each bus arbitration period, bus slots dedicated to TX tx are used for reading one message only, resulting in an equivalent tx slot length of one bus period, i.e., $S_{tx} = P_b^{tx}$. To bound the TX latency $D^{\text{tx}}(m, tx, q, b)$, we first derive the maximum number of bus slots, denoted as $N(m, q, b)$, that can be required in the worst case for reading m from memory q: In each bus arbitration slot, a total of $\lceil S_b/\text{ST}(q, b) \rceil$ consecutive memory accesses can be initiated and completed. Hence, given m's memory demand $\text{MD}(m)$ (number of memory accesses required for reading m), $N(m, q, b)$ is calculated as:

$$N(m, q, b) = \left\lceil \text{MD}(m) \cdot \left\lceil \frac{S_b}{\text{ST}(q, b)} \right\rceil^{-1} \right\rceil \quad (6)$$

Given $N(m, q, b)$, we use Equation (7) to bound the TX latency based on the arbitration tuple of tx on bus b, i.e., $(S_b, W_b^{tx}, P_b^{tx})$, and the arbitration tuple of m on tx, i.e., $(S_{tx}, W_{tx}^m, P_{tx}^m)$ where $S_{tx} = P_b^{tx}$ as discussed above. The first summand in Equation (7) bounds the overall memory service time for the $\text{MD}(m)$ memory accesses required for reading m from memory in absence of bus and TX interferences. The second summand calculates the worst-case bus interference imposed when reading m: Here, the first factor gives the number of bus arbitration periods that may pass before acquiring all $N(m, q, b)$ bus slots required for reading m, and the second factor gives the worst-case wait time imposed per bus period. Finally, the third summand in Equation (7) calculates the worst-case TX interference: Here, the first factor calculates the number of tx arbitration periods involved in the transfer of m, and the second factor gives the worst-case wait time per tx period. We use the same analysis to bound the RX latency, i.e., $D^{\text{rx}}(m, rx, \hat{q}, \hat{b})$ in Equation (5).

$$\begin{aligned} D^{\text{tx}}(m, tx, q, b) = \text{MD}(m) \cdot \text{ST}(q, b) \ + \left\lceil \frac{N(m, q, b)}{W_b^{tx}} \right\rceil \cdot (P_b^{tx} - W_b^{tx} \cdot S_b) \\ + \left\lceil \left\lceil \frac{N(m, q, b)}{W_b^{tx}} \right\rceil \cdot \frac{1}{W_{tx}^m} \right\rceil \cdot (P_{tx}^m - W_{tx}^m \cdot S_{tx}) \end{aligned} \tag{7}$$

5.2.2 NoC Latency

We bound the worst-case NoC routing latency of message m over route ρ based on the arbitration tuple of m on the links in ρ, i.e., $(S_\rho, W_\rho^m, P_\rho^m)$, using Equation (8) adopted from [48]. Here, f_m denotes the number of m's flits which is calculated based on its payload size PLD(m), $|\rho|$ gives the length of route ρ in number of hops, τ^{noc} denotes the length of one NoC clock cycle which also gives the length of one link arbitration slot ($S_\rho = \tau^{\text{noc}}$), and D^{router} gives the latency of a NoC router in clock cycles. The first summand in Equation (8) derives the transfer latency of f_m flits in absence of interferences on ρ. The second summand bounds the worst-case interferences on ρ: Here, the first factor gives the maximum number of link arbitration periods in which m's flits may be stalled due to interfering flows, and the second factor gives the worst-case wait time per link arbitration period, see [48].

$$D^{\text{noc}}(m, \rho) = (f_m - 1 + |\rho| \cdot D^{\text{router}}) \cdot \tau^{\text{noc}} + \left(\left\lceil \frac{f_m}{W_\rho^m} \right\rceil - 1 + |\rho| \right) \cdot (P_\rho^m - W_\rho^m \cdot \tau^{\text{noc}}) \tag{8}$$

5.3 Worst-Case Throughput and Latency

Given the WCRT of each task $t \in T$, in short, WCRT(t), and the WCTT of each inter-tile message $m \in M$, in short, WCTT(m), the worst-case throughput of the mapping is calculated using Equation (9). Likewise, the worst-case application latency (makespan) is calculated using Equation (10) where Π denotes the set of all end-to-end paths in the application graph.

$$\mathcal{TH} = \max \left\{ \max_{t \in T} \{\text{WCRT}(t)\}, \max_{m \in M} \{\text{WCTT}(m)\} \right\}^{-1} \tag{9}$$

$$\mathcal{L} = \max_{\pi \in \Pi} \left\{ \sum\nolimits_{t \in (T \cap \pi)} \text{WCRT}(t) + \sum\nolimits_{m \in (M \cap \pi)} \text{WCTT}(m) \right\} \tag{10}$$

6 Experimental Results

This section presents the results of a series of experiments for a variety of applications and architectures to compare the performance of the proposed isolation-aware approach with existing fixed-isolation-scheme approaches w.r.t. the quality of delivered mappings.

6.1 Experiment Setup

Applications and Architectures. We use four real-time applications from the domains of networking (7 tasks, 9 messages), consumer (11 tasks, 12 messages), telecommunication (14 tasks, 20 messages), and automotive (18 tasks, 21 messages), provided by the Embedded

System Synthesis Benchmarks Suite (E3S) [11]. For target platform, we use three heterogeneous many-core architectures with 4×4, 5×5, and 6×6 tiles, respectively. Each architecture is composed of three tile types. Each tile comprises four homogeneous cores, a shared memory, a NA with separate TX and RX units, and a shared memory bus. Every shared resource has a WRR arbitration policy configured as follows: For each core, an arbitration capacity of $K = 10$, a slot length of $S = 50\,us$, and a context switch overhead of $D = 10\,us$ is considered. On each bus, each bus master (TX, RX, and four cores) has an arbitration weight of $W = 1$, resulting in a bus arbitration capacity of $K = 6$. The length of each bus slot S is set equal to the memory service time of 7 clock cycles. For NoC links, we consider an arbitration capacity of $K = 10$ and a slot length $S = \tau^{\text{noc}} = 10\,ns$. For each TX/RX, an arbitration capacity of $K = 10$ and a slot length equal to the bus arbitration period is considered, cf. Section 5.2.1.

Design Objectives. We use three design objectives which are commonly considered in the DSE of predictable and composable many-core systems: (I) *worst-case latency* derived using the proposed timing analysis, (II) *resource usage* calculated (in number of cores) as the sum of time slots reserved on each core divided by core arbitration capacity $K = 10$, and (III) *energy consumption*, calculated based on the processor power parameters provided by [11] for each investigated benchmark application and the NoC/bus energy model from [52], assuming a link length of $2\,mm$ and a bus length of $5\,mm$.

Design Space Exploration. To perform the DSE, we use the OpenDSE framework [38] and the NSGA-II [10] multi-objective evolutionary algorithm provided by the optimization framework Opt4J [30]. Each run of the DSE features 4,000 iterations with 25 mappings generated per iteration and a population size of 100 mappings. The results reported in this section are an average over 20 runs of the DSE for each application on each architecture.

Investigated Approaches. We compare the proposed isolation-aware DSE approach with the three major fixed-isolation-scheme DSE approaches, namely, core sharing, core reservation, and tile reservation. For each approach, the DSE delivers a set of mappings with Pareto-optimal trade-offs in the space of the three design objectives above. We compare the DSE approaches in terms of the quality of mappings they deliver.

Quality Metric. To compare the quality of mappings obtained by the investigated DSE approaches, we use the well-established *ϵ-dominance* metric [28] from the domain of multi-objective optimization, defined as follows: Let $F \subseteq {\mathbb{R}^+}^N$ represent a set of Pareto-optimal mappings $f \in F$ obtained by an optimization approach in the space of design objectives $o_1, ..., o_N$ to be minimized. Let $S \subseteq {\mathbb{R}^+}^N$ represent a reference set containing the true Pareto-optimal mappings for the optimization problem in question. To assess the quality of mappings in F w.r.t. S, ϵ-dominance provides a unary indicator $\epsilon_F \in [0, 1)$ calculated as:

$$\epsilon_F = \min\{0 \le \epsilon < 1 \mid \forall s \in S : \exists f \in F \text{ s.t. } (1 - \epsilon) \cdot f_{o_n} \le s_{o_n}\ \forall n = 1, ..., N\} \tag{11}$$

where f_{o_n} and s_{o_n} denote the quality of mappings $f \in F$ and $s \in S$, respectively, w.r.t. design objective o_n. A *smaller* value for ϵ_F indicates a smaller distance between the mappings in F and those in the reference set S w.r.t. the design objectives which, in turn, denotes a *higher quality* of mappings in F. For our following experiments, the reference set S is constructed by collecting the mappings obtained by the four approaches under analysis in one set.

6.2 Result Discussion

Figure 8 illustrates, for each benchmark application on each many-core architecture, the ϵ-dominance indicator for the proposed approach and the three fixed-isolation-scheme approaches across their exploration run time. The plots in each row correspond to the same application, and the plots in each column correspond to the same architecture. In general, the quality of mappings collected by each approach improves as the exploration progresses, resulting in a decrease in the ϵ indicator of each approach throughout the course of its optimization iterations. Also, the exploration time of each approaches grows with the application size (compare the plots in one column) and architecture size (compare the plots in one row). The results illustrated in Figure 8 uniformly verify that (a) the proposed approach always outperforms the approaches with a fixed isolation scheme, irrespective of the choice of application and architecture, which is indicated by its lower ϵ-dominance indicator at the end of its exploration. Moreover, (b) the proposed approach exhibits an exploration run time within the same range as the other approaches and, hence, does not impact the scalability of the optimization approach adversely. Among the other approaches, core reservation generally performs better than core sharing and tile reservation for most of cases. Core sharing exhibits the worst quality of solutions for all applications and architectures. Considering all 12 combinations of applications and architectures, the proposed isolation-aware approach achieves an ϵ-dominance improvement of up to 67% (compared with the core-sharing approach) with an average improvement of 26% over the three fixed-isolation-scheme approaches.

To reason about these quality differences, we investigate the distribution of the mappings delivered by each DSE approach in the 3D objective space. Figure 9 illustrates this using two 2D projections of the objective space for two exemplary applications on the 6×6 architecture. Similar distributions are obtained for other applications and architectures. For each application (column), the x-axes in both projections denote resource usage while the y-axes denote worst-case latency (top) and energy consumption (bottom). As illustrated, solutions delivered by the core-sharing approach exhibit low resource usage, as this approach allocates a *minimal* resource budget, just enough to meet the deadlines of tasks/messages. This, however, implies a high worst-case inter-application interference and, thus, considerably high worst-case latency and energy consumption. The tile-reservation approach, on the other hand, allocates compute tiles exclusively (thus, higher resource usage) which eliminates all on-tile inter-application interferences (thus, lower latency and energy). When considering all objectives together, both of these approaches need a large scaling factor ϵ to compensate for their inferior objective values, explaining their poor (high) ϵ-dominance indices in Figure 8. A compromise between these approaches is achieved by core reservation which restricts the exclusive allocation of resources to cores, leading to a moderate trade-off between resource usage, latency, and energy consumption and, thus, a better (lower) ϵ-dominance index.

The fixed isolation scheme of the approaches above excludes large parts of the actual solution space and, thus, restricts their coverage of the objective space to a considerably smaller sub-region. Contrary to them, the proposed approach covers and extends beyond the solution space of these approaches as it explores the isolation schemes in combination within each mapping, also reflected by the wide spread of its solutions in Figure 9. As a result, it can find solutions of higher quality and, thus, outperforms the other approaches as confirmed by its lower ϵ-dominance index in Figure 8 for all investigated applications and architectures.

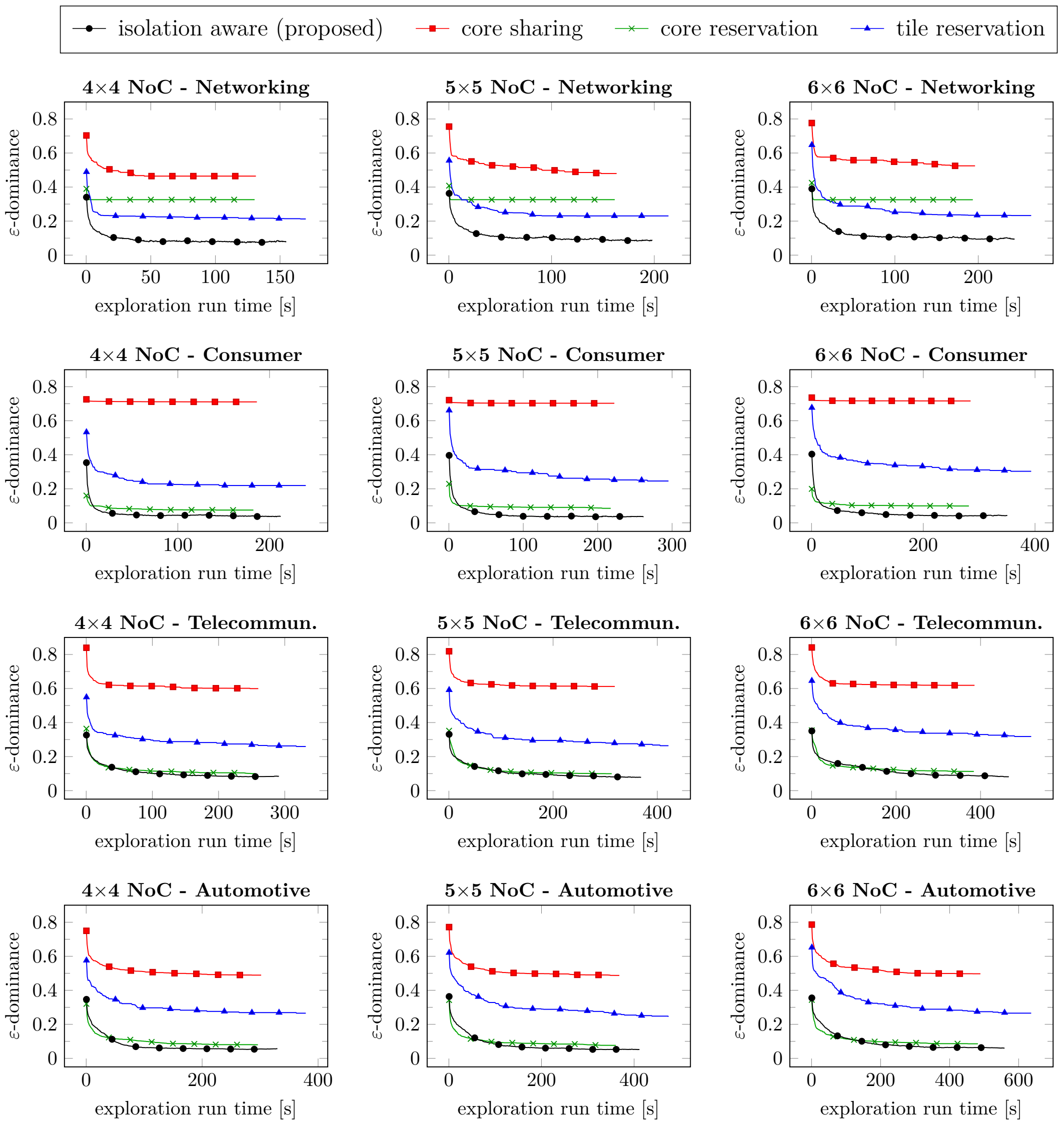


Figure 8 ϵ-dominance of the investigated DSE approaches versus their exploration run time. Plots in each column (row) correspond to the same many-core architecture (application). The proposed isolation-aware approach outperforms existing fixed-isolation-scheme approaches for all applications and architectures, denoted by its lower ϵ-dominance indicator.

7 Conclusion

Applications in composable many-core systems are typically developed with the assumption of a fixed inter-application isolation scheme which restricts the resource allocation policy of the applications and, therefore, their quality trade-off w.r.t. resource usage and worst-case timing. To lift this restriction, we have proposed (a) an *isolation-aware Design Space Exploration (DSE)* which explores the isolation schemes per allocated core/tile within each mapping and (b) an *isolation-aware timing analysis* to formally bound the worst-case timing properties of each explored mapping. For a variety of hard real-time applications and many-core architectures, we have experimentally demonstrated the advantage of the proposed approach over existing fixed-isolation-scheme approaches w.r.t. the quality of the delivered solutions in terms of resource usage, worst-case latency, and energy consumption.

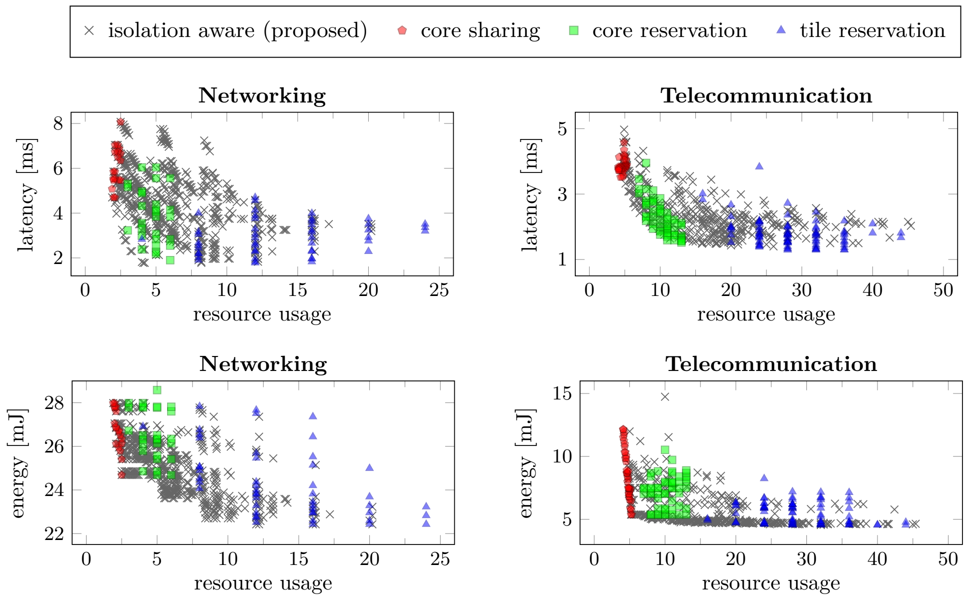


Figure 9 2D projections of the 3D objective space showing the spread of solutions delivered by each DSE approach for two exemplary applications (columns) on the 6×6 many-core architecture.

References

1 Benny Akesson, Anca Molnos, Andreas Hansson, Jude Ambrose Angelo, and Kees Goossens. Composability and predictability for independent application development, verification, and execution. In *Multiprocessor System-on-Chip*, pages 25–56. Springer, 2011.

2 Sebastian Altmeyer, Robert I Davis, Leandro Indrusiak, Claire Maiza, Vincent Nelis, and Jan Reineke. A generic and compositional framework for multicore response time analysis. In *Proceedings of the Conference on Real Time Networks and Systems (RTNS)*, pages 129–138. ACM, 2015.

3 Tobias Blickle, Jürgen Teich, and Lothar Thiele. System-level synthesis using evolutionary algorithms. *Design Automation for Embedded Systems*, 3(1):23–58, 1998.

4 Tilera Corporation. Tile Processor Architecture Overview for the TILE-Gx Series, 2012.

5 William J Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed systems*, 3(2):194–205, 1992.

6 Dakshina Dasari, Vincent Nelis, and Benny Akesson. A framework for memory contention analysis in multi-core platforms. *Real-Time Systems*, 52(3):272–322, 2016.

7 Lawrence Davis. *Handbook of genetic algorithms.* VNR computer library. Van Nostrand Reinhold, 1991.

8 Robert I Davis, Sebastian Altmeyer, Leandro S Indrusiak, Claire Maiza, Vincent Nelis, and Jan Reineke. An extensible framework for multicore response time analysis. *Real-Time Systems*, pages 1–55, 2017.

9 Benoît Dupont de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benoit Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frédéric Riss, et al. A clustered manycore processor architecture for embedded and accelerated applications. In *Proceedings of High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2013.

10 Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

11 Robert Dick. Embedded system synthesis benchmarks suite (E3S), 2010. URL: `http://ziyang.eecs.umich.edu/~dickrp/e3sdd/`.

12 Carlos M. Fonseca and Peter J. Fleming. Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion and Generalization. In *Proceedings of the International Conference on Genetic Algorithms*, pages 416–423. Morgan Kaufmann Publishers Inc., 1993.

13 Carlos M Fonseca and Peter J Fleming. An overview of evolutionary algorithms in multiobjective optimization. *Evolutionary computation*, 3(1):1–16, 1995.

14 Michael R Garey and David S Johnson. A Guide to the Theory of NP-Completeness. *Computers and Intractability*, pages 37–79, 1990.

15 Georgia Giannopoulou, Kai Lampka, Nikolay Stoimenov, and Lothar Thiele. Timed model checking with abstractions: Towards worst-case response time analysis in resource-sharing manycore systems. In *Proceedings of the International Conference on Embedded Software*, pages 63–72. ACM, 2012.

16 Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, and Lothar Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *Proceedings of the International Conference on Embedded Software*, page 17. ACM, 2013.

17 Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, Lothar Thiele, and Benoît Dupont de Dinechin. Mixed-criticality scheduling on cluster-based manycores with shared communication and storage resources. *Real-Time Systems*, 52(4):399–449, 2016.

18 Michael Glaß, Jürgen Teich, Martin Lukasiewycz, and Felix Reimann. Hybrid optimization techniques for system-level design space exploration. In *Handbook of Hardware/Software Codesign*, volume 1, pages 217–246. Springer, 2017.

19 Kees Goossens, Arnaldo Azevedo, Karthik Chandrasekar, Manil Dev Gomony, Sven Goossens, Martijn Koedam, Yonghui Li, Davit Mirzoyan, Anca Molnos, Ashkan Beyranvand Nejad, et al. Virtual execution platforms for mixed-time-criticality systems: the CompSOC architecture and design flow. *ACM SIGBED Review*, 10(3):23–34, 2013.

20 Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards compositionality in execution time analysis: definition and challenges. *ACM SIGBED Review*, 12(1):28–36, 2015.

21 Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1):2, 2009.

22 Jan Heisswolf, Ralf König, Martin Kupper, and Jürgen Becker. Providing multiple hard latency and throughput guarantees for packet switching networks on chip. *Computers & Electrical Engineering*, 39(8):2603–2622, 2013.

23 Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 108–109. IEEE, 2010.

24 Timon Kelter, Tim Harde, Peter Marwedel, and Heiko Falk. Evaluation of resource arbitration methods for multi-core real-time systems. In *Proceedings of the workshop on Worst-Case Execution Time Analysis (WCET)*. Schloss Dagstuhl-Leibniz-Zentrum fr̈ Informatik, 2013.

25 James Kennedy. Particle swarm optimization. *Encyclopedia of Machine Learning*, pages 760–766, 2010.

26 Pham Nam Khanh, Amit Kumar Singh, Akash Kumar, and Khin Mi Mi Aung. Incorporating energy and throughput awareness in design space exploration and run-time mapping for heterogeneous MPSoCs. In *Proceedings of the Euromicro Conference on Digital System Design (DSD)*, pages 513–521. IEEE, 2013.

27 Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

28 Marco Laumanns, Lothar Thiele, Kalyanmoy Deb, and Eckart Zitzler. Combining convergence and diversity in evolutionary multiobjective optimization. *Evolutionary Computation*, 10(3):263–282, 2002.

29 Martin Lukasiewycz, Michael Glaß, Christian Haubelt, and Jürgen Teich. SAT-decoding in evolutionary algorithms for discrete constrained optimization problems. In *Proceedings of the IEEE Congress onEvolutionary Computation*, pages 935–942. IEEE, 2007.

30 Martin Lukasiewycz, Michael Glaß, Felix Reimann, and Jürgen Teich. Opt4J: a modular framework for meta-heuristic optimization. In *Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1723–1730. ACM, 2011.

31 Martin Lukasiewycz, Shanker Shreejith, and Suhaib A Fahmy. System simulation and optimization using reconfigurable hardware. In *International Symposium on Integrated Circuits (ISIC)*, pages 468–471, 2014.

32 Guilherme Madalozzo, Liana Duenha, Rodolfo Azevedo, and Fernando G Moraes. Scalability evaluation in many-core systems due to the memory organization. In *Proceedings of the International Conference on Electronics, Circuits and Systems (ICECS)*, pages 396–399. IEEE, 2016.

33 Giovanni Mariani, Prabhat Avasare, Geert Vanmeerbeeck, Chantal Ykman-Couvreur, Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. An industrial design space exploration framework for supporting run-time resource management on multi-core systems. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 196–201, 2010.

34 Tulika Mitra, Jürgen Teich, and Lothar Thiele. Time-Critical Systems Design: A Survey. *IEEE Design & Test*, 35(2):8–26, 2018.

35 Lionel M Ni and Philip K McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26(2):62–76, 1993.

36 Roberta Piscitelli and Andy D Pimentel. Design space pruning through hybrid analysis in system-level design space exploration. In *Proceedings of the Conference on Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 781–786. IEEE, 2012.

37 Behnaz Pourmohseni, Stefan Wildermann, Michael Glaß, and Jürgen Teich. Hard real-time application mapping reconfiguration for NoC-based many-core systems. *Real-Time Systems*, pages 1–37, 2019.

38 Felix Reimann, Martin Lukasiewycz, Michael Glaß, and Fedor Smirnov. OpenDSE – open design space exploration framework, 2018. URL: `http://opendse.sourceforge.net/`.

39 Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.

40 Hamza Rihani, Matthieu Moy, Claire Maiza, Robert I Davis, and Sebastian Altmeyer. Response Time Analysis of Synchronous Data Flow Programs on a Many-Core Processor. In *Proceedings of the conference on Real-Time Networks and Systems (RTNS)*, pages 67–76. ACM, 2016.

41 Benjamin Rouxel, Steven Derrien, and Isabelle Puaut. Tightening contention delays while scheduling parallel applications on multi-core architectures. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):164, 2017.

42 Pradip Kumar Sahu, Tapan Shah, Kanchan Manna, and Santanu Chattopadhyay. Application mapping onto mesh-based network-on-chip using discrete particle swarm optimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(2):300–312, 2014.

43 Z. Shi and A. Burns. Real-Time Communication Analysis for On-Chip Networks with Wormhole Switching. In *International Symposium on Networks-on-Chip (NOCS)*, pages 161–170, 2008.

44 Amit Kumar Singh, Akash Kumar, and Thambipillai Srikanthan. Accelerating throughput-aware runtime mapping for heterogeneous MPSoCs. *ACM Transaction on Design Automation of Electronic Systems (TODAES)*, 18(1):9:1–9:29, 2013.

45 Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. Mapping on multi/many-core systems: survey of current and emerging trends. In *Proceedings of the Design Automation Conference (DAC)*, pages 1–10, 2013.

46 Stefanos Skalistis and Alena Simalatsar. Worst-case execution time analysis for many-core architectures with NoC. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 211–227. Springer, 2016.

47 Stefanos Skalistis and Alena Simalatsar. Near-optimal deployment of dataflow applications on many-core platforms with real-time guarantees. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 752–757. IEEE, 2017.

48 Andreas Weichslgartner, Deepak Gangadharan, Stefan Wildermann, Michael Glaß, and Jürgen Teich. DAARM: Design-time application analysis and run-time mapping for predictable execution in many-core systems. In *Proceedings of the International Conference on Hardware/-Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, 2014.

49 Andreas Weichslgartner, Stefan Wildermann, Deepak Gangadharan, Michael Glaß, and Jürgen Teich. A design-time/run-time application mapping methodology for predictable execution time in MPSoCs. *ACM Transactions on Embedded Computing Systems (TECS)*, 2018.

50 Andreas Weichslgartner, Stefan Wildermann, Michael Glaß, and Jürgen Teich. *Invasive Computing for mapping parallel programs to many-core architectures*. Springer, 2018.

51 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem-overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

52 Pascal T Wolkotte, Gerard JM Smit, Nikolay Kavaldjiev, Jens E Becker, and Jürgen Becker. Energy model of networks-on-chip and a bus. In *Proceedings of the International Symposium on System-on-Chip (SoC)*, pages 82–85, 2005.

53 Chantal Ykman-Couvreur, Prabhat Avasare, Giovanni Mariani, Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. Linking run-time resource management of embedded multi-core platforms with automated design-time exploration. *IET Computers and Digital Techniques*, 5(2):123–135, 2011.

54 Jia Zhan, Nikolay Stoimenov, Jin Ouyang, Lothar Thiele, Vijaykrishnan Narayanan, and Yuan Xie. Designing energy-efficient NoC for real-time embedded systems through slack optimization. In *Proceedings of the Design Automation Conference (DAC)*, pages 1–6. IEEE, 2013.

GEDF Tardiness: Open Problems Involving Uniform Multiprocessors and Affinity Masks Resolved

Stephen Tang
Department of Computer Science, University of North Carolina at Chapel Hill, NC, USA
sytang@cs.unc.edu

Sergey Voronov
Department of Computer Science, University of North Carolina at Chapel Hill, NC, USA
rdkl@cs.unc.edu

James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill, NC, USA
anderson@cs.unc.edu

Abstract

Prior work has shown that the global earliest-deadline-first (GEDF) scheduler is *soft real-time (SRT)-optimal* for sporadic task systems in a variety of contexts, meaning that bounded deadline tardiness can be guaranteed under it for any task system that does not cause platform overutilization. However, one particularly compelling context has remained elusive: multiprocessor platforms in which tasks have *affinity masks* that determine the processors where they may execute. Actual GEDF implementations, such as the `SCHED_DEADLINE` class in Linux, have dealt with this unresolved question by foregoing SRT guarantees once affinity masks are set. This unresolved question, as it pertains to `SCHED_DEADLINE`, was included by Peter Zijlstra in a list of important open problems affecting Linux in his keynote talk at ECRTS 2017. In this paper, this question is resolved along with another open problem that at first blush seems unrelated but actually is. Specifically, both problems are closed by establishing two results. First, a proof strategy used previously to establish GEDF tardiness bounds that are exponential in size on heterogeneous uniform multiprocessors is generalized to show that polynomial bounds exist on a wider class of platforms. Second, both uniform multiprocessors and identical multiprocessors with affinities are shown to be within this class. These results yield the first polynomial GEDF tardiness bounds for the uniform case and the first such bounds of any kind for the identical-with-affinities case.

2012 ACM Subject Classification Software and its engineering → Real-time schedulability

Keywords and phrases scheduling theory, multicore, processor affinity masks, GEDF, uniform multiprocessors

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.13

Related Version A full version of the paper is available at `http://www.cs.unc.edu/~anderson/papers.html`.

Funding Work supported by NSF grants CNS 1409175, CNS 1563845, CNS 1717589, and CPS 1837337, ARO grant W911NF-17-1-0294, and funding from General Motors.

1 Introduction

The global earliest-deadline-first (GEDF) scheduler has received considerable prior attention. One attractive property of GEDF is that its use ensures guaranteed bounded deadline tardiness on certain multiprocessor platform types for any sporadic task system that does not cause platform overutilization [5, 9]. In this sense, GEDF is considered an *optimal soft* *real-time (SRT)* scheduler. This SRT-optimality is significant enough to warrant mention in the documentation of `SCHED_DEADLINE` [2], Linux's implementation of GEDF. A practical use by `SCHED_DEADLINE` of this optimality is the ability to perform online admission control.

A major caveat of GEDF's SRT-optimality is that it was originally proven only for processors with identical speeds [5]. Processor models that break this assumption, called *heterogeneous multiprocessors*, tend to fundamentally break the proof techniques applied to the identical model. Only one prior work [9] has succeeded in extending the SRT-optimality of GEDF to a multiprocessor model with heterogeneity, namely the *uniform model*, which allows speed differences among processors. This work required years of thought and several new proof techniques but only yielded tardiness bounds that are exponential in size.

Affinity masks. Heterogeneity is also introduced by the usage of *affinity masks.* A task's affinity mask (typically a bit vector) indicates the processors upon which it may execute. Affinity masks are useful for preventing excessive task migrations and can be used to take better advantage of cache hierarchies. Also, global, clustered, and partitioned scheduling can all be expressed using affinity masks.

Affinity masks introduce heterogeneity by removing the symmetry among processors. An important property used to prove the SRT-optimality of GEDF without affinities is that the existence of an idle processor is a sufficient condition for a pending task to begin executing. With this property, showing that a task of interest makes progress merely requires showing that *some* processor becomes idle in a bounded amount of time. With affinity masks, this proof strategy does not work. In particular, this strategy's use would require showing that a processor *allowed by the considered task's affinity mask* becomes idle, and doing so over the entire space of arbitrary affinity masks leads to a case explosion.

This difficulty has left unresolved the question of whether GEDF retains its SRT-optimality with the introduction of affinity masks. This unresolved question has rendered systems that support both GEDF and affinity masks incomplete. For example, in the case of `SCHED_DEADLINE`, either admission control is disabled or affinity masks are forbidden altogether. This specific gap between theory and implementation was mentioned by Peter Zijlstra [10] in a keynote talk at ECRTS 2017 and by Luca Abeni [1] at RTSS 2017.

Contributions. In this paper, we close the open problem of whether GEDF retains its SRT-optimality on identical multiprocessors with affinities by showing that it does, provided the GEDF implementation maintains a certain task-migration property. We also establish the first ever polynomial tardiness bounds for GEDF on uniform multiprocessors. While these two platform models (uniform and identical with affinities) may seem unrelated to each other, we shall see that they are.

These results hinge on three proof innovations. First, we introduce a layer of abstraction between the processor models we consider and the tardiness analysis. This abstraction layer is a property we call "HP-LAG," which we prove is satisfied by GEDF on the models we consider for all sporadic task systems that do not cause platform overutilization.[1] This strategy allows us to reason about tardiness without regard for specifics concerning the underlying platform, and as a result, we are able to avoid the aforementioned case explosion.

Second, for the statement made in the prior paragraph concerning GEDF to be valid, the definition of GEDF itself must be tailored for the specific platform type under consideration. Prior work [9] has shown how to do so for the case of a uniform platform, and we show here

[1] The concept of "overutilization" is more nuanced on uniform platforms and platforms with affinities than on identical platforms with no affinities.

how to do so for an identical platform with affinities by specifying rules that GEDF must adhere to in this case. These rules specify when and how tasks must be migrated as scheduling decisions are made. Now that these rules are known, actual implementations of GEDF on systems that support affinity masks should be designed to uphold them. Unfortunately, as we discuss in App. C, available online [7], the current `SCHED_DEADLINE` implementation does not do so, so a refinement of it is needed if affinity masks are to be used alongside admission control. In App. A, also available online [7], we present an algorithm that implements our GEDF scheduling rules for affinity masks with lower time complexity than `SCHED_DEADLINE`, given some preprocessing.

The last innovation is our improved tardiness analysis with less pessimism than prior work, particularly with respect to uniform multiprocessor platforms. This required maintaining an exponential number of invariants relative to prior work.

Organization. In the rest of this paper, we cover needed background and define the important concept of Lag, which is widely used in tardiness analysis (Sec. 2), list some general Lag properties that are predicated upon tasks being periodic and executing for their worst-case requirements (Sec. 3), define HP-LAG and present our tardiness analysis for HP-LAG-compliant schedulers using said properties (Sec. 4), show that GEDF (if implemented appropriately) is HP-LAG-compliant on both uniform multiprocessors (Sec. 5) and identical multiprocessors with affinities (Sec. 6), show that our results extend to sporadic tasks that may execute for less than their worst-case requirements (Sec. 7), explain why these results cannot be easily extended to more general models (Sec. 8), and conclude (Sec. 9).

2 Background

We consider the problem of scheduling n implicit-deadline sporadic tasks $\tau = \{\tau_1, \ldots, \tau_n\}$ on a multiprocessor $\pi = \{\pi_1, \ldots, \pi_m\}$ with m processors. We consider time to be continuous. Each task τ_i releases a sequence of jobs with a minimum separation of T_i time units between releases; T_i is called τ_i's *period*. The j^{th} released job of τ_i is denoted $J_{i,j}$, and its release time is denoted $r_{i,j}$. When this separation between the jobs of each task τ_i is exactly T_i, the system is called *periodic*. With the assumption of implicit deadlines, the *deadline* of a job of τ_i is exactly T_i time units after its release; the deadline of job $J_{i,j}$ is denoted $d_{i,j} = r_{i,j} + T_i$. The amount of work that is needed to complete a job of τ_i is bounded by τ_i's *worst-case execution requirement (WCER)* C_i, the largest of which among the tasks in τ we denote as C_{max}. Note that C_i is typically called the worst-case execution *time* in the literature. This is because much of the literature assumes that processors complete one unit of execution in one unit of time. This assumption does not hold for some of the hardware platform models we consider in this work (see Sec. 2.1). τ_i's *utilization* is defined as $u_i = C_i/T_i$. We let u_{min} be the smallest utilization among the tasks in τ. For any $\tau' \subseteq \tau$, we let $U_{\tau'}$ denote $\sum_{\tau_i \in \tau'} u_i$. A job is *pending* at time t if it has been released but has not completed. Likewise, a task is *pending* at t if any of its jobs are pending at t. A job is *ready* at t if it is the earliest released pending job from its task.

▶ **Definition 1.** *If task τ_i is pending at time t, then we define its release time at t, denoted $r_i(t)$, and its deadline at t, denoted $d_i(t)$, as the release time and deadline, respectively, of its ready job at time t. If τ_i is not pending at t, then we define $d_i(t) = \infty$.*

If a job has a deadline at time t_d and completes at time t_c, then its *tardiness* is defined as $\max(0, t_c - t_d)$. The tardiness of a task is the supremum of the tardiness of any of its jobs. If this value is finite, then we say that the task has *bounded tardiness.*

In the real-time literature, both hard real-time (HRT) and soft real-time (SRT) systems are considered. While SRT can be defined in different ways, we adopt the following definitions. A task set τ is HRT-schedulable (resp., SRT-schedulable) under a given scheduling algorithm if each task in τ has 0 (resp., bounded) tardiness in any schedule for τ generated by that algorithm. A task set τ is HRT-feasible (resp., SRT-feasible) if it is HRT-schedulable (resp., SRT-schedulable) under some scheduling algorithm. A scheduler is HRT-optimal (resp., SRT-optimal) if it can schedule any HRT-feasible (resp., SRT-feasible) system.

2.1 Multiprocessor Platform Models

There exist several multiprocessor platform models in the literature that differ by how execution speeds are allowed to vary. We formalize execution speeds as follows. Hereafter, we let $A(\mathcal{S}, \tau_i, t, t')$ denote the cumulative execution allocated to jobs of task τ_i in the schedule $\mathcal{S}$ in the time interval (t, t').

▶ **Definition 2.** *A time interval (t, t') is a continuous scheduling interval if the assignment of tasks to processors at t is maintained throughout (t, t').*

▶ **Definition 3.** *Suppose task τ_i executes on processor π_j over the continuous scheduling interval (t, t') in schedule $\mathcal{S}$. If the speed of τ_i on π_j is s, then $A(\mathcal{S}, \tau_i, t, t') = s(t' - t)$.*

In order of increasing generality, the platform models of relevance to us are as follows.

1. **Identical.** All tasks execute with speed 1.0 on all processors.
2. **Uniform.** Speeds may vary by processor, but not by task. Any task on processor π_i executes with speed s_i, which may differ from 1.0.
3. **Unrelated.** Speeds may vary by processor and by task. Task τ_i executes on processor π_j with speed $s_{i,j}$.

In addition to these models, the migration scheme can be one of two types.

1. **Global.** A task can be scheduled on any processor.
2. **Affinity.** A task can only be scheduled on a specific set of processors as defined by its *affinity mask*. We let $\alpha_i \subseteq \pi$ denote the set of processors allowed by τ_i's affinity mask.

Note that global, clustered, and partitioned scheduling can all be defined using affinity masks. We separate global scheduling as a separate case because some later proofs focus on it exclusively. From this point on, global scheduling is assumed unless the -Aff suffix is appended. For example, Identical and Identical-Aff refer to an identical multiprocessor under global and affinity scheduling, respectively. Note that Unrelated generalizes not only Identical and Uniform, but also affinity scheduling, as $\pi_j \notin \alpha_i$ can be represented by letting $s_{i,j} = 0$. Hence, all combinations of platform and migration scheme are generalized by Unrelated.

GEDF has been proven to be SRT-optimal under Identical, but no prior work has generalized this to Identical-Aff. GEDF's SRT-optimality has been generalized to Uniform [9], but with exponential tardiness bounds. In this work, we establish the SRT-optimality of GEDF with polynomial tardiness under both Uniform and Identical-Aff.

The SRT-optimality of GEDF under Uniform-Aff and Unrelated are difficult or impossible to prove using the techniques of this work. We describe why in Sec. 8.

We will later show that the typical GEDF scheduling rules under Identical may be insufficient to achieve SRT-optimality under the more general models. As such, we will later define extended GEDF scheduling rules for Identical-Aff and Uniform, respectively (our rules for Uniform are actually from [9]).

The standard gedf scheduler under Identical is:

IG-GEDF:	At every time instant, if more than m tasks are pending, then the m pending tasks with the earliest deadlines are scheduled; otherwise, all pending tasks are scheduled.

The prefix "IG" denotes that this rule applies under Identical with global scheduling. We will keep this notation when extending GEDF, denoting the extended GEDF rules for Identical-Aff and Uniform as IA-GEDF and UG-GEDF, respectively. Under all GEDF variants we consider, we assume deadline ties are broken in some arbitrary but consistent way (e.g., by task index). For any time instant t, for tasks τ_i and τ_j, we let $d_i(t) \prec d_j(t)$ denote that either $d_i(t) < d_j(t)$ holds or $d_i(t) = d_j(t)$ holds with the tie broken in τ_i's favor. As we shall see, IA-GEDF and UG-GEDF reduce to IG-GEDF when the underlying processor model is Identical (note that both Uniform and Identical-Aff generalize Identical).

2.2 Lag

As in [5], our analysis is based on the concept of Lag. Lag compares the execution of a task in a "real" schedule $\mathcal{R}$ to its allocation in an "ideal" schedule $\mathcal{I}$.

▶ **Definition 4.** *A non-fluid schedule is a schedule such that at any time instant t, there exists some $\delta > 0$ such that $(t, t + \delta)$ is a continuous scheduling interval.*

Implementable schedulers are non-fluid.

▶ **Definition 5.** *We let $\mathcal{R}$ denote a non-fluid schedule produced under a considered scheduling algorithm and multiprocessor platform.*

▶ **Definition 6.** *We let $\mathcal{I}$ denote a schedule that executes task τ_i on processor π_i of speed u_i.*

Notice that $\mathcal{I}$ is defined with respect to a Uniform multiprocessor consisting of n processors $\pi_1, ..., \pi_n$ with speeds $u_1, ..., u_n$, respectively, and not the actual platform π.

Under the implicit-deadline sporadic task model, every job executes in $\mathcal{I}$ from its release until its completion without interference from other jobs or tasks (different tasks run on different processors). If a job's execution requirement is smaller than the WCER of its task, then the job completes in the ideal schedule before its deadline; otherwise, it completes exactly at its deadline. Thus, in $\mathcal{I}$, at most one job from every task is ever scheduled.

We are now ready to formally define Lag.

▶ **Definition 7.** *For a single task τ_i, $\mathsf{Lag}_i(t) = A(\mathcal{I}, \tau_i, 0, t) - A(\mathcal{R}, \tau_i, 0, t)$. For the subset $\tau' \subseteq \tau$, $\mathsf{LAG}(\tau',\ t) = \sum_{\tau_i \in \tau'} \mathsf{Lag}_i(t)$.*

3 General Lag Properties

In [9], Yang and Anderson showed how to generalize IG-GEDF to obtain a variant, which we denote as UG-GEDF, that is SRT-optimal under Uniform. Yang and Anderson's proof relied on several properties of Lag that we make use of in this work. We repeat these properties and proofs verbatim from [9], with minor wording changes. However, unlike [9], where these properties were considered in the context of Uniform, we consider them in the context of Unrelated. Because Unrelated generalizes all the models in Sec. 2.1, these Lag properties apply to all the models considered in this work.

Lemmas 10–12 rely on the following assumptions, which we henceforth assume until stated otherwise.

Every task is periodic. (P)

Every job $J_{i,j}$ has an execution requirement equal to C_i (the worst case for τ_i). (W)

So as to leave no doubt that the properties considered in this section hold under Unrelated, we begin by listing all of the model-related concepts the proofs below will use:

- the task-system parameters C_i, T_i, u_i, $r_i(t)$, and $d_i(t)$;
- Lag and LAG, as defined in Def. 7;
- the fact that $\mathcal{I}$ continuously executes task τ_i with speed u_i, which follows from (P) and (W), hence the need for these assumptions

[9] showed that removing (P) and (W) cannot cause greater tardiness in UG-GEDF under Uniform. We will show the same for IA-GEDF under Identical-Aff later in Sec. 7.

▶ **Lemma 8** (Property 1 of [9]). *$\forall \tau' \subseteq \tau$: $\mathsf{LAG}(\tau',\ t)$ is a continuous function of t.*

Proof. By definition, $A(\mathcal{S}, \tau_i, t, t')$ is a continuous function in t and t'. By Def. 7, $\mathsf{LAG}(\tau',\ t) = \sum_{\tau_i \in \tau'} \mathsf{Lag}_{\tau_i}(t) = \sum_{\tau_i \in \tau'} (A(\mathcal{I}, \tau_i, 0, t) - A(\mathcal{R}, \tau_i, 0, t))$. Because $\mathsf{LAG}(\tau',\ t)$ is a finite sum of continuous functions in t, it is continuous in t. ◀

▶ **Lemma 9** (Lemma 1 of [9]). *$\mathsf{Lag}_i(t) > 0 \Rightarrow \tau_i$ has a pending job at t.*

Proof. We prove the lemma by contradiction. Suppose that $\mathsf{Lag}_i(t) > 0$ holds but τ_i is not pending at t in $\mathcal{R}$. Then, all jobs of τ_i released before or at t have completed by t. Let W denote the total execution requirement of such jobs such that $W = A(\mathcal{R}, \tau_i, 0, t)$. In $\mathcal{I}$, only released jobs can be scheduled and will not execute for more than their execution requirement. Thus, $A(\mathcal{I}, \tau_i, 0, t) \leq W$ holds as well. Therefore, by Def. 7, we have $\mathsf{Lag}_i(t) = A(\mathcal{I}, \tau_i, 0, t) - A(\mathcal{R}, \tau_i, 0, t) \leq 0$, a contradiction. ◀

▶ **Lemma 10** (Lemma 2 of [9]). *If task τ_i is pending at time t in $\mathcal{R}$, then*

$$t - \frac{\mathsf{Lag}_i(t)}{u_i} < d_i(t) \leq t - \frac{\mathsf{Lag}_i(t)}{u_i} + T_i. \tag{1}$$

Proof. Let $e_i(t)$ denote the remaining execution requirement for the ready job $J_{i,j}$ of τ_i at time t. Because this job is ready, it must not be complete, hence

$$0 < e_i(t) \leq C_i. \tag{2}$$

All jobs of τ_i prior to $J_{i,j}$ must have been completed by time t. Let E denote the total execution requirement of these jobs. Then,

$$A(\mathcal{R}, \tau_i, 0, t) = E + C_i - e_i(t). \tag{3}$$

In $\mathcal{I}$, all prior jobs of τ_i have completed by $r_i(t)$. Within $(r_i(t), t)$, $\mathcal{I}$ continuously executes τ_i on a processor with speed u_i. Thus,

$$A(\mathcal{I}, \tau_i, 0, t) = E + (t - r_i(t))u_i. \tag{4}$$

Given these facts, an expression for $\mathsf{Lag}_i(t)$ can be derived as follows.

$$\begin{aligned}\mathsf{Lag}_i(t) &= \{\text{by Def. 7}\}\\ &\quad A(\mathcal{I},\tau_i,0,t) - A(\mathcal{R},\tau_i,0,t)\\ &= \{\text{by (3) and (4)}\}\\ &\quad (t - r_i(t))u_i - (C_i - e_i(t))\\ &= \{\text{because } d_i(t) = r_i(t) + T_i\}\\ &\quad (t - d_i(t) + T_i)u_i - (C_i - e_i(t))\\ &= \{\text{because } u_iT_i = C_i\}\\ &\quad (t - d_i(t))u_i + e_i(t)\end{aligned}$$

By (2) and the above expression, we have

$$(t - d_i(t))u_i < \mathsf{Lag}_i(t) \leq (t - d_i(t))u_i + C_i, \tag{5}$$

which (using $T_i = C_i/u_i$) can be rearranged to obtain (1). ◀

▶ **Corollary 11** (Corollary 1 of [9]). *If for some $L > 0$ we have $\forall t, \mathsf{Lag}_i(t) \leq L$, then the tardiness of task τ_i does not exceed L/u_i.*

Proof. We prove the corollary by contradiction. Suppose that

$$\mathsf{Lag}_i(t) \leq L \tag{6}$$

holds but τ_i has tardiness exceeding L/u_i. Then, there exists a job $J_{i,j}$ that is pending at time $t \geq d_{i,j}$ where

$$t - d_{i,j} > L/u_i. \tag{7}$$

Because $J_{i,j}$ is pending at t, τ_i's ready job cannot have been released later than $J_{i,j}$. Thus, $d_i(t) \leq d_{i,j}$. Therefore,

$$\begin{aligned}t - d_i(t) &\geq t - d_{i,j}\\ &> \{\text{by (7)}\}\\ &\quad L/u_i\\ &\geq \{\text{by (6)}\}\\ &\quad \mathsf{Lag}_i(t)/u_i.\end{aligned}$$

Rearrangement yields $t - \mathsf{Lag}_i(t)/u_i > d_i(t)$, which contradicts Lemma 10. ◀

▶ **Lemma 12** (Lemma 4 of [9]). *If a task τ_i has a pending job at t and for a task τ_j we have*

$$\frac{1}{u_j}\mathsf{Lag}_j(t) + T_{max} \leq \frac{1}{u_i}\mathsf{Lag}_i(t), \tag{8}$$

then $d_i(t) < d_j(t)$.

Proof. Assume τ_j is pending at t, as otherwise $d_j(t) = \infty$, and we trivially have $d_i(t) < d_j(t)$.

$$\begin{aligned}
d_i(t) \leq &\ \{\text{by Lemma 10}\} \\
&\quad t - \frac{\mathsf{Lag}_i(t)}{u_i} + T_i \\
&\leq \{\text{by (8)}\} \\
&\quad t - \frac{\mathsf{Lag}_j(t)}{u_j} - T_{max} + T_i \\
&\leq \{\text{because } -T_{max} + T_i \leq 0\} \\
&\quad t - \frac{\mathsf{Lag}_j(t)}{u_j} \\
&< \{\text{by Lemma 10}\} \\
&\quad d_j(t)
\end{aligned}$$

◀

As a reminder, we are considering the properties in this section in the context of Unrelated. This means these properties apply to all of our considered models.

4 Tardiness Bounds for HP-LAG-Compliant Schedulers

In this paper, we consider two Identical generalizations: Uniform and Identical-Aff. In order to prove GEDF's SRT-optimality under these different processor models, we provide an abstraction layer called HP-LAG. Informally, HP-LAG states that the LAG of any subset of tasks τ' with the earliest deadlines is temporarily non-increasing.

HP-LAG: For any time instant t, if $\tau' \subseteq \tau$ is a set of pending tasks such that $\forall \tau_h \in \tau'$ and $\forall \tau_\ell \in \tau/\tau'$ we have $d_h(t) < d_\ell(t)$, then $\exists \delta > 0$ such that $\forall t' \in (t, t+\delta) : \mathsf{LAG}(\tau',\ t) \geq \mathsf{LAG}(\tau',\ t')$.

▶ **Definition 13.** *We say that a scheduler is HP-LAG-compliant under a given platform if HP-LAG holds for any feasible task system τ under said platform model.*

In this section, we show that every HP-LAG-compliant scheduler ensures bounded tardiness under its considered platform model. We do this by extending the approach of [9] by maintaining as invariants bounds on the LAG of *every* task subset; in [9], only a *linear* (with respect to m) number of invariants is instead maintained. The LAG bound we define for any subset of tasks $\tau' \subseteq \tau$ is

$$\beta(\tau') = \frac{T_{max}}{2u_{min}} U_{\tau'} \left(2U_\tau - U_{\tau'}\right). \tag{9}$$

We will prove by contradiction that $\forall \tau' \subseteq \tau\ \forall t : \mathsf{LAG}(\tau',\ t) \leq \beta(\tau')$ holds for any HP-LAG-compliant scheduler. We continue to consider all properties in the context of Unrelated. Thus, Lemmas 16, 17, 18, and 19 below hold for any scheduler that is HP-LAG-compliant under any processor model that Unrelated generalizes. We begin by defining a set of time instants that must exist if our LAG bounds are violated.

▶ **Definition 14.** *We call a time instant t invalid if $\exists \tau' \subseteq \tau$ such that $\forall \delta > 0\ \exists t' \in (t, t+\delta)$: $\mathsf{LAG}(\tau',\ t') > \beta(\tau')$. τ' is called an attestant set of the invalid instant t.*

Note that for any invalid instant, $\emptyset$ is never an attestant set because for any time instant t we have $\mathsf{LAG}(\emptyset,\ t) = 0$ and $\beta(\emptyset) = 0$.

▶ **Definition 15.** *If at least one invalid time instant exists, then we call the first*[2] *such instant a boundary instant, denoted t_b. We let τ^b denote an arbitrary attestant set of t_b. We call any task from τ^b a boundary task.*

Because t_b is the first invalid instant, we can prove specific bounds on Lag values at t_b.

▶ **Lemma 16.** *For the boundary instant t_b, the following three expressions hold.*

$$\forall \tau' \subseteq \tau : \mathsf{LAG}(\tau',\ t_b)\ \leq \beta(\tau') \tag{10}$$

$$\mathsf{LAG}(\tau^b,\ t_b) = \beta(\tau^b) \tag{11}$$

$$\forall \delta > 0\ \exists t' \in (t_b, t_b+\delta) : \mathsf{LAG}(\tau^b,\ t') > \beta(\tau^b) \tag{12}$$

Proof. We prove (10) by contradiction. If for some $\tau' \subseteq \tau$, $\mathsf{LAG}(\tau',\ t_b) > \beta(\tau')$, then, by Lemma 8 (continuity of $\mathsf{LAG}(\tau',\ t)$), $\exists \delta > 0\ \forall t' \in (t_b - \delta, t_b)$: $\mathsf{LAG}(\tau',\ t') > \beta(\tau')$. Thus, time instant $t_b - \delta$ is invalid with attestant set τ', which contradicts Def. 15.

By Defs. 14 and 15, $\forall \delta > 0\ \exists t' \in (t_b, t_b+\delta)$: $\mathsf{LAG}(\tau^b,\ t') > \beta(\tau^b)$. By Lemma 8 (continuity of $\mathsf{LAG}(\tau^b,\ t)$), we have $\mathsf{LAG}(\tau^b,\ t_b) \geq \beta(\tau^b)$. By (10), $\mathsf{LAG}(\tau^b,\ t_b) \leq \beta(\tau^b)$, so (11) holds.

(12) follows from Defs. 14 and 15. ◀

▶ **Lemma 17.** *For any boundary task τ_i at t_b, $\mathsf{Lag}_i(t_b) \geq \dfrac{T_{max}}{2u_{min}}(2u_iU_\tau - 2u_iU_{\tau^b} + u_i^2)$.*

Proof.

$$\begin{aligned}
\mathsf{Lag}_i(t_b) &= \{\text{by Def. 7}\}\\
&\quad \mathsf{LAG}(\tau^b,\ t_b) - \mathsf{LAG}(\tau^b/\{\tau_i\},\ t_b)\\
&\geq \{\text{by (11), } \mathsf{LAG}(\tau^b,\ t_b) = \beta(\tau^b)\text{, and by (10), } \mathsf{LAG}(\tau^b/\{\tau_i\},\ t_b) \leq \beta(\tau^b/\{\tau_i\})\}\\
&\quad \beta(\tau^b) - \beta(\tau^b/\{\tau_i\})\\
&= \{\text{by (9)}\}\\
&\quad \frac{T_{max}}{2u_{min}}\left[U_{\tau^b}\left(2U_\tau - U_{\tau^b}\right)\right] - \frac{T_{max}}{2u_{min}}\left[U_{\tau^b/\{\tau_i\}}\left(2U_\tau - U_{\tau^b/\{\tau_i\}}\right)\right]\\
&= \{\text{by the definition of } U_{\tau^b/\{\tau_i\}}\}\\
&\quad \frac{T_{max}}{2u_{min}}\left[U_{\tau^b}\left(2U_\tau - U_{\tau^b}\right)\right] - \frac{T_{max}}{2u_{min}}\left[\left(U_{\tau^b} - u_i\right)\left(2U_\tau - U_{\tau^b} + u_i\right)\right]\\
&= \{\text{rearranging}\}\\
&\quad \frac{T_{max}}{2u_{min}}(2u_iU_\tau - 2u_iU_{\tau^b} + u_i^2)
\end{aligned}$$ ◀

Note that the Lag lower bound from Lemma 17 is strictly positive, because $U_\tau \geq U_{\tau^b}$. Thus, by Lemma 9, any boundary task τ_i is pending at t_b. This proves the following lemma.

▶ **Lemma 18.** *At the boundary time instant t_b, every boundary task has a pending job.*

As shown next, similar reasoning as in Lemma 17 can be used to upper bound the Lag of non-boundary tasks. This allows us to establish a relationship between the deadlines of boundary and non-boundary tasks.

[2] It can be shown that the infimum of all invalid instants is itself an invalid instant. Hence, the first invalid instant, t_b, is well-defined.

▶ **Lemma 19.** *At time instant t_b, every boundary task has an earlier deadline than any non-boundary task (i.e., from τ/τ^b).*

Proof. For any non-boundary task $\tau_j \in \tau/\tau^b$, we have the following.

$$\begin{aligned}
\mathsf{Lag}_j(t_b) &= \{\text{by Def. 7}\}\\
&\quad \mathsf{LAG}(\tau^b \cup \{\tau_j\},\ t_b) - \mathsf{LAG}(\tau^b,\ t_b)\\
&\le \{\text{by (10), } \mathsf{LAG}(\tau^b \cup \{\tau_j\},\ t_b) \le \beta(\tau^b \cup \{\tau_j\}), \text{ and by (11), } \mathsf{LAG}(\tau^b,\ t_b) = \beta(\tau^b)\}\\
&\quad \beta(\tau^b \cup \{\tau_j\}) - \beta(\tau^b)\\
&= \{\text{by (9)}\}\\
&\quad \frac{T_{max}}{2u_{min}}\left[U_{\tau^b\cup\{\tau_j\}}\left(2U_\tau - U_{\tau^b\cup\{\tau_j\}}\right)\right] - \frac{T_{max}}{2u_{min}}\left[U_{\tau^b}\left(2U_\tau - U_{\tau^b}\right)\right]\\
&= \{\text{by the definition of } U_{\tau^b\cup\{\tau_j\}}\}\\
&\quad \frac{T_{max}}{2u_{min}}\left[\left(U_{\tau^b} + u_j\right)\left(2U_\tau - U_{\tau^b} - u_j\right)\right] - \frac{T_{max}}{2u_{min}}\left[U_{\tau^b}\left(2U_\tau - U_{\tau^b}\right)\right]\\
&= \{\text{rearranging}\}\\
&\quad \frac{T_{max}}{2u_{min}}(2u_jU_\tau - 2u_jU_{\tau^b} - u_j^2)
\end{aligned} \tag{13}$$

To conclude the proof, we show that the Lag of a boundary task $\tau_i \in \tau^b$ and the Lag of a non-boundary task $\tau_j \in \tau/\tau^b$ together satisfy the requirement specified in Lemma 12.

$$\begin{aligned}
\frac{1}{u_j}\mathsf{Lag}_j(t_b) + T_{max} &\le \{\text{by (13)}\}\\
&\quad \frac{1}{u_j}\frac{T_{max}}{2u_{min}}(2u_jU_\tau - 2u_jU_{\tau^b} - u_j^2) + T_{max}\\
&= \{\text{factor in } 1/u_j \text{ and out } 1/u_i \text{ from } 2u_jU_\tau - 2u_jU_{\tau^b}\}\\
&\quad \frac{1}{u_i}\frac{T_{max}}{2u_{min}}(2u_iU_\tau - 2u_iU_{\tau^b}) + T_{max}\left(1 - \frac{u_j}{2u_{min}}\right)\\
&\le \{2u_{min} - u_j = u_{min} + (u_{min} - u_j) \le u_{min} \le u_i\}\\
&\quad \frac{1}{u_i}\frac{T_{max}}{2u_{min}}(2u_iU_\tau - 2u_iU_{\tau^b}) + T_{max}\frac{1}{u_i}\left(\frac{u_i^2}{2u_{min}}\right)\\
&\le \{\text{by Lemma 17}\}\\
&\quad \mathsf{Lag}_i(t_b)
\end{aligned} \tag{14}$$

By Lemma 18, a boundary task τ_i is pending, and by Lemma 12, its deadline is earlier than task $\tau_j \in \tau/\tau^b$ (if τ_j has no pending job, then $d_j(t) = \infty$ by Def. 1) at time t_b. ◀

▶ **Theorem 20.** *If a scheduler is HP-LAG-compliant under its considered platform, then for any feasible task system τ, the tardiness of task $\tau_i \in \tau$ is at most*

$$\frac{T_{max}}{2u_{min}}(2U_\tau - u_i). \tag{15}$$

Proof. If there exists at least one invalid instant, we can define the boundary time instant t_b with an attestant set τ^b. By Lemma 19, tasks in τ^b have earlier deadlines than any task in τ/τ^b. Thus, by HP-LAG with $\tau' = \tau^b$,

$$\exists\delta > 0\ \forall t \in (t_b, t_b + \delta):\ \mathsf{LAG}(\tau^b,\ t_b) \ge \mathsf{LAG}(\tau^b,\ t). \tag{16}$$

However, by Lemma 16,

$$\forall \delta > 0\ \exists t \in (t_b, t_b + \delta) :\ \mathsf{LAG}(\tau^b,\ t) > \beta(\tau^b) = \mathsf{LAG}(\tau^b,\ t_b),$$

which contradicts (16). Because the existence of t_b leads to a contradiction, there is no first invalid instant, and hence there are no invalid time instants. Thus, by Def. 14,

$$\forall \tau' \subseteq \tau\ \forall t \geq 0\ \exists \delta > 0\ \forall t' \in (t, t + \delta) :\ \mathsf{LAG}(\tau',\ t') \leq \beta(\tau').$$

By Lemma 8, it follows that

$$\forall \tau' \subseteq \tau\ \forall t \geq 0 :\ \mathsf{LAG}(\tau',\ t) \leq \beta(\tau'). \tag{17}$$

Hence, for any task τ_i and any time instant t,

$$\begin{aligned} \mathsf{Lag}_i(t) &= \{\text{by Def. 7}\} \\ &\quad \mathsf{LAG}(\{\tau_i\},\ t) \\ &\leq \{\text{by (17) with } \tau' = \{\tau_i\}\} \\ &\quad \beta(\{\tau_i\}) \\ &= \{\text{by (9)}\} \\ &\quad \frac{T_{max}}{2u_{min}} u_i(2U_\tau - u_i). \end{aligned}$$

Thus, by Corollary 11, task τ_i has maximum tardiness at most $\dfrac{T_{max}}{2u_{min}}(2U_\tau - u_i)$. ◀

The theorem above is proved under the context of Unrelated. Thus, the theorem holds for HP-LAG-compliant schedulers under Uniform and Identical-Aff because these models are special cases of Unrelated. In Secs. 5 and 6, we demonstrate that the GEDF generalizations discussed in this work are HP-LAG-compliant.

5 GEDF Tardiness Bounds under the Uniform Model

In this section, we show that UG-GEDF, the generalization of IG-GEDF under Uniform in [9], is HP-LAG-compliant. This result enables us to apply Theorem 20 to obtain tardiness bounds for UG-GEDF that are superior to those in [9].

5.1 Refining GEDF for the Uniform Model

IG-GEDF is not SRT-optimal under Uniform. Consider a single-task system $\tau = \{\tau_1\}$ with $C_1 = 1$ and $T_1 = 2$ (hence $u_1 = 0.5$) running on $\pi = \{\pi_1, \pi_2\}$ with $s_1 = 1$ and $s_2 = 0.1$. This system is clearly feasible if τ_1 always executes on the faster processor π_1. Under IG-GEDF, however, it is legal for τ_1 to be continuously scheduled on the slower processor π_2. This would lead to unbounded tardiness, as π_2's speed is lower than τ_1's utilization.

Such counterexamples led Yang and Anderson to define UG-GEDF as below. For the remainder of this section, we assume that processors are indexed by decreasing speed.

UG-GEDF:	At any time instant, the ready job of the pending task with the k^{th} earliest deadline is scheduled on π_k for $k \in [1, m]$.

5.2 HP-LAG-Compliance for UG-GEDF

To prove HP-LAG-compliance, we reference the feasibility condition under Uniform, which references the following definition.

▶ **Definition 21.** *Let $S_k = \sum_{i=1}^{k} s_i$ for $k \leq m$.*

When tasks are indexed by decreasing utilization, the feasibility condition is as follows [6, 9].

$$\forall k \leq m : \sum_{i=i}^{k} u_i \leq S_k \tag{18}$$

$$U_\tau \leq S_m \tag{19}$$

In terms of subsets of tasks, this condition is equivalent to

$$\forall \tau' \subseteq \tau : U_{\tau'} \leq S_{\min(|\tau'|,m)}. \tag{20}$$

▶ **Lemma 22.** *UG-GEDF is HP-LAG-compliant under* Uniform*.*

Proof. By Def. 13, we need only consider the case that task system τ is feasible under Uniform. Let $\tau' \subseteq \tau$ be any subset as defined in HP-LAG for any time instant t. HP-LAG states that the tasks in τ' have earlier deadlines at time t than any tasks outside of τ'. Under UG-GEDF, tasks from τ' occupy the $\min(|\tau'|, m)$ fastest processors. Because UG-GEDF is a non-fluid scheduler (see Def. 4), for any time instant t, for some $\delta > 0$ we have that $(t, t+\delta)$ is a continuous scheduling interval (see Def. 2). For any $t' \in (t, t+\delta)$,

$$\begin{aligned}
\mathsf{LAG}(\tau',\ t') =& \{\text{by Def. 7}\} \\
& \mathsf{LAG}(\tau',\ t) + \sum_{\tau_i \in \tau'} A(\mathcal{I}, \tau_i, t, t') - \sum_{\tau_i \in \tau'} A(\mathcal{R}, \tau_i, t, t') \\
=& \{\text{by Defs. 3 and 6}\} \\
& \mathsf{LAG}(\tau',\ t) + \sum_{\tau_i \in \tau'} u_i(t'-t) - \sum_{\tau_i \in \tau'} A(\mathcal{R}, \tau_i, t, t') \\
=& \{\text{tasks from } \tau' \text{ occupy processors with speeds } s_1, ..., s_{\min(|\tau'|,m)}\} \\
& \mathsf{LAG}(\tau',\ t) + \sum_{\tau_i \in \tau'} u_i(t'-t) - \sum_{i=1}^{\min(|\tau'|,m)} s_i(t'-t) \\
=& \{\text{by Def. 21}\} \\
& \mathsf{LAG}(\tau',\ t) + \sum_{\tau_i \in \tau'} u_i(t'-t) - S_{\min(|\tau'|,m)} \cdot (t'-t) \\
=& \mathsf{LAG}(\tau',\ t) + (t'-t)\left[U_{\tau'} - S_{\min(|\tau'|,m)}\right] \\
\leq& \{t' > t \text{ by definition and } U_{\tau'} \leq S_{\min(|\tau'|,m)} \text{ by (20)}\} \\
& \mathsf{LAG}(\tau',\ t).
\end{aligned}$$

Therefore, UG-GEDF is HP-LAG-compliant. ◀

Because we have proven in Lemma 22 that UG-GEDF is HP-LAG-compliant, we have the following corollary of Theorem 20.

▶ **Corollary 23.** *Under UG-GEDF on a* Uniform *multiprocessor executing a feasible task system, the tardiness of any task τ_i is at most (15).*

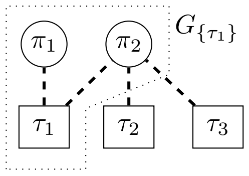


Figure 1 Affinity graph (AG) example.

We assumed without loss of generality that tasks (processors) are indexed by decreasing utilizations (speeds) to make stating UG-GEDF and the Uniform feasibility condition simpler. We no longer keep these assumptions in the following sections.

6 GEDF Tardiness Bounds under the Identical Model with Affinities

In this section, we generalize the IG-GEDF scheduling rules under the context of Identical-Aff. We show that the resulting scheduling policy, IA-GEDF, is HP-LAG-compliant. Thus, Theorem 20 ensures bounded tardiness because Identical-Aff is a special case of Unrelated.

Under Identical-Aff, all processors have speed equal to 1.0. As in [8], we use in our analysis the concept of an affinity graph.

▶ **Definition 24.** *An affinity graph (AG) G_τ of a task system τ on platform π with affinity masks is a undirected bipartite graph containing one vertex for each task and each processor. An edge exists between τ_i and π_j if and only if $\pi_j \in \alpha_i$. For any $\tau' \subseteq \tau$, we let $G_{\tau'}$ denote the subgraph of G_τ containing only the vertices that correspond to the tasks in τ' and the processors in the union of their affinity masks.*

▶ **Example 25.** An example G_τ for the task system $\tau = \{\tau_1, \tau_2, \tau_3\}$ on $\pi = \{\pi_1, \pi_2\}$ with $\alpha_1 = \{\pi_1, \pi_2\}$ and $\alpha_2 = \alpha_3 = \{\pi_2\}$ is given in Fig. 1. The same figure also shows $G_{\{\tau_1\}}$.

6.1 Refining GEDF for the Identical Model with Affinities

Unlike under Identical or Uniform, it is not always possible to schedule the m tasks with the earliest deadlines under Identical-Aff. Consider the example in Fig. 1 with two processors and three tasks. If all tasks are pending at a time instant t and the deadlines are such that $d_2(t) < d_3(t) < d_1(t)$ holds, then the tasks with the earliest deadlines, τ_2 and τ_3, cannot be simultaneously scheduled because they share a single processor.

The choice of processor assignments can also leave a processor idle, i.e., with no job to execute. Consider again Fig. 1 with the assumption that only τ_1 is pending and is assigned to π_2. Suppose that τ_2 at time instant t releases a job such that $d_1(t) < d_2(t)$. Under IG-GEDF, a lower-priority task such as τ_2 would be scheduled on π_1, but affinity-mask restrictions disallow this. Because $d_1(t) < d_2(t)$, τ_1 has higher priority, and τ_2 is not scheduled, leaving processor π_1 idle. However, forcing τ_1 to migrate to π_1 is a more efficient use of processor capacity in this example. The problem here is that the availability of processors for different tasks under affinity scheduling is not symmetrical.

Under IG-GEDF, a preemption only affects the preempting and preempted tasks and a single processor. We define IA-GEDF to extend this preemption rule to avoid unnecessary idleness. To formally specify IA-GEDF, we require several graph-theory definitions.

▶ **Definition 26.** *A matching of a graph G with edge set E is an edge set $M \subseteq E$ such that no two edges in M share a common vertex. A vertex is matched if it is an endpoint of one of the edges in the matching; otherwise, the vertex is unmatched.*

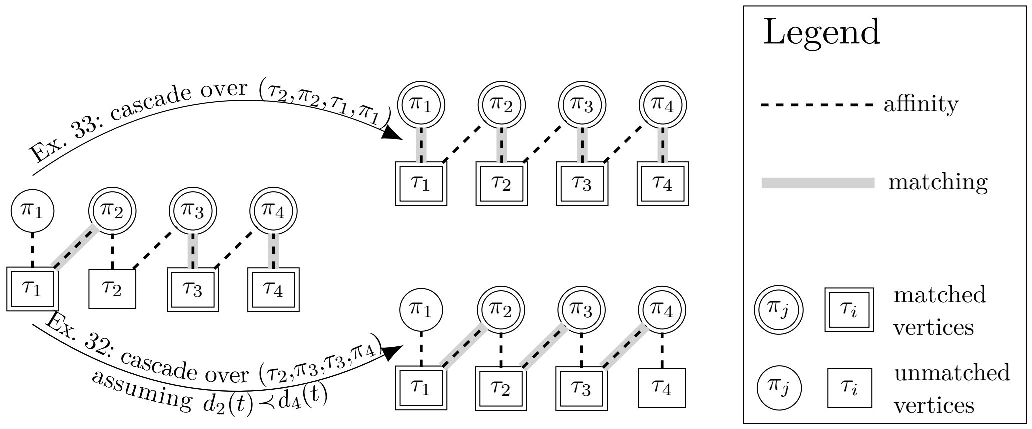

Legend
affinity
matching
matched vertices
unmatched vertices
Ex. 33: cascade over (τ_2,π_2,τ_1,π_1)
Ex. 32: cascade over (τ_2,π_3,τ_3,π_4) assuming $d_2(t) \prec d_4(t)$
π_1 π_2 π_3 π_4
τ_1 τ_2 τ_3 τ_4
π_j τ_i

Figure 2 Two examples of a scheduling cascade.

▶ **Example 27.** An example matching can be found in the left graph of Fig. 2. We have the matching $M = \{(\tau_1, \pi_2), (\tau_3, \pi_3), (\tau_4, \pi_4)\}$, with vertices π_1 and τ_2 being unmatched.

Observe that any assignment of tasks to processors at runtime that obeys the tasks' affinity masks defines a matching of G_τ (and vice versa). While the concept of a matching can be applied to any graph, we restrict attention to only AGs and their subgraphs.

▶ **Definition 28.** *An alternating path p of a matching in a graph is a path that begins with an unmatched "task" vertex, has edges that alternately are in the matching and not in the matching, and ends with a "processor" vertex. An augmenting path p of a matching is an alternating path that ends with an unmatched vertex.*

▶ **Example 29.** In the left graph of Fig. 2, $(\tau_2, \pi_3, \tau_3, \pi_4)$ is an alternating path that is not an augmenting path for the given matching, and $(\tau_2,\pi_2,\tau_1,\pi_1)$ is an augmenting path.

Observe that an alternating path must have an odd number of edges because its first and last vertices are task and processor vertices, respectively. The following lemma establishes a relationship between augmenting and alternating paths.

▶ **Lemma 30.** *Consider a matching M for G_τ and $\tau' \subseteq \tau$. Let M' denote the set of edges of M that are present in $G_{\tau'}$. Then, M' is a matching in $G_{\tau'}$. Furthermore, if p is an augmenting path of the matching M' in $G_{\tau'}$, then p is an alternating path of M in G_τ.*

Proof. By Def. 26, no two edges in M share a common vertex. This does not change when removing edges and vertices from G_τ and edges from M, through which we get $G_{\tau'}$ and M', respectively. Thus, M' is also a matching in $G_{\tau'}$.

Let the first vertex of p (as mentioned in the lemma statement) be task τ_i. By Def. 28, τ_i is unmatched in $G_{\tau'}$. By Def. 24, $G_{\tau'}$ contains τ_i and all the processors in its affinity mask. Hence, all edges from τ_i that are in G_τ are also in $G_{\tau'}$. Thus, τ_i is also unmatched in G_τ.

By definition, M' contains all the edges in M that are also in $G_{\tau'}$. Hence, an edge of $G_{\tau'}$ is in M' if and only if it is also in M, which applies to all edges of p because it is contained in $G_{\tau'}$. Because an augmenting path is a special case of an alternating path, edges of p are alternately in and not in M', and hence, alternately in and not in M. This fact and the fact that the first vertex of p, task τ_i, is unmatched in M make p an alternating path in G_τ. ◀

Algorithm 1: The scheduling cascade algorithm.

```
Input: Matching $M$ of the AG $G_\tau$ at time $t$;
Alternating path $p = (\tau_{i_1}, \pi_{j_1}, \tau_{i_2}, \pi_{j_2}, \ldots, \tau_{i_k}, \pi_{j_k})$ such that if $\tau_\ell$ exists such that
$(\tau_\ell, \pi_{j_k}) \in M$, then $d_{i_1}(t) \prec d_\ell(t)$
1 if $\pi_{j_k}$ is matched in $M$ then
2     $\tau_\ell \leftarrow \pi_{j_k}$'s matched task;
3     Remove edge $(\tau_\ell, \pi_{j_k})$ from $M$;
4 for $r \in [1, k-1]$ do
5     Remove edge $(\tau_{i_{r+1}}, \pi_{j_r})$ from $M$;
6 for $r \in [1, k]$ do
7     Add edge $(\tau_{i_r}, \pi_{j_r})$ to $M$;
8 return
```

We now have sufficient terminology to define a generalized notion of a preemption that occurs under Identical-Aff and that we call a *scheduling cascade*. Informally, in a scheduling cascade, an unscheduled task is scheduled by making an idle processor busy or by unscheduling a different task, perhaps on a different processor, with a later deadline. Note that this may require several migrations.

▶ **Definition 31.** *A scheduling cascade at time t via the alternating path p in G_τ changes the task-to-processor assignments (i.e., the matching M at t in G_τ) via Alg. 1.*

We can write $p = (\tau_{i_1}, \pi_{j_1}, \tau_{i_2}, \pi_{j_2}, \ldots, \tau_{i_k}, \pi_{j_k})$ for some $k \geq 1$ and task (resp., processor) indicies $i_1, i_2, \ldots, i_k$ (resp., $j_1, j_2, \ldots, j_k$) because the first vertex of p is a task by Def. 28. Because p is alternating, $\forall r \in [1, k-1] : (\tau_{i_{r+1}}, \pi_{j_r}) \in M$ and $\forall r \in [1, k] : (\tau_{i_r}, \pi_{j_r}) \notin M$.

Note that M remains a matching after a scheduling cascade. All tasks and processors in p are unmatched prior to line 6 and each iteration of line 7 matches a distinct task and processor from the other iterations.

▶ **Example 32.** Consider the lower scheduling cascade in Fig. 2. In the scheduling cascade, we have $\tau_{i_1} = \tau_2$ and $\pi_{j_k} = \pi_4$. Because π_4 is matched to task τ_4, we have $\tau_\ell = \tau_4$. Thus, the condition $d_{i_1}(t) \prec d_\ell(t)$ becomes $d_2(t) \prec d_4(t)$ in this example, and we remove edge (τ_4, π_4) from the matching (line 3). Of the edges in the alternating path, we remove edge (τ_3, π_3) (line 5) and add edges (τ_2, π_3) and (τ_3, π_4) (line 7). This results in a new matching, as indicated in Fig. 2.

▶ **Example 33.** Consider the higher scheduling cascade in Fig. 2. In the scheduling cascade, we have $\tau_{i_1} = \tau_2$ and $\pi_{j_k} = \pi_1$. We do not execute line 3 because π_1 is unmatched. Of the edges in the alternating path, we remove edge (τ_1, π_2) (line 5) and add edges (τ_2, π_2) and (τ_1, π_1) (line 7). This results in a new matching, as indicated in Fig. 2.

The net result of a scheduling cascade is that task τ_{i_1} that was not scheduled prior to the scheduling cascade is now scheduled, and task τ_ℓ (if it exists) with later deadline than τ_{i_1} is now not scheduled. All other tasks that were scheduled in matching M continue to be scheduled after the scheduling cascade, though the other tasks in p have migrated.

To define our GEDF scheduling policy with affinity masks, we define the task-to-processor assignments at scheduling events, i.e., job releases or job completions, and assume these assignments hold between scheduling events.

IA-GEDF:	At every scheduling event, task-to-processor assignments are made such that afterwards no scheduling cascade is possible. These assignments do not change until the next scheduling event.

One might assume that a simpler scheduling policy may be sufficient, but issues arise when weaker scheduling rules are used. For example, `SCHED_DEADLINE` under Identical-Aff is not HP-LAG-compliant. These details can be found in App. C, available online [7].

Because we do not explicitly specify the task-to-processor assignments, there may exist multiple assignments that satisfy IA-GEDF at every scheduling event. Note that every scheduling cascade either schedules an additional task or replaces a scheduled task with an unscheduled task with an earlier deadline. Thus, the total number of possible scheduling cascades per scheduling event is finite. To show IA-GEDF is possible to implement, we created an $O(mn)$ algorithm that computes task-to-processor assignments at every scheduling event that obeys IA-GEDF. With offline preprocessing, the time complexity is reduced to $O(m + \log n)$ per scheduling event. The algorithm and preprocessing details are available in App. A, available online [7]. In App. C, also available online [7], we show that `SCHED_DEADLINE` under Identical-Aff has higher time complexity per scheduling event.

Note that a preemption in IG-GEDF can be interpreted as a scheduling cascade with an alternating path of a single edge. Thus, IA-GEDF reduces to IG-GEDF under Identical.

6.2 HP-LAG-Compliance for IA-GEDF

Here we consider only feasible task systems τ because HP-LAG-compliance is defined only for such systems. Exact feasibility conditions under Identical-Aff were established in [8], but in our reasoning about IA-GEDF's HP-LAG-compliance, we only use a necessary condition for a feasible task system, provided in Lemma 35.

▶ **Definition 34.** *A matching M of a graph G is a maximal matching if $|M| \geq |M'|$ for any matching M' of G, where $|M|$ denotes the number of edges of the matching M.*

▶ **Lemma 35.** *If a task system τ is feasible under Identical-Aff, then for any task subset $\tau' \subseteq \tau$, a maximal matching M' under $G_{\tau'}$ has $|M'| \geq U_{\tau'}$ edges.*

Proof. Suppose otherwise that τ is feasible and there exists τ' such that a maximal matching M' under $G_{\tau'}$ has $|M'|$ edges with $|M'| < U_{\tau'}$. Because M' is maximal, the tasks of τ' are scheduled on at most $|M'|$ processors at any time instant, and hence, for any schedule $\mathcal{R}$ and time instant t, $\sum_{\tau_i \in \tau'} A(\mathcal{R}, \tau_i, 0, t) \leq |M'|t$. Hence, by Defs. 3, 6, 7, and the definition of $U_{\tau'}$, we have $\mathsf{LAG}(\tau',\ t) = \sum_{\tau_i \in \tau'} A(\mathcal{I}, \tau_i, 0, t) - \sum_{\tau_i \in \tau'} A(\mathcal{R}, \tau_i, 0, t) \geq U_{\tau'}t - |M'|t$. Because $U_{\tau'} > |M'|$, $U_{\tau'}t - |M'|t \to \infty$ as $t \to \infty$. Thus, $\mathsf{LAG}(\tau',\ t)$ is unbounded under any schedule $\mathcal{R}$, making τ' unfeasible, which contradicts the asumption that τ is feasible. ◀

We use Berge's Theorem to prove that IA-GEDF is HP-LAG-compliant. Berge's definition of an augmenting path reduces to Def. 28 in the context of an AG and its subgraphs.

▶ **Theorem 36** (Theorem 1 of [3], Berge's Theorem). *A matching M of a graph G is maximal if and only if there is no augmenting path for M and G.*

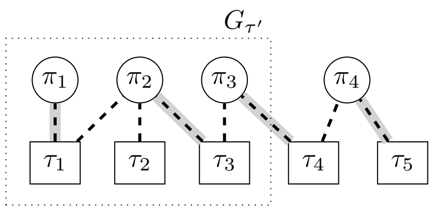


Figure 3 An example graph $G_{\tau'}$ for the proof of Lemma 37. Solid gray edges represent the task-to-processor assignments prior to a scheduling cascade and dashed edges represent affinity.

▶ **Lemma 37.** *IA-GEDF is HP-LAG-compliant under Identical-Aff.*

Proof. We use Fig. 3 to illustrate the key steps of the proof. Consider a matching M in G_τ defined by the IA-GEDF task-to-processor assignments at time t. Let τ' be as defined in HP-LAG at time t ($\tau' = \{\tau_1, \tau_2, \tau_3\}$ in Fig. 3). Consider a matching M' in $G_{\tau'}$ defined by the assignments of tasks in τ' to processors ($M' = \{(\tau_1, \pi_1), (\tau_3, \pi_2)\}$).

We will first show that M' is maximal in $G_{\tau'}$. Suppose otherwise that M' is not maximal in $G_{\tau'}$. Then, by Theorem 36, there exists an augmenting path p for M' in $G_{\tau'}$ ($p = (\tau_2, \pi_2, \tau_3, \pi_3)$). By Lemma 30, p is an alternating path for M in G_τ.

Consider the task τ_h that is represented by the first vertex of p (τ_2), and the processor π_j that is represented by the last vertex of p (π_3). If there is a task τ_ℓ that is scheduled on π_j (τ_4 on π_3), then $\tau_h \in \tau'$ and $\tau_\ell \in \tau/\tau'$ because p is an augmenting path in $G_{\tau'}$. By the definition of τ' in HP-LAG, we have $d_h(t) < d_\ell(t)$ ($d_2(t) < d_4(t)$). Thus, because we have satisfied the input requirements of Alg. 1, we can perform a scheduling cascade via p at time t (remove (τ_4, π_3) and (τ_3, π_2) and add (τ_3, π_3) and (τ_2, π_2) to the matching), which contradicts our definition of IA-GEDF. Hence, M' is maximal.

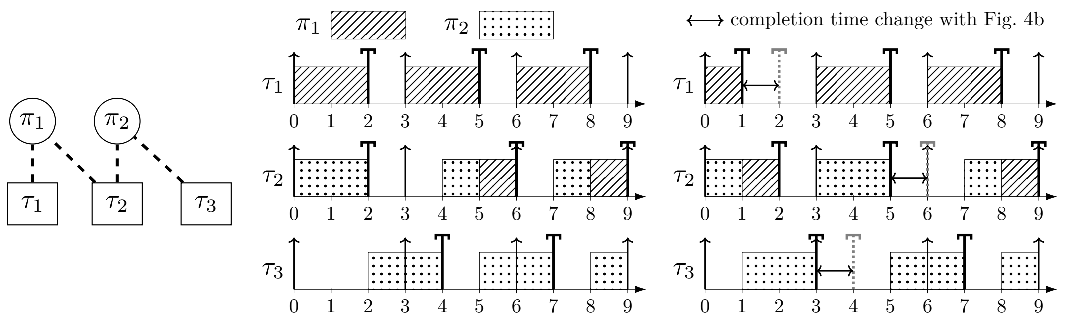
Because IA-GEDF produces a non-fluid schedule (Def. 4), the interval $(t, t+\delta)$ must be a continuous scheduling interval for some $\delta > 0$. Thus, for all $t' \in (t, t+\delta)$,

$$\begin{aligned}
\mathsf{LAG}(\tau',\ t') =& \{\text{by Def. 7}\}\\
& \mathsf{LAG}(\tau',\ t) + \sum_{\tau_i \in \tau'} A(\mathcal{I}, \tau_i, t, t') - \sum_{\tau_i \in \tau'} A(\mathcal{R}, \tau_i, t, t')\\
=& \{\text{by Defs. 3 and 6}\}\\
& \mathsf{LAG}(\tau',\ t) + \sum_{\tau_i \in \tau'} (t'-t)u_i - \sum_{\tau_i \in \tau'} A(\mathcal{R}, \tau_i, t, t')\\
=& \{\text{the number of processors scheduling tasks of } \tau' \text{ is } |M'|\}\\
& \mathsf{LAG}(\tau',\ t) + \sum_{\tau_i \in \tau'} (t'-t)u_i - (t'-t)|M'|\\
=& \{\sum_{\tau_i \in \tau'} u_i = U_{\tau'}\}\\
& \mathsf{LAG}(\tau',\ t) + (t'-t)(U_{\tau'} - |M'|)\\
\le& \{t < t' \text{ and by Lemma 35, } U_{\tau'} \le |M'|\}\\
& \mathsf{LAG}(\tau',\ t).
\end{aligned}$$

Therefore, IA-GEDF is HP-LAG-compliant. ◀

Applying Theorem 20 yields the following.

▶ **Corollary 38.** *Under IA-GEDF on a Identical-Aff multiprocessor executing a feasible task system, the tardiness of any task τ_i is at most (15).*



(a) Affinity graph. **(b)** IA-GEDF schedule with (W). **(c)** IA-GEDF schedule where the first job of τ_1 completes early.

Figure 4 An example task system where removing assumption (W) only causes jobs to complete earlier. When the completion times in the two schedules differ for a job, a dashed gray marker in 4c indicates its original completion time in 4b. Every task has $(C, T) = (2, 3)$.

7 Extending to the Sporadic Task Model

As in [9], we made assumptions (P) and (W). It was proven in Theorems 3 and 4 of [9] that any tardiness bounds derived for UG-GEDF under Uniform assuming (P) and (W) hold without these assumptions. Thus, Corollary 23, which establishes tardiness bounds for UG-GEDF under Uniform, applies to sporadic tasks.

It remains to show that these assumptions can be removed from Corollary 38, which pertains to IA-GEDF on Identical-Aff. We use reasoning similar to [9] to show this, though some details are changed due to the different scheduler and platform. Due to space constraints, we present a proof sketch and provide the formal proofs in App. B, available online [7].

Removing assumption (W). The intuition here is that reducing the execution requirement of a job cannot cause jobs to complete later. Consider Figs. 4b and 4c, which show two schedules under IA-GEDF for two periodic instances of the task system defined in Fig. 4a. In Fig. 4b, assumption (W) is true, while in Fig. 4c, job $J_{1,1}$ completes with 1.0 less execution unit than τ_1's WCER of 2.0. As a result, jobs $J_{2,2}$ and $J_{3,1}$ complete earlier in Fig. 4c than in Fig. 4b, while all other jobs complete at the same time in both schedules (the two schedules converge at time $t = 6$). Thus, reducing the execution time of a job did not increase tardiness for any other job. We prove in App. B, available online [7], that this is always the case. By applying this fact inductively, it follows that tardiness bounds derived assuming (W) hold in systems without (W) under IA-GEDF on Identical-Aff.

Removing assumption (P). In order to remove (P), we use the *varying-period periodic (VPP)* task model, defined in [9]. A VPP task τ_i^V of task set τ^V is defined through its utilization u_i^V. Every job $J_{i,j}^V$ has its own WCER $C_{i,j}^V$. $\max_j C_{i,j}^V$ is denoted as C_i^V. Unlike the sporadic task model, each VPP task τ_i^V releases its first job $J_{i,1}^V$ at time $t = 0$. Afterwards, each job $J_{i,j+1}$ is released *exactly* $T_{i,j}^V = C_{i,j}^V / u_i^V$ time units after the release of $J_{i,j}^V$ for $j \geq 1$. C_i^V / u_i^V is denoted as T_i^V.

It was shown in App. A of [9] that sporadic task systems are special cases of VPP task systems. For example, in Fig. 5, the first timeline represents a sporadic release of jobs by some task τ_i and the second timeline represents a VPP release of jobs by a VPP task τ_i^V with $u_i = u_i^V$ and $C_i = C_i^V$. When the separation between jobs of τ_i is greater than T_i,

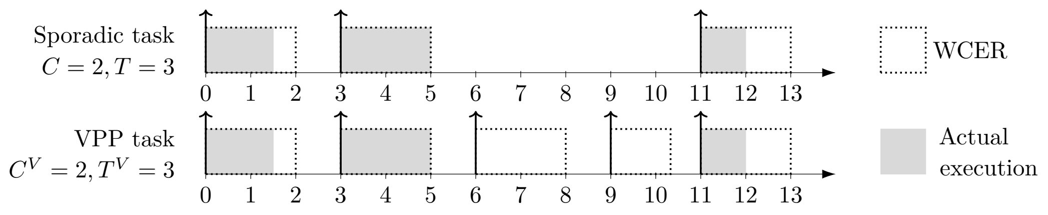


Figure 5 Transformation of a sporadic task to a VPP task.

arbitrarily small VPP jobs with an execution requirement of 0.0 are inserted between the gap in releases, thereby making the second timeline a valid VPP release sequence. Thus, the sporadic release of jobs is also a VPP release of jobs.

Because sporadic task systems are special cases of VPP task systems, our proof obligation is to show that our tardiness bounds apply under the VPP model. The tardiness bounds under IA-GEDF depend only on properties of HP-LAG schedulers, which in turn depend only on the properties in Sec. 3. The fact that these properties hold under the VPP model with assumption (W) (with some reinterpretation of parameters, e.g., substituting u_i with u_i^V and C_i with $C_{i,j}^V$ in proofs) was shown in App. A of [9]. Hence, because removing (W) does not increase tardiness, Corollary 38 holds with VPP task systems (substituting C_{max} with $\max_i C_i^V$ in (15)). If the VPP task system is also a sporadic system, as in Fig. 5, then $\max_i C_i^V = \max_i C_i = C_{max}$. Thus, the tardiness bound in Corollary 38 for sporadic tasks is exactly as written in (15).

8 Problems with Extending to the Uniform Model with Affinities

Of the models listed in Sec. 2.1, we have not addressed Uniform-Aff and Unrelated. In this section, we explain why extending our proof techniques to these models is difficult. The exact proof strategy used in this work cannot be directly applied to the more general models in Sec. 2.1 because HP-LAG may not hold.

▶ **Theorem 39.** *No non-fluid scheduler always satisfies HP-LAG under Uniform-Aff.*

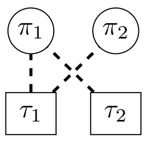
Proof. We prove the theorem by constructing a feasible task system, deadline ordering, and Uniform-Aff platform for which no scheduler satisfies HP-LAG. Consider the task system $\tau = \{\tau_1, \tau_2\}$ with $(C_1, T_1) = (3, 2)$ and $(C_2, T_2) = (4, 4)$. Then, $u_1 = 1.5$ and $u_2 = 1$. τ runs on two processors $\pi = \{\pi_1, \pi_2\}$ with $s_1 = 2$ and $s_2 = 1$. G_τ is illustrated in Fig. 6a.

We know this system is feasible from the schedule in Fig. 6b, which contains a timeline for each task that describes what processor, if any, schedules the task at any time instant. The schedule repeats every four time units. This schedule provides six units of execution to τ_1 and four units of execution to τ_2 every four time units. Because the schedule provides execution to each task at a long-run rate equal to its utilization ($6/4 = 3/2 = u_1$ and $4/4 = u_2$), both tasks have bounded tardiness.

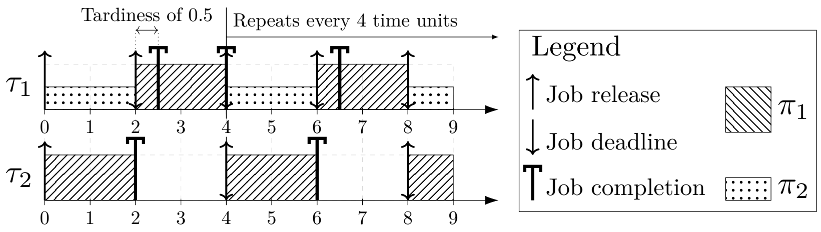
Let the deadline ordering at some time instant t be $d_1(t) < d_2(t)$ (e.g., $t = 0$). Suppose some non-fluid scheduling algorithm is HP-LAG-compliant at t. By Def. 4, we have that $(t, t + \delta)$ is a continuous scheduling interval for some $\delta > 0$. Note that τ' as defined in HP-LAG for this task system may be either $\{\tau_1\}$ or $\{\tau_1, \tau_2\}$ at time t. HP-LAG states for these two task subsets that $\forall t' \in (t, t + \delta)$:

$$\mathsf{LAG}(\{\tau_1\},\ t') \leq \mathsf{LAG}(\{\tau_1\},\ t) \tag{21}$$

$$\mathsf{LAG}(\{\tau_1,\ \tau_2\}, t') \leq \mathsf{LAG}(\{\tau_1,\ \tau_2\}, t) \tag{22}$$



(a) Affinity graph.



(b) A SRT schedule (with a maximum tardiness of 0.5) of a periodic instance of the task system in Theorem 39. The height of a scheduling interval represents the speed of the allocated processor.

Figure 6 Task system considered in the proof of Theorem 39.

We show that τ_1 must execute on π_1 over $(t, t+\delta)$ by contradiction. Consider otherwise that there exists a time instant in $(t, t+\delta)$ where τ_1 executes on π_2. Because $(t, t+\delta)$ is a continuous scheduling interval, by Def. 2, τ_1 executes on π_2 for all $t' \in (t, t+\delta)$. Thus,

$$\begin{aligned}
\mathsf{LAG}(\{\tau_1\},\ t') &= \{\text{by Def. 7}\}\\
&\quad \mathsf{LAG}(\{\tau_1\},\ t) + A(\mathcal{I}, \tau_1, t, t') - A(\mathcal{R}, \tau_1, t, t')\\
&= \{\text{by Defs. 3 and 6}\}\\
&\quad \mathsf{LAG}(\{\tau_1\},\ t) + u_1(t'-t) - A(\mathcal{R}, \tau_1, t, t')\\
&= \{\text{by Def. 3 and the assumption that } \tau_1 \text{ is scheduled on } \pi_2\}\\
&\quad \mathsf{LAG}(\{\tau_1\},\ t) + u_1(t'-t) - s_2(t'-t)\\
&= \{u_1 = 1.5 \text{ and } s_2 = 1\}\\
&\quad \mathsf{LAG}(\{\tau_1\},\ t) + 0.5(t'-t)\\
&> \{t'-t > 0 \text{ by definition}\}\\
&\quad \mathsf{LAG}(\{\tau_1\},\ t),
\end{aligned}$$

which contradicts (21). A similar contradiction arises when τ_1 is not executing for any instant in $(t, t+\delta)$, so any HP-LAG compliant scheduler *must* schedule τ_1 on π_1 during $(t, t+\delta)$. Scheduling τ_1 on π_1 means τ_2 is not scheduled over this interval, because π_1 is the only processor in τ_2's affinity mask.

Consider the LAG of $\{\tau_1, \tau_2\}$ for all $t' \in (t, t+\delta)$ given that τ_1 executes on π_1 and τ_2 is not executing over this interval. Through reasoning similar to that above, we can conclude $\mathsf{LAG}(\{\tau_1,\ \tau_2\}, t') = \mathsf{LAG}(\{\tau_1,\ \tau_2\}, t) + 0.5(t'-t) > \mathsf{LAG}(\{\tau_1,\ \tau_2\}, t)$, which contradicts (22). Because no non-fluid scheduler can simultaneously satisfy (21) and (22), no non-fluid scheduler can satisfy HP-LAG for this feasible task system. ◀

9 Conclusion

We have derived the first ever GEDF tardiness bounds that are polynomial in the number of processors under Uniform. We have also derived for the first time generalized GEDF scheduling rules that are provably SRT-optimal under Identical-Aff. This result shows that the open problem mentioned by Peter Zijlstra and Luca Abeni can be resolved by altering `SCHED_DEADLINE` to be HP-LAG-compliant. In App. A, available online [7], we have provided an algorithm that implements our generalized GEDF scheduling rules with lower time complexity per scheduling event than `SCHED_DEADLINE`, given some preprocessing.

Note that the proofs in Sec. 4 only require that the values of β satisfy certain linear constraints. Thus, lower β values than ours can be derived using linear programming (though the constraint set grows exponentially with the task count). This suggests that the properties in Sec. 3 are sufficient to derive tighter analytical tardiness bounds than ours.

In future work, we will investigate dynamic task systems, where tasks may enter or exit the system and affinity masks may change at runtime. The interaction of affinity masks with dynamic task systems is particularly relevant to `SCHED_DEADLINE`, as admission control with affinity masks is currently broken. We plan to investigate what restrictions must be placed on these dynamics to avoid compromising bounded tardiness, as done in prior work [4] on GEDF without affinity masks. We are also interested in how overhead accounting and non-preemptive sections might be handled as well as how algorithms that satisfy IA-GEDF might be constructed with lower time complexity than in App. A.

The SRT-optimality of GEDF on more general platforms also remains an open problem. We have shown via a counterexample that HP-LAG, a property that applies to both Uniform and Identical-Aff individually, does not hold when the models are combined. It is unknown whether a weaker version of HP-LAG exists that applies to the more general processor models while still being sufficient to bound tardiness. If not, these open problems may require new proof techniques.

References

1 Luca Abeni. SCHED_DEADLINE: a real-time CPU scheduler for Linux. 2nd TuTor at the 38th IEEE Real-Time Systems Symposium, 2017. URL: `https://tutor2017.inria.fr/sched_deadline/`.

2 Luca Abeni et al. Deadline Task Scheduling. Linux kernel documentation, 2018. URL: `https://github.com/torvalds/linux/blob/master/Documentation/scheduler/sched-deadline.txt`.

3 Claude Berge. Two theorems in graph theory. *Proceedings of the National Academy of Sciences*, 43(9):842–844, 1957.

4 Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. Task reweighting under global scheduling on multiprocessors. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 123–167, 2006.

5 UmaMaheswari C. Devi and James H. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. *Real-Time Systems*, 38(2):133–189, 2008.

6 Shelby Funk, Joel Goossens, and Sanjoy Baruah. On-line scheduling on uniform multiprocessors. In *Proceedings of the 22th IEEE Real-Time Systems Symposium*, pages 183–192, 2001.

7 Stephen Tang, Sergey Voronov, and James H. Anderson. GEDF tardiness: Open problems involving uniform multiprocessors and affinity masks resolved. Full version of this paper, available at `http://www.cs.unc.edu/~anderson/papers.html`.

8 Sergey Voronov and James H. Anderson. AM-Red: An optimal semi-partitioned scheduler assuming arbitrary affinity masks. In *Proceedings of the 39th IEEE Real-Time Systems Symposium*, pages 408–420, 2018.

9 Kecheng Yang and James H. Anderson. On the Soft Real-Time Optimality of Global EDF on Uniform Multiprocessors. In *Proceedings of the 38th IEEE Real-Time Systems Symposium*, pages 319–330, 2017.

10 Peter Zijlstra. An update on Real-Time scheduling on Linux. Keynote talk at the 29th Euromicro Conference on Real-Time Systems, 2017. URL: `https://www.ecrts.org/archives/index.php?id=284`.

Dual Priority Scheduling is Not Optimal

Pontus Ekberg
Uppsala University, Sweden
http://user.it.uu.se/~ponek616/
pontus.ekberg@it.uu.se

Abstract

In dual priority scheduling, periodic tasks are executed in a fixed-priority manner, but each job has two phases with different priorities. The second phase is entered after a fixed amount of time has passed since the release of the job, at which point the job changes its priority. Dual priority scheduling was introduced by Burns and Wellings in 1993 and was shown to successfully schedule many task sets that are not schedulable with ordinary (single) fixed-priority scheduling. Burns and Wellings conjectured that dual priority scheduling is an optimal scheduling algorithm for synchronous periodic tasks with implicit deadlines on preemptive uniprocessors. We demonstrate the falsity of this conjecture, as well as of some related conjectures that have since been stated. This is achieved by means of computer-verified counterexamples.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Software and its engineering → Scheduling

Keywords and phrases Scheduling, real time systems, dual priority

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.14

Supplement Material ECRTS 2019 Artifact Evaluation approved artifact available at
https://dx.doi.org/10.4230/DARTS.5.1.1
The source code for a C program that verifies the correctness of the counterexamples in this paper can be found at https://github.com/pontusekberg/dualpriotest/.

Funding *Pontus Ekberg*: This work was supported in part by Vetenskapsrådet, grant 2018-04446.

Acknowledgements The author wants to thank Martina Maggio and Joël Goossens for independently verifying the counterexamples given in this paper, using their own computer implementations.

1 Introduction

Dual priority scheduling was introduced by Burns and Wellings [1] as a technique to schedule implicit-deadline periodic task sets with high utilization "whilst retaining an essentially static priority model". It has the capability to successfully schedule many high-utilization task sets, similar to what can be achieved by earliest deadline first (EDF) scheduling, but it also has low overhead and easy implementation, similar to that of ordinary fixed-priority scheduling.

Curiously, dual priority scheduling works very well despite its simplicity and mostly static behavior. In fact, it works so well that no feasible task set that is not dual priority schedulable has ever been found. Burns and Wellings conjectured in 1993 that dual priority scheduling is *optimal* for scheduling synchronous periodic tasks with implicit deadlines on a preemptive uniprocessor, just like EDF or Least Laxity First scheduling. This open problem has since been restated several times (e.g., in [4]) and stronger conjectures have also been given [7, 6] (these are detailed in Section 1.2). Renewed interest was likely sparked by a paper by Burns [2] dedicated to the problem in the Real-Time Scheduling Open Problems Seminar in 2010.

In this paper we give a counterexample to the conjecture by Burns and Wellings, as well as counterexamples to the related conjectures. These counterexamples can not feasibly be verified by hand because of the vast number of cases that have to be considered. Fortunately, it is straightforward (though still time consuming) to do so by computer. A program that verifies all the counterexamples in this paper can be found by following the link under the heading *Supplement Material* above.

1.1 System model and definitions

We consider dual priority scheduling of synchronous periodic task sets on a preemptive uniprocessor. A task set $\mathcal{T}$ consists of n tasks, $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$, where e_i and p_i are positive integers denoting the worst-case execution time and period of task τ_i, respectively.

In dual priority scheduling, each task τ_i is assigned two fixed priorities: a phase 1 priority π_i^1 and a phase 2 priority π_i^2. Each task is also assigned a phase change point δ_i, which is an integer such that $0 \leqslant \delta_i \leqslant p_i$.

Jobs from task τ_i are initially assigned the phase 1 priority π_i^1. If the job is still active δ_i time units after its release time, it changes its priority to π_i^2 for the remainder of its execution. Jobs are scheduled preemptively and strictly in priority order. Note that a task τ_i with $\delta_i = 0$ or $\delta_i = p_i$ is effectively a single-priority task.

We assume that priorities are distinct and that lower numbers denote higher priorities. We call a complete setting of priorities and phase change points to the tasks in a task set $\mathcal{T}$ a dual priority *configuration* of $\mathcal{T}$. If $\mathcal{T}$ meets all deadlines with a particular configuration, we call that a schedulable configuration.

There are $(2n)! \times \prod_{i=1}^{n}(p_i + 1)$ unique configurations for a task set $\mathcal{T}$ with n tasks, since there are $(2n)!$ different priority settings and $\prod_{i=1}^{n}(p_i + 1)$ different ways of assigning the phase change points. We will also consider some restrictions on the priority orderings:

▶ **Definition 1** (phase 1 RM). *A configuration of $\mathcal{T}$ is* phase 1 RM *if for each pair of tasks $\tau_i, \tau_j \in \mathcal{T}$, we have $\pi_i^1 < \pi_j^1$ if $p_i < p_j$.*

▶ **Definition 2** (phase 2 promoted). *A configuration of $\mathcal{T}$ is* phase 2 promoted *if for each task $\tau_i \in \mathcal{T}$, we have $\pi_i^2 < \pi_i^1$.*

▶ **Definition 3** (RM+RM). *A configuration of $\mathcal{T}$ is* RM+RM *if* (1) *for each pair of tasks $\tau_i, \tau_j \in \mathcal{T}$, we have $\pi_i^1 < \pi_j^1$ and $\pi_i^2 < \pi_j^2$ if $p_i < p_j$, and* (2) $\max_i(\pi_i^2) < \min_i(\pi_i^1)$.

Note that there are many priority settings that fulfill the requirements for phase 1 RM and phase 2 promoted (with distinct periods there are $\binom{2n}{n} \times n!$ and $(2n)!/2^n$ such priority settings, respectively). For RM+RM, however, there is exactly one valid priority setting if periods are distinct. For each priority setting there are $\prod_{i=1}^{n}(p_i + 1)$ unique configurations.

1.2 Related work and conjectures

In the original formulation by Burns and Wellings [1], all configurations were assumed to be both phase 1 RM and phase 2 promoted (see Definitions 1 and 2). Under these restrictions they conjectured that any feasible task set is dual priority schedulable.

▶ **Conjecture 4** (Burns and Wellings [1]). *For any feasible synchronous periodic task set $\mathcal{T}$ with implicit deadlines, there exists a schedulable dual priority configuration of $\mathcal{T}$ that is phase 1 RM and phase 2 promoted.*

However, in Burns' paper at the Real-Time Scheduling Open Problems Seminar [2], it was only assumed that configurations are phase 2 promoted, but conjectured[1] that phase 1 RM is an optimal choice for priorities.

[1] Burns did not use the word “conjecture” here, but wrote that “The phase 1 priorities are probably Rate Monotonic.”

▶ **Conjecture 5** (Burns [2], but see the caveat in footnote 1)**.** *If there exists a schedulable dual priority configuration of $\mathcal{T}$, then there exists a schedulable configuration of $\mathcal{T}$ that is phase 1 RM.*

George et al. [7] and Fautrel et al. [6] made the stronger conjecture that RM+RM is an optimal choice for priorities.

▶ **Conjecture 6** (George et al. [7], Fautrel et al. [6])**.** *If there exists a schedulable dual priority configuration of $\mathcal{T}$, then there exists a schedulable configuration of $\mathcal{T}$ that is RM+RM.*

Fautrel et al. [6] also presented a heuristic called the First Deadline Missed Strategy (FDMS) for assigning the phase change points under RM+RM priorities. FDMS works by initially setting δ_i equal to p_i for every task τ_i. Then it simulates the hyper-period, and the first task τ_i to miss a deadline gets δ_i decremented by 1. This process is then repeated until no task misses a deadline (success) or a task that misses a deadline cannot further decrease its phase change point (failure). Fautrel et al. conjectured that FDMS is an optimal strategy for assigning phase change points with RM+RM priorities.

▶ **Conjecture 7** (Fautrel et al. [6])**.** *If there exists a schedulable dual priority configuration of $\mathcal{T}$ that is RM+RM, then the FDMS heuristic will find a schedulable configuration for $\mathcal{T}$.*

For task sets with only $n = 2$ tasks, Burns [2] proved that dual priority scheduling is indeed optimal. Burns also noted that given k priority levels per task (and $k-1$ phase change points), k-priority scheduling can be made to simulate EDF and is therefore optimal. Pathan [11, 12] showed that $k = n$ priority levels is enough to simulate EDF, where n is the number of tasks. Pathan's result is also applicable to sporadic tasks and tasks without implicit deadlines. Nasri and Fohler [10] also considered scheduling with more than 2 priority levels per task and designed a heuristic for assigning the phase change points.

George et al. [7] demonstrated some properties of dual priority scheduling that differ from those of ordinary (single) fixed-priority scheduling: (1) The first job of each task does not always have the longest response time, (2) a first deadline miss can occur after the first busy period, and (3) the synchronous arrival sequence is not always the worst case.

In addition to the FDMS heuristic conjectured optimal above, Fautrel et al. [6] also designed another heuristic for finding a schedulable configuration. The latter was shown to be sub-optimal, but also faster than FDMS as it does not require repeated simulations of the hyper-period. Gu et al. [8] also designed heuristics for finding schedulable configurations, and presented a sufficient schedulability test for dual priority scheduling that is applicable to constrained-deadline periodic and sporadic tasks.

Last, there are other uses for dual priority scheduling as well. An early example is the work by Davis and Wellings [3]. Rather than using dual priority scheduling to maximize utilization of hard real-time tasks, they used it to improve the timeliness of soft real-time tasks that would otherwise run in the background. By giving the soft real-time tasks priorities in between the phase 1 and phase 2 priorities of the hard-real time tasks, they get the opportunity to run except when a hard-real time task is nearing its deadline.

2 Counterexamples

Here we list counterexamples that disprove the four conjectures in Section 1.2. Because Conjecture 6 implies Conjecture 5, we can disprove both by disproving Conjecture 5. We therefore have three counterexamples: one for disproving Conjecture 4, one for disproving Conjectures 5 and 6, and one for disproving Conjecture 7.

Verifying these counterexamples by hand is intractable, but they are readily verifiable by computer. A C program that verifies the counterexamples can be downloaded from the link given under *Supplement Material* on the first page. The run times reported below are for this program running on an AMD Ryzen 7 1700X CPU with Linux 4.20, compiled by GCC 8.2.1 and `-O3` optimization settings (memory usage is negligible).

The first counterexample actually disproves a weaker version of Conjecture 4 (thus obtaining a stronger result) by virtue of being a feasible task set that is not schedulable with *any* dual priority configuration.

▶ **Counterexample 8.** *The following task set is feasible, but is not dual priority schedulable with any configuration.*

	e_i	p_i
τ_1	8	19
τ_2	13	29
τ_3	9	151
τ_4	14	197

hyper-period:	16 390 597
utilization:	$16\,390\,550 \,/\, 16\,390\,597 \approx 0.9999971$
simulated configurations:	728 082 432 000
verification run time:	~ 61 *hours*

This counterexample is verified by simulating dual priority scheduling of the task set with all $8! \times \prod_{i=1}^{4}(p_i + 1) = 728\,082\,432\,000$ *possible configurations and noting that some deadline is missed in every simulation. The task set is feasible because its utilization is less than 1. [9]*

The above counterexample shows that dual priority scheduling is not optimal for synchronous periodic tasks with implicit deadlines. A corollary is that it is also not optimal for sporadic tasks with implicit deadlines because clearly the task set is feasible also if sporadic, but is still not dual priority schedulable. The sub-optimality also carries over to any task model that is a generalization of either of these two.[2]

The next counterexample is a task set that is dual priority schedulable, but not so with any configuration that is phase 1 RM. It thus disproves Conjectures 5 and 6.

▶ **Counterexample 9.** *The following task set is dual priority schedulable, but not with any configuration that is phase 1 RM.*

	e_i	p_i
τ_1	13	29
τ_2	17	47
τ_3	4	89
τ_4	28	193

hyper-period:	23 412 251
utilization:	$23\,412\,240 \,/\, 23\,412\,251 \approx 0.9999995$
simulated configurations:	42 239 232 001
verification run time:	~ 13 *hours*

This counterexample is verified by

(i) *simulating dual priority scheduling with all* $\binom{8}{4} \times 4! \times \prod_{i=1}^{4}(p_i + 1) = 42\,239\,232\,000$ *possible configurations that are phase 1 RM and noting that some deadline is missed in every simulation, and*

(ii) *simulating the full hyper-period with the (non-phase 1 RM) configuration given below and noting that no deadlines are missed with this configuration.*

A schedulable configuration that is not phase 1 RM:

	π_i^1	π_i^2	δ_i
τ_1	4	0	13
τ_2	5	1	17
τ_3	7	2	42
τ_4	6	3	139

[2] In fact, Gu et al. [8] restated Burns and Wellings' conjecture for constrained-deadline sporadic tasks. We can conclude that this variant also does not hold.

The last counterexample demonstrates that the FDMS heuristic is not guaranteed to find a valid setting of phase change points for RM+RM priorities, even though a schedulable configuration that is RM+RM exists. This disproves Conjecture 7.

▶ **Counterexample 10.** *The following task set is dual priority schedulable with a configuration that is RM+RM, but the FDMS heuristic fails to find a schedulable configuration.*

	e_i	p_i
τ_1	6	11
τ_2	6	20
τ_3	4	46
τ_4	5	74

hyper-period:	187 220
utilization:	46 804 / 46 805 $\approx$ 0.9999786
simulated configurations:	134
verification run time:	$\sim$ 0.01 *seconds*

This counterexample is verified by

(i) *setting priorities to be RM+RM and noting that the FDMS heuristic fails to find phase change points that give a schedulable configuration (FDMS will try* 133 *configurations before failing), and*

(ii) *simulating the full hyper-period with the RM+RM configuration given below and noting that no deadlines are missed with this configuration.*

A schedulable configuration that is RM+RM:

	π_i^1	π_i^2	δ_i
τ_1	4	0	5
τ_2	5	1	3
τ_3	6	2	25
τ_4	7	3	35

In contrast to Counterexample 8, it can be noted that the results of Counterexamples 9 and 10 do not immediately carry over to sporadic tasks. The reason is that the synchronous arrival sequence is not necessarily the worst case [7], and so simulating it for the given custom configurations without any deadline misses does not guarantee schedulability for sporadic tasks with the same configurations.

3 On searching for counterexamples

For completeness we briefly outline the process of finding the reported counterexamples. Roughly speaking, to find these counterexamples one needs some intuition about which classes of task sets to search among (in addition to patience and a fast method for evaluating the schedulability of task sets).

The search space. After some trial and error, and after eventually finding Counterexample 10 using ad hoc methods, a more focused search was carried out in a search space delimited as follows.

- Exactly four tasks per task set.
- Task periods chosen from the first 100 prime numbers (i.e., the primes from 2 to 541).
- Product of periods at most 35 000 000.
- Utilization in the interval $[0.99999, 1]$.

In more detail, task periods were chosen uniformly at random (without replacement) from the 100 first primes, and if their product was at most 35 000 000, each possible combination of execution times for which total utilization ended up in the interval $[0.99999, 1]$ was used to generate a separate candidate task set.

"Quickly" determining schedulability. Each candidate task set was first evaluated using the FDMS heuristic [6] with RM+RM priorities. Despite being suboptimal (as shown in this paper), this heuristic could establish dual priority schedulability for the vast majority of task sets. FDMS runs in exponential time, but for these small task sets it was fast, and importantly much faster than exhaustive checking of configurations. Only task sets that were not FDMS-schedulable were finally evaluated using exhaustive methods as potential counterexamples to Conjectures 4 or 5.

Of course, checking is aborted as soon as a schedulable configuration has been found, and for each configuration, simulation is stopped at the first deadline miss. The latter point is very important as it keeps us from simulating the full hyper-period over and over. In fact, most configurations of the reported counterexamples lead to a deadline miss almost immediately. For example, in Counterexample 8 more than 99.7% of all configurations lead to a deadline miss within the largest period p_4 in that task set (i.e., within the first 197 time units). The average number of time units simulated over the 728 082 432 000 configurations is less than 31, which is smaller than 0.00019% of the hyper-period.

The bound of 35 000 000 on the product of the periods of the generated task sets is put in place to ensure reasonably fast checking of schedulability by limiting the number of combinations of phase change points. It also puts a limit on the hyper-period, but because of the above observation that we tend to not simulate even close to the full hyper-period, this is likely a less important effect.

Why four tasks in each task set? The counterexamples found all have four tasks, which seems to be a sweet spot of being complicated enough for dual priority scheduling to fail, and still small enough for exhaustive checking of all configurations to be feasible. We know from Burns [2] that there are no counterexamples with two tasks, and we have not been able to find counterexamples to any of the conjectures with three tasks either. On the other hand, five tasks seem to be beyond what can be checked with reasonable time and effort: As an example, consider what would happen if we added a fifth task with period 100 to the task set of Counterexample 8 (and adjusted execution times in some suitable way). The number of possible permutations of priorities would increase by a factor of $(2 \times 5)!/(2 \times 4)! = 90$, and the possible combinations of phase change points would increase by a factor of 101. The number of configurations we would have to check would therefore increase by a factor of 9090, and assuming everything else stays the same, the time required for checking all configurations would increase from around 2.5 days to over 60 years! We would have to be patient indeed, or at least have access to lots of processing capacity.

Why prime periods? We want the generated task sets to be as difficult as possible to schedule, and the intuition is that, in addition to high utilization, pairwise coprime periods[3] could be correlated to this. The reasoning is that the behavior of a given dual priority configuration is static with respect to the pattern of job releases (the phase change points are always at a certain offset from the job releases). Some configurations might therefore work well with some patterns, but not with others. By having all possible patterns of job releases occur in the hyper-period, the chance that no configuration will work might therefore increase. We know from the Chinese remainder theorem that all patterns will occur if the periods are pairwise coprime.

[3] Picking the periods from a list of primes was just a straightforward way of making sure they are always pairwise coprime.

The occurrence of counterexamples in the search space. The search space described above is somewhat ad hoc, and no claim is made that it is in any meaningful way the worst for dual priority scheduling. Here is, in any case, a characterization of the occurrence of "interesting" task sets in it. For this, candidate task sets were randomly generated according to the description above until a first unschedulable task set (like the one in Counterexample 8) was found. This happened after evaluating a total of 130 255 candidate task sets. These task sets are classified in the table below.

	# task sets	% of explored search space
Schedulable with RM+RM using FDMS	129 823	~ 99.67%
Schedulable with RM+RM, but not using FDMS	0	0%
Schedulable with phase 1 RM, but not RM+RM	284	~ 0.22%
Schedulable, but not with phase 1 RM	147	~ 0.11%
Unschedulable	1	~ 0.0008%

It can be noted that even in this search space, which was heavily narrowed down to what was believed to be "difficult" task sets, almost all task sets are still schedulable with the FDMS heuristic and RM+RM priorities. It is also interesting that among the evaluated task sets in this experiment, FDMS succeeded in finding a schedulable configuration to all task sets that were schedulable with RM+RM. (However, as is evidenced by Counterexample 10, there are cases where FDMS fails to find schedulable RM+RM configurations.)

4 Conclusions and open problems

Even though we show negative results about dual priority scheduling in this paper, most importantly that it is not an *optimal* algorithm, this does not directly imply that dual priority scheduling is not, in general, a *good* algorithm. It still successfully schedules the vast majority of task sets that have been tried – presumably the very reason it was conjectured optimal in the first place. One could argue, with some justification, that it is close enough to optimal for any *practical* uses. However, there are still major challenges ahead to make full use of the apparent near-optimality of dual priority scheduling. Here we list some of those challenges (see also Burns [2]).

1. **Can we efficiently test whether a schedulable configuration exists?** Had dual priority scheduling been optimal this would be trivial by simply checking whether the utilization is at most 100%, but now this cannot be a sufficient test. The current state-of-the art, to try all possible configurations and see if one works, is utterly unsatisfactory. This problem is clearly in PSPACE, but lower bounds are unknown.
2. **Can we efficiently find a schedulable configuration if one exists?** It is not unlikely that a good solution to the first point would somehow allow us to also find a schedulable configuration, but it is not necessarily so. For example, a non-constructive proof of optimality would have allowed us to use the 100% utilization bound to check for existence of a schedulable configuration, but it might not have helped us find one.
3. **Can we efficiently test whether a given configuration is schedulable?** The current state-of-the-art is to simulate a full hyper-period, which is straightforward but not very scalable. It also does not seem to be applicable to sporadic tasks as George et al. [7] have shown that the synchronous arrival sequence is not guaranteed to be the worst case. This problem is clearly in PSPACE and is also (weakly) NP-hard as it generalizes the ordinary fixed-priority schedulability problem [5].

4. **What is the utilization bound for dual priority scheduling?** Dual priority scheduling is suboptimal, but is more powerful than ordinary fixed-priority scheduling. Its utilization bound must therefore be in the half-open interval $[\ln(2), 1)$,[4] but where? It can be noted that a constant utilization bound can not be an exact schedulability test here.
5. **Does the apparent near-optimality of dual priority scheduling diminish with larger task sets?** Dual priority scheduling appears to work very well, but out of necessity it has only been systematically evaluated on small task sets so far (something that solutions to the issues above might change). At the same time, there is evidence that the number of priorities each task might need is related to the number of tasks in the task set: Burns [2] proved that dual priority scheduling is optimal for task sets with $n = 2$ tasks, while Pathan [11, 12] has shown that, in general, n-priority scheduling is optimal for task sets with n tasks. Does dual priority scheduling seem so good because our observations are biased?
6. **Is phase 2 promoted an optimal choice for priorities?** It is usually assumed in the literature that configurations should be phase 2 promoted (see Definition 2). While this seems very sensible, it has to the best of the author's knowledge never been proven optimal. After all, phase 1 RM and RM+RM seem like sensible choices as well, but they turn out to not be optimal.
7. **Would rational phase change points improve schedulability?** We have assumed that the phase change points, like task parameters, are integer. Integer phase change points are a reasonable assumption as we could always pick the clock cycle as our time unit, and it also seems to be in line with previous work on dual priority scheduling, even though it is not always spelled out. But what if phase change points can be rational? An interesting theoretical question is whether this adds any extra power. Naturally, it is impossible to exhaustively check all configurations in such a model.
8. **Is k-priority scheduling optimal for some constant k?** We know now that dual ($k = 2$) priority scheduling is not optimal, but also that given enough priority levels k per task, k-priority scheduling can be made to simulate EDF and is therefore optimal. The smallest bound on k known for optimal scheduling is $k = n$, where n is the number of tasks, as shown by Pathan [11, 12]. Is there some constant k, such that k-priority scheduling is optimal?
9. **Is there some deeper insight to be gained about why some task sets are unschedulable?** We have demonstrated that dual priority scheduling is not optimal, but the brute-force method with which this is done leaves something to be desired. It is not easy to see what actually makes a certain task set unschedulable, like the one in Counterexample 8. For a given configuration we could, in principle, analyze the concrete schedule to see where the "wrong" scheduling decision was made and explain why and how this happened. Explaining in a satisfiable way why none of billions of different configurations work seems to require some entirely different type of insight, however. This is insight that is currently lacking. Of course, it is entirely possible that there does not actually exist a more elegant and general characterization of dual priority unschedulability than "a task set is unschedulable if all configurations lead to a deadline miss." But if a more elegant characterization does exist, finding it would be key to deeper understanding. Such a characterization would likely be needed for a satisfactory solution to the first point in this list as well.

[4] As is evidenced by Counterexample 8, the utilization bound must in fact be in the slightly smaller interval $[\ln(2), 16\,390\,550 \,/\, 16\,390\,597)$.

References

1 A. Burns and A.J. Wellings. DUAL PRIORITY ASSIGNMENT: A Practical Method for Increasing Processor Utilisation. In *Proceedings of the 5th Euromicro Workshop on Real-Time Systems*, pages 48–53, 1993. doi:10.1109/EMWRT.1993.639052.

2 Alan Burns. Dual priority scheduling: Is the processor utilisation bound 100%? In *Proceedings of the 1st International Real-Time Scheduling Open Problems Seminar (RTSOPS)*, 2010. URL: https://www.cs.york.ac.uk/ftpdir/reports/2010/YCS/455/YCS-2010-455.pdf#page=9.

3 Robert Davis and Andy Wellings. Dual Priority Scheduling. In *Proceedings of the 16th Real-Time Systems Symposium (RTSS)*, pages 100–109, December 1995. doi:10.1109/REAL.1995.495200.

4 Robert I. Davis, Liliana Cucu-Grosjean, Marko Bertogna, and Alan Burns. A review of priority assignment in real-time systems. *Journal of Systems Architecture*, 65:64–82, 2016. doi:10.1016/j.sysarc.2016.04.002.

5 Pontus Ekberg and Wang Yi. Fixed-Priority Schedulability of Sporadic Tasks on Uniprocessors is NP-hard. In *Proceedings of the 38th Real-Time Systems Symposium (RTSS)*, pages 139–146, 2017. doi:10.1109/RTSS.2017.00020.

6 Tristan Fautrel, Laurent George, Joël Goossens, Damien Masson, and Paul Rodriguez. A Practical Sub-Optimal Solution for the Dual Priority Scheduling Problem. In *Proceedings of the 13th International Symposium on Industrial Embedded Systems (SIES)*, June 2018. doi:10.1109/SIES.2018.8442075.

7 Laurent George, Joël Goossens, and Damien Masson. Dual Priority and EDF: a closer look. In *Proceedings of the Work-in-Progress Session of 35th Real-Time Systems Symposium (RTSS-WiP)*, December 2014. URL: https://hal.archives-ouvertes.fr/hal-01217433.

8 Xiaozhe Gu, Arvind Easwaran, and Risat Pathan. Design and Analysis for Dual Priority Scheduling. In *Proceedings of the 21st International Symposium on Real-Time Distributed Computing (ISORC)*, pages 164–173, 2018. doi:10.1109/ISORC.2018.00033.

9 C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973. doi:10.1145/321738.321743.

10 Mitra Nasri and Gerhard Fohler. Some Results in Rate Monotonic Scheduling with Priority Promotion. In *Proceedings of the Work-in-Progress Session of the 27th Euromicro Conference on Real-Time Systems (ECRTS-WiP)*, pages 5–8, 2015. URL: https://people.mpi-sws.org/~mitra/papers/Nasri_WIPECRTS15.pdf.

11 Risat Mahmud Pathan. Unifying Fixed- and Dynamic-Priority Scheduling based on Priority Promotion and an Improved Ready Queue Management Technique. In *Proceedings of the 21st Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 209–220, April 2015. doi:10.1109/RTAS.2015.7108444.

12 Risat Mahmud Pathan. Real-Time Scheduling on Uni- and Multiprocessors based on Priority Promotions. *Leibniz Transactions on Embedded Systems*, 3(1):02–1–02:29, 2016. doi:10.4230/LITES-v003-i001-a002.

NPM-BUNDLE: Non-Preemptive Multitask Scheduling for Jobs with BUNDLE-Based Thread-Level Scheduling

Corey Tessler
Wayne State University, Detroit, Michigan, USA
corey.tessler@wayne.edu

Nathan Fisher
Wayne State University, Detroit, Michigan, USA
fishern@wayne.edu

Abstract

The BUNDLE and BUNDLEP scheduling algorithms are cache-cognizant thread-level scheduling algorithms and associated worst case execution time and cache overhead (WCETO) techniques for hard real-time multi-threaded tasks. The BUNDLE-based approaches utilize the inter-thread cache benefit to reduce WCETO values for jobs. Currently, the BUNDLE-based approaches are limited to scheduling a single task. This work aims to expand the applicability of BUNDLE-based scheduling to multiple task multi-threaded task sets.

BUNDLE-based scheduling leverages knowledge of potential cache conflicts to selectively preempt one thread in favor of another from the same job. This thread-level preemption is a requirement for the run-time behavior and WCETO calculation to receive the benefit of BUNDLE-based approaches. This work proposes scheduling BUNDLE-based jobs non-preemptively according to the earliest deadline first (EDF) policy. Jobs are forbidden from preempting one another, while threads within a job are allowed to preempt other threads.

An accompanying schedulability test is provided, named Threads Per Job (TPJ). TPJ is a novel schedulability test, input is a task set specification which may be transformed (under certain restrictions); dividing threads among tasks in an effort to find a feasible task set. Enhanced by the flexibility to transform task sets and taking advantage of the inter-thread cache benefit, the evaluation shows TPJ scheduling task sets fully preemptive EDF cannot.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Software and its engineering → Real-time schedulability

Keywords and phrases Scheduling algorithms, Cache Memory, Multi-threading, Static Analysis

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.15

Supplement Material ECRTS 2019 Artifact Evaluation approved artifact available at `https://dx.doi.org/10.4230/DARTS.5.1.2`

Funding The research presented in this paper was supported by the National Science Foundation under Grant Nos. CNS-1618185 and IIS-1724227.

1 Introduction

Hard real-time multi-threaded task systems which incorporate cache memory, must account for the variation in execution time and cache related preemption delays found in single-threaded task systems. For multi-threaded task systems, the complexity of cache interactions is increased due to thread-level cache interference and preemptions. Worst-case execution time (WCET) and schedulability analysis of hard real-time multi-threaded tasks commonly treat threads independently [21] or utilize cache management techniques [33] to limit the cache interference.

Analysis techniques focusing on independent treatment or limiting of cache interference exclude the possible benefit of caches. Multi-threaded tasks may benefit from caches. By virtue of sharing the same address space one thread of a task may cache values on behalf of another reducing the total execution time to complete both. This positive effect is referred to as the *inter-thread cache benefit* [26].

Currently, only the BUNDLE [26] and BUNDLEP [27] analysis techniques and cache congnizant thread-level scheduling algorithms incorporate the inter-thread cache benefit into WCET and schedulability analysis. These BUNDLE-based approaches are currently limited to a single multi-threaded task. The primary focus of this work is to provide a scheduling algorithm and schedulability test for multi-threaded task sets with multiple tasks, where individual jobs utilize BUNDLE-based scheduling. As the first scheduling algorithm to incorporate BUNDLE-based thread-level scheduling, a non-preemptive algorithm was chosen to avoid necessary modifications to BUNDLE and BUNDLEP. Non-preemptive EDF was selected as the task-level scheduler, as the proposed schedulability test presented in Section 4 is based upon Baruah's limited-preemption for EDF [6] algorithm.

An additional consideration is made for alternative approaches and the unforeseen benefits to schedulability of thread-level schedulers of non-preemptive multi-threaded jobs. If the WCET of jobs can be expressed as a strictly increasing discrete concave function of the number of threads per job, the schedulability test developed for this work applies without modification to the BUNDLE-based approaches or non-preemptive EDF scheduling.

In the following sections, the key contributions are:

1. A model of hard real-time multi-threaded tasks which is compatible with existing single-threaded models, where tasks sets may be transformed through division of threads.
2. A schedulability test named Threads Per Job (TPJ) that provides a schedulability result and transformed feasible task set if the specified task set could not be scheduled non-preemptively.
3. Proof of TPJ's optimality with respect to non-preemptive multi-threaded feasibility.
4. An improvement to Baruah's [6] non-preemptive chunk algorithm, increasing chunk sizes.
5. An evaluation of over 500,000 task sets, comparing the schedulability ratio of TPJ to those of non-preemptive and (limited) preemptive EDF, with an accompanying implementation available for download [28].

These contributions are presented following the related research of Section 2. Section 3 introduces the proposed model, application of non-preemptive EDF scheduling for thread-level schedulers, and the requirements of task transformation. Section 4 introduces then improves upon the non-preemptive chunk algorithm [6], followed by the TPJ schedulability algorithm and proof of feasibility. Section 5 compares the schedulability ratio of TPJ to other non-preemptive and preemptive scheduling algorithms, before concluding with Section 6.

2 Discussion of Related Research

Single-Threaded Tasks. The challenge of dealing with the non-uniformity of execution times in real-time systems due to cache misses or hits has received considerable attention [34, 30]. In particular, much of the prior real-time systems work on understanding caches vis-à-vis scheduling has focused upon the contention in the cache due to tasks preempting each other. Roughly speaking, a large majority of this research can be classified into two categories: *cache-related preemption delay (CRPD) analysis* and *deferred/limited-preemption scheduling*. The goal of CRPD analysis is to bound the number of cache blocks of a task that need to be reloaded due to evictions caused by a preempting task. The foundation of CRPD analysis is the development of techniques for counting and bounding the number of blocks affected by preemption; this is achieved by categorizing a task's cache blocks into sets of useful cache blocks (UCBs) or evicting cache block (ECBs) [17, 31]. The size of these sets can be used as an upper bound on the cache cost of a preemption. Subsequent research based upon this UCB and ECB categorization has refined these sets and incorporated the CRPD analysis into schedulability analysis [1, 2, 3, 20, 24, 25]. However, please note that these CRPD approaches only quantify the cache effect of preemption into existing scheduling approaches and do not change any scheduling decision based upon the knowledge of preemption.

In limited/deferred-preemption scheduling, a higher-priority task may preempt a lower-priority task only when some condition is satisfied. The overall effect of deferring or limiting preemptions is to reduce the number of times a task may be preempted during its execution. The hope is that by limiting the number of preemptions this will lead to a decrease in the execution time of job due to the cache overhead of preemption. Different conditions for deferring preemptions have been considered. The fixed preemption-point approach [11] selects specific locations in a task code that are most appropriate for the program but preserve the system schedulability. The preemption-threshold scheduling approach [32] sets a threshold that only task with higher-priority than this threshold may preempt a currently-executing lower priority task. The floating preemption-point model [6, 19] computes the maximum time duration that a lower-priority task may delay the preemption of a higher-priority task. Each of the deferred preemption approaches have been shown to limit the number of preemptions but do not incorporate the CRPD overhead cost in its decision on how to defer preemption.

More recently, a line of research has emerged to combine the aspects of CRPD analysis and limited/deferred preemption scheduling by explicitly placing preemption points in the code to minimize CRPD effects. Early heuristics were proposed by Simonson and Patel [23] and Lee et al. [17]. Bril et al. [10] integrated CRPD analysis into preemption-threshold scheduling. Bertogna et al. [8] provide a more formal approach for optimally determining preemptions in programs that can be represented by linear control flowgraphs given the CRPD overhead of each preemption and a bound on the maximum non-preemption region [6]. Later work, extended this to more general control flowgraphs [22] or more precise CRPD characterizations of the preemption costs [14]. However, all of this aforementioned research assumes each task is single-threaded. The techniques proposed in this paper extend the CRPD and limited preemption concepts to scheduling multi-threaded tasks by combining and extending the limited-preemption scheduling results of Baruah [6] to the cache-cognizant thread-level scheduling algorithms that minimize cache contention between threads called `BUNDLE` [26] and `BUNDLEP` [27].

Multi-Threaded Tasks. Cache interference amplifies the variation in execution times of multi-threaded task sets. Threads of the same task share cache locations, with the potential to increase misses and hits depending on the order of execution of threads. This variability is an addition to the variation already present when considering CRPD with other tasks.

There are few works we are aware which directly address the inter-thread variability due to caches in multi-threaded task sets. The approaches focus on isolating execution or managing cache behavior. Memory-Centric Scheduling [4] isolates contentious execution by scheduling tasks according to their cache behavior. To create such isolation, tasks must be PREM-compliant [21], with distinct load and execution phases. Cache management utilize techniques that limit the contention in the cache, such as coloring and blocking found in [33]. These approaches come at a cost of modified or restricted executable objects, reduced cache sizes, or additional cache misses of blocked lines. Yet, with these limitations, the inter-thread variability is not accounted for within multi-threaded tasks.

`BUNDLE` [26] and `BUNDLEP` [27] address inter-thread variability due to cache interactions. These `BUNDLE`-based approaches analyze executable object coupled with a cache-cognizant thread-level scheduling algorithm without the added detriment of modified (or restricted) objects, or cache management penalties. We are not aware of any other technique that addresses inter-thread variability, with the execption of Calandrino's [13] limited cache spread. However, the results of [13] are strictly empirical.

3 Model

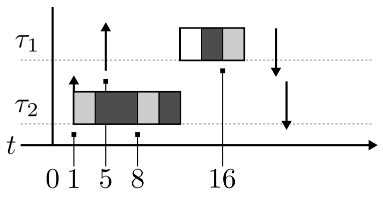
To permit non-preemptive jobs that utilize thread-level schedulers, a new model is proposed in this work. The set of n multi-threaded tasks is given by $\tau = \{\tau_0, \tau_1, ..., \tau_{n-1}\}$. Each job of a task $\tau_i = (p_i, d_i, m_i, c_i : \mathbb{N} \mapsto \mathbb{R}^+)$ has a minimum inter-arrival time of p_i and relative deadline d_i. For every job release of τ_i, a positive integer m_i identical threads are released. Each thread of τ_i executes over the same object o_i on the shared processor. An object is a set of executable machine instructions, mapping to one set of in memory addresses, such that all threads execute the same instruction from the same address. All threads share the same deadline as their job. The worst-case execution time (WCET) of τ_i is a function of the number of threads per job, $c_i(m_i)$.

Scheduling and schedulability analysis proposed in this work relies upon a relationship between the number of threads scheduled per multi-threaded job and the WCET of the job executed non-preemptively. To clarify, the scheduling mechanism proposed in this work precludes preemptions between jobs of different tasks. For threads within a job of a task, a thread-level scheduler may execute threads preemptively. Figure 1a illustrates the scheduling behavior.

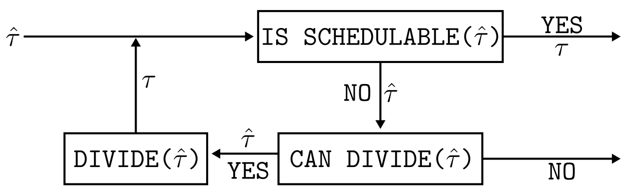
In Figure 1a, at $t = 1$ a job of τ_2 is released. The job of τ_2 cannot be preempted by the job of τ_1 released at $t = 5$. During the execution of τ_2, the two threads (given distinct colors) may preempt one another according to the thread-level scheduler, at $t = 8$ for instance. Thread-level scheduling and preemption decisions are not prescribed by this work. The thread-level scheduling policies of τ_1 and τ_2 are independent of the non-preemptive task-level scheduling of non-preemptive EDF used in this work.

Thread-level scheduling algorithms must be characterized by a WCET function $c_i(m_i)$ for m_i threads per job and $c_i(m_i)$ must be strictly increasing discrete and concave (detailed in Subsection 3.2). Thread-level schedulers that produce concave $c_i(m_i)$ functions establish a relationship between the execution requirements of a task and the number of threads, where the requirement for one job of m_i threads is less than m_i jobs of one thread. For `BUNDLE`-based schedulers, concavity is the result of the inter-thread cache benefit, where $c_i(m) - c_i(m-1) \geq c_i(m+1) - c_i(m)$; it is this relationship the proposed scheduling behavior and analysis seek to exploit.

Not all tasks and thread-level schedulers will produce concave WCET functions. For a task τ_i with a convex WCET function (where there is no benefit in grouping threads together), the m_i threads of τ_i may be replaced with m_i single-threaded tasks. These single-threaded have vacuously concave WCET functions by virtue of executing no more than one thread.



(a) Scheduling Behavior.



(b) Schedulability and Transformable Task Sets.

Figure 1 Scheduling and Schedulability of the Proposed Model.

The task set τ provided by the system designer to schedulability analysis is referred to as the task set *specification*. Commonly [5, 18, 6, 8, 11, 12], task set specifications are immutable in hard-real time models. The number of tasks, their WCET time, period, and deadline are provided by the system designer, not to be changed. Schedulability analysis determines if the task set specification is feasible. In this work, task sets are transformable (obeying some restrictions).

Transformation of a task set exploits the concavity of execution requirements, redistributing the threads of individual tasks to multiple tasks. A greater number of threads per job reduces the WCET of a task but increases the non-preemptive execution requirement. Conversely, a fewer number of threads per task increases the total WCET for all tasks while decreasing the non-preemptive execution requirement. Schedulability analysis in this non-preemptive setting encompasses the search for a distribution of the fixed number threads from the task set specification to a variable number of tasks, resolving the tension between a greater number of tasks and a greater number of threads per task to find a feasible task set.

Under the proposed model, schedulability analysis is a process that begins by considering the current task set named the *anterior* task set $\hat{\tau}$. If the set is schedulable, the set is unmodified and processing ceases with a positive result. If the task set $\hat{\tau}$ cannot be scheduled as described, the task set is transformed into a *posterior* task set τ, and processed again as an anterior set. Processing ceases with a negative result when there are no available transformations of $\hat{\tau}$.

Figure 1b illustrates the schedulability analysis process. Division is the transformative operation of the process and is described in Subsection 3.1. The figure highlights the ability of a single task set to be both anterior and posterior to different sets during processing. To aid in explanation, properties of a task may be referred to in terms of the set the task was transformed from and to. By example, if the number of threads assigned to τ_i in the anterior set $\hat{\tau}$ is reduced by one in the posterior task set τ, the posterior threads of τ_i may be written as $m_i = \hat{m}_i - 1$.

As a process, schedulability analysis of the specified task set serves two purposes under this model. The first, is to determine if there exists a posterior task set which is feasible. Second, to produce the feasible posterior task set if one exists. It is the feasible posterior task set τ found by schedulability analysis that is then deployed on the target architecture. From the system designer's perspective, each task $\tau_i \in \tau$ of the specified task set is a request to execute m_i threads of the object o_i with shared periods p_i and deadlines d_i for **any** posterior task set τ. A task set specification is flexible, for one object there may be multiple tasks with variable numbers of threads per job. However, the specified m_i of a task is a ceiling on any m_i of a posterior task.

3.1 Dividing and Task Parts

A task set may be transformed by *dividing* tasks of the set. Dividing a task reduces the number of threads executed by each job, splitting the anterior task into two or more tasks in the posterior set.

▶ **Definition 1** (Task Division). *In the anterior task set $\hat{\tau}$, a task $\tau_i = (p_i, d_i, c_i(m_i))$ may be* divided *into two (or more) posterior tasks τ_j and τ_k with three restrictions: 1.) the periods of τ_j and τ_k are equal to the period of τ_i 2.) the relative deadlines of τ_j and τ_k are equal to the deadline of τ_i 3.) the sum of threads of τ_j and τ_k are equal to τ_i 4.) the objects of τ_i, τ_j, and τ_k are equal. Enumerated, the restrictions are:*

1. $p_i = p_j = p_k$
2. $d_i = d_j = p_k$
3. $m_i = m_j + m_k$
4. $o_i = o_j = o_k$

▶ **Definition 2** (Partial Tasks). *When an anterior task τ_i is divided into τ_j and τ_k posterior tasks, τ_j and τ_k are referred to as* partial tasks *or* parts *of τ_i.*

▶ **Definition 3** (Partial Task Set). *For convenience, the set of posterior tasks of τ_i is denoted Φ_i and called the* partial task set *of τ_i, where $m_i = \sum_{\tau_k \in \Phi_i} m_k$.*

3.2 Worst-Case Execution Time Function Growth

Motivation for the task model and schedulability analysis process proposed in this work stems from the inter-thread cache benefit of BUNDLE-based scheduling [26, 27]. The previous works [26, 27] are limited to a single task; this work extends the method (non-preemptively) to multiple tasks. Schedulability analysis for BUNDLE-based scheduling algorithms produce, for each task τ_i, a worst-case execution time combined with cache overhead (WCETO) function $c_i(m)$ in terms of m the number of threads per job scheduled in a cache-cognizant manner. For tasks that benefit from BUNDLE-based scheduling and analysis, $c_i(m)$ is a strictly increasing discrete concave function. Tasks that do not are made vacuously concave by restricting jobs to release one thread.

In the WCETO analysis of BUNDLE and BUNDLEP, threads are assigned to paths through the conflict-free region graph of the executable object which maximize their contribution to $c_i(m_i)$. When considering the addition of a thread $m_i + 1$, only the greatest increase in $c_i(m_i)$ is permitted. Subsequently, the addition of thread $m_i + 2$ must increase $c_i(m_i)$ by less than or equal to the increase from $m_i + 1$ or the increase of $m_i + 1$ would not have been maximal. Therefore, for any $m_a < m_b < m_c$ the point $(m_b, c_i(m_b))$ lies above the straight line described by $(m_a, c_i(m_a))$ and $(m_c, c_i(m_c))$ – subsequently, $c_i(m_i)$ is concave.

A consequence of $c_i(m)$'s strictly increasing discrete concavity is a limit on the increase of the WCET as the number of threads increases. This property is referred to as the *concave restricted growth* (*concave growth* for brevity) of $c_i(m)$ and is leveraged in Sections 4 and 5.

▶ **Property 1** (Concavity Restriction on WCET Growth). *For a strictly increasing discrete concave WCET function $c_i(m)$:*

$$\forall m \in \mathbb{N}^+ \mid c_i(m) - c_i(m-1) \geq c_i(m+1) - c_i(m) \tag{1}$$

It then follows for $m_x \geq m_y > 0$

$$\begin{aligned}
c_i(m_x+1) - c_i(m_x) &\leq c_i(m_x) - c_i(m_x-1)\\
&\leq c_i(m_x-1) - c_i(m_x-2)\\
&\dots\\
&\leq c_i(m_y) - c_i(m_y+1)\\
&\leq c_i(m_y) - c_i(m_y-1)
\end{aligned}$$

A WCET function $c_i(m)$ that obeys Property 1, will produce a value for $c_i(m+1)$ threads which is greater than $c_i(m)$. The difference between $c_i(m+1)$ and $c_i(m)$ must be less than or equal to the difference of $c_i(m)$ and $c_i(m-1)$. As the number of threads increase, $c_i(m)$ increases at a decreasing (or stable) rate.

For the purposes of comparison and evaluation in Section 5, an upper bound on the growth of $c_i(m)$ is called the *growth factor* $\mathbb{F}_i$ of τ_i. Growth factors relate the WCET of one thread $c_i(1)$ to the WCET of an arbitrary number of threads $c_i(m)$ for $m > 0$. A growth factor $\mathbb{F}_i \in (0, 1]$, for a task τ_i, is a real number that satisfies Equation 2.

▶ **Definition 4** (Growth Factor for τ_i).

$$\forall m \mid c_i(m) \leq c_i(1) + (m-1) \cdot \mathbb{F} \cdot c_i(1) \tag{2}$$

For an $\mathbb{F}$ satisfying Equation 2, the pessimistic upper bound provides a linear function that can be rearranged to find an upper bound on the WCET of one thread in terms of m threads. The result is Equation 3, which will be used in the evaluation Section 5 when constructing task sets. Note, since $m \in \mathbb{N}$ each increase of m increases $c_i(m)$ by $\mathbb{F} \cdot c_i(1)$.

$$c_i(m) = c_i(1) + (m-1) \cdot \mathbb{F} \cdot c_i(1) \tag{3}$$

4 Non-Preemptive EDF Schedulability

Preemptive earliest deadline first (EDF) schedulability analysis for sporadic task sets has been well studied [18, 5, 15]. In the fully preemptive setting for which the algorithm is optimal, the overhead of a large number of preemptions may be a detriment to schedulability. Baruah [6] addresses this concern with an algorithm for calculating the non-preemptive chunk size q_i of each task $\tau_i \in \tau$. The non-preemptive chunk size q_i guarantees that task τ_i may execute up to q_i time units non-preemptively without introducing a deadline miss for any task in τ scheduled by preemptive EDF.

Section 4.3 introduces the non-preemptive feasibility algorithm Thread Per Job (TPJ) based upon the non-preemptive chunks algorithm from [6]. TPJ differs from the non-preemptive chunks algorithm by requiring the non-preemptive chunk size q_i of each task τ_i to be greater than or equal to its WCET: $c_i(m_i) \leq q_i$. As such, all jobs can be scheduled non-preemptively without fear of a deadline miss. To clearly convey TPJ, a description of the non-preemptive chunks algorithm and its dependencies is provided in the immediate subsection. Subsection 4.2 describes, by example, the available improvements to the non-preemptive chunks algorithm [6]. Subsection 4.4 defines and proves TPJ's optimality.

4.1 Non-Preemptive Chunks

The non-preemptive chunks algorithm depends on the demand bound function, EDF feasibility, ordering of absolute deadlines, and slack for the task set τ. Ordered absolute deadlines are given by $\{D_1, D_2, ...\}$ with $D_n < D_{n+1}$ for all n, where each task $\tau_i \in \tau$ contributes deadlines $D = k \cdot p_i + d_i$ for $k \in \mathbb{Z}^+$.

For a sporadic task τ_i the demand bound function for a task DBF(τ_i,t) is an upper bound on the amount of execution requirement generated from jobs released by τ_i over t units of time. The demand bound function is presented as Equation 4 as DBF(τ_i,t) modified from [5] to suit the task set model used in this work.

▶ **Definition 5** (Demand Bound Function for a Task τ_i and Interval t).

$$\text{DBF}(\tau_i{,}t) = \max\left(0, \left(\left\lfloor \frac{t-d_i}{p_i} \right\rfloor + 1\right) \cdot c_i(m_i)\right) \tag{4}$$

When necessary for brevity, Equation 5 will be used to represent the sum of demand of all tasks over an interval of length t.

▶ **Definition 6** (Demand Bound Function for the Task Set τ and Interval t).

$$\text{DBF}(\tau{,}t) = \sum_{\tau_i \in \tau} \text{DBF}(\tau_i{,}t) \tag{5}$$

Slack of the task set τ at deadline D_k is given by Equation 6. Intuitively, slack is the minimum time the processor will be idle over an interval. It is the difference between the demand over the interval and the length of the interval.

▶ **Definition 7** (Slack at Deadline D_k).

$$\textsc{slack}(D_k) = \min_{j \le k} \left(D_j - \sum_{\tau_i \in \tau} \textsc{dbf}(\tau_i, D_k) \right) \tag{6}$$

For EDF, feasibility is determined by examining increasing time intervals and calculating the demand and supply. If demand exceeds supply, the system is infeasible. Equation 7 provides a formal definition of feasibility for the task set τ.

▶ **Definition 8** (EDF Feasibility Demand Bound Test).

$$\forall t \ge 0, \left(\sum_{\tau_i \in \tau} \textsc{dbf}(\tau_i, t) \right) \le t \tag{7}$$

In [6], the number of time instants tested by Equation 7 is limited to the values of the ordered set of absolute deadlines $\{D_1, D_2, ...\}$. The ordered set of absolute deadlines is an infinite set, impractical for feasibility test. There is an upper bound on the value of all time instants (absolute deadlines) that must be tested and is denoted $T^*(\tau)$. Taken from [15], $T^*(\tau)$ is given by Equation 8 below. Among all tasks the largest deadline is $d_{\max} = \max_{\tau_j \in \tau}(d_j)$. Utilization of τ_j is defined as $U_j = \frac{c_j(m_j)}{p_j}$. Among all tasks, the greatest difference of period and deadline is given by $\Delta_{max} = \max_{\tau_i \in \tau}(p_i - d_i)$. The hyper-period of all tasks (the least common multiple of all relative deadlines) is given by P.

▶ **Definition 9** (Feasibility Test Bound t for τ).

$$T^*(\tau) = \min \left(P, \max \left(d_{max}, \frac{1}{1-U} \cdot \Delta_{max} \cdot \sum_{i=0}^{n-1} U_i \right) \right) \tag{8}$$

The non-preemptive chunks algorithm from [6] is presented (with additional details) as pseudocode in Algorithm 1 and named NP-CHUNKS. In addition to determining if the task set is schedulable under EDF, the algorithm produces a non-preemptive chunk size q_j for each task $\tau_j \in \tau$. Jobs of τ_j may execute up to q_j time units non-preemptively without negatively impacting schedulability. This setting, where a task τ_j may execute non-preemptively for some period of time q_j is referred to as *limited-preemption.*

Algorithm 1 Non-Preemptive Chunks (NP-CHUNKS).

```
1: $\textsc{slack}(D_1) \leftarrow D_1 - \sum_{\tau_i \in \tau} \textsc{dbf}(\tau_i, D_1)$
2: for $\tau_j \in \{\tau_i \in \tau \mid (d_i = D_1)\}$ do
3:     $q_j \leftarrow c_j(m_j)$
4: end for
5: for $k \in \{D_2, D_3, ...,\}$ do
6:     if $D_k > T^*(\tau)$ then
7:         return feasible
8:     end if
9:     $\textsc{slack}(D_k) \leftarrow \min(\textsc{slack}(D_{k-1}), D_k - \sum_{\tau_i \in \tau} \textsc{dbf}(\tau_i, D_k))$
10:    if $\textsc{slack}(D_k) < 0$ then
11:        return infeasible
12:    end if
13:    for $\tau_j \in \{\tau_i \in \tau \mid (d_i = D_k)\}$ do
14:        $q_j \leftarrow \textsc{slack}(D_k)$
15:    end for
16: end for
```

For a detailed description of NP-CHUNKS refer to [6]. To summarize, NP-CHUNKS begins by seeding the slack of the smallest interval D_1 and the non-preemptive chunk size of tasks with the smallest relative deadline equal to their WCET. During each iteration of $D_k \in \{D_2, D_3, ..., \}$, the slack for the interval D_k is calculated as the minimum of the current slack and the previous slack value. If there is less than zero slack, the system is infeasible. If the slack is zero or greater, each task with relative deadline equal to the current interval size is assigned the available slack as the task's non-preemptive chunk size. A task τ_j is assigned a non-preemptive chunk once, before assignment $q_j = \varnothing$ afterwards $q_j \neq \varnothing$. If the interval being examined D_k exceeds $T^*(\tau)$, the task set must be schedulable.

4.2 Improving the Non-Preemptive Chunk Size

From the description of NP-CHUNKS in [6], there is an opportunity to improve the available slack for each of the k deadlines considered. Alogrithm 1 is pessimistic in the amount of available slack at any deadline D_k. To illustrate, consider the task set and intermediate values described by Table 1.

Table 1 Example Task Set $\tau = \{\tau_0, \tau_1, \tau_2\}$.

i	p_i	d_i	m_i	$c_i(m_i)$
τ_0	4	2	1	1
τ_1	3	3	1	1
τ_2	3	3	1	1

P	D_k	$\tau_j : d_j = D_k$	DBF(τ,D_i)	SLACK(D_i)	q_j
12	$D_1 = 2$	τ_0	1	1	1
	$D_2 = 3$	τ_1	3	0	**0**
		τ_2	3	0	**0**

There are three tasks in the task set of Table 1, with utilization of approximately 0.92. For τ_0, initialization assigns a non-preemptive chunk of $q_0 = 1$ time units. By observation, after release τ_0 may be delayed from execution by at most one time unit or it will miss its deadline. Consequently, the non-preemptive chunk size available to τ_1 and τ_2 is 1. As such NP-CHUNKS would be expected to find $q_0 = 1, q_1 = 1, q_2 = 1$.

Note, it is not possible for τ_0 to be blocked for 1 or more time units if both τ_1 and τ_2 execute non-preemptively for 1 time unit each. If τ_0 is blocked for less than 1 time unit by τ_1, then τ_0 will be the highest priority task when τ_1 completes (similarly for τ_2). It is impossible for τ_0 to be blocked 1 time unit or more by τ_1 or τ_2, τ_0 would have to be released at the same time instant as τ_1 or τ_2 and τ_1 or τ_2 would have to execute before τ_0, since the relative deadline of τ_0 is less than the other two, limited-preemption EDF executes τ_0: the task with earliest absolute deadline.

For τ_0, q_0 is calculated as expected $q_0 = c_0(m_0) = 1$, by Lines 2-4 of Algorithm 1. However, τ_1 has a non-preemptive chunk size of $q_1 = 0$. The reason is Line 9, where SLACK(D_2) is calculated which includes the execution demand of τ_1 and τ_2. Slack is an upper bound on the non-preemptive chunk size assigned to a task (in this case τ_1). Giving a task the available slack permits the task to execute longer, delaying higher priority jobs from executing in the interval by delaying them for as much time as there is slack.

By example in Table 1, the available slack for τ_1 is determined from the interval of length $D_2 = 3$. The execution requirement of τ_1 and τ_2 is included in DBF(τ,3) because $d_1 = d_2 = 3$. Thus SLACK(D_2) is zero. Since τ_1's execution requirement is already included, it cannot further interfere over the interval D_2. Furthermore, τ_1 must have executed some portion without being preempted or the system would not be schedulable. Inclusion of τ_1's execution requirement within the interval over which slack is calculated for is pessimistic with respect to the non-preemptive chunk q_1 in this specific example, and q_j in general.

In the pseudocode implementation of NP-CHUNKS adopted from [6], Line 9 calculates the non-preemptive chunk size according Equation 9 (Equation 7 of Theorem 1 in [6]).

Comparing Line 9 of Algorithm 1 to Equation 9 a mismatch between the algorithm and the infeasibility test is illuminated.

▶ **Definition 10** (Infeasibility Test, Equation 7, from [6]).

$$\exists \tau_j \in \tau, t \in [0, d_j) \mid t < q_j + \sum_{\substack{i=0 \\ i \neq j}}^{n-1} \text{DBF}(\tau_i, t) \tag{9}$$

If the condition of Equation 9 is satisfied for a task set τ, the task set is unschedulable given a limited-preemption task set with assigned non-preemptive chunks q. The interval considered in the demand of Equation 9 is over $[0, d_j)$. The demand used in Algorithm 1 to calculate q_j is over the interval $[0, d_j]$. Extending the interval to include d_j introduces the pessimism identified by the example and is not required by Equation 9.

Table 1 illustrates the pessimism of NP-CHUNKS found in [6]. The example uses the notation of assigning non-preemptive chunks to individual tasks from [6]. A later work [7] uses a different notation, assigning non-preemptive chunks to interval lengths for the remaining execution of a job. The conceptual pessimism of including demand for tasks with deadline equal to the current interval (described by Table 1) is also found in [7].

4.3 Threads per Job (TPJ) Scheduling Algorithm

In this work, the NP-CHUNKS algorithm is modified for several purposes. First, the unnecessary pessimism is removed from chunk calculations. Second, the schedulability test is adapted to the model used herein. Lastly, when a given assignment of tasks and threads are infeasible, tasks are divided (when possible) to fit into their chunks. The division process is repeated until the task set is feasible, or no possible divisions remain and the task set is reported as infeasible. The algorithm is named the *Threads Per Job* (TPJ) scheduling algorithm.

A full description of TPJ is presented at the end of this subsection. To reach the complete description, an intermediate algorithm named *Bigger Non-Preemptive Chunks* (BNC) is presented as pseudocode in Algorithm 2. BNC removes the pessimism described in Section 4.2. The algorithm takes advantage of a property of the demand function $\text{DBF}(\tau, t)$ noted in [6].

▶ **Property 2** (Demand Change). *Demand for a task does not change for values of t that do not equal an absolute deadline. In terms of the set of ordered absolute deadlines,* $\text{DBF}(\tau, D_{i-1}) = \text{DBF}(\tau, D_i - \epsilon)$, *for* $0 < \epsilon \leq (D_i - D_{i-1})$.

Algorithm 2 Bigger Non-Preemptive Chunks (BNC).

```
 1: $\text{SLACK}(D_0) \leftarrow \infty$
 2: for $k \in \{D_1, D_2, D_3, ...,\}$ do
 3:     if $D_k > T^*(\tau)$ then
 4:         return feasible
 5:     end if
 6:     $\text{SLACK}(D_k) \leftarrow \min\left(\text{SLACK}(D_{k-1}), D_k - \sum_{\tau_i \in \tau} \text{DBF}(\tau_i, D_k)\right)$
 7:     if $\text{SLACK}(D_k) < 0$ then
 8:         return infeasible
 9:     end if
10:     for $\tau_j \in \{\tau_i \in \tau \mid (d_i = D_k)\}$ do
11:         $q_j \leftarrow \min(c_j(m_j), \text{SLACK}(D_{k-1}))$
12:     end for
13: end for
```

Line 11 of Algorithm 2 implements the improvement of BNC over NP-CHUNKS. The non-preemptive chunk q_j of task τ_j is taken from the slack of the previous interval D_{k-1} or the task's WCET $c_j(m_j)$, whichever is smaller. The algorithm verifies the condition set by Equation 9, selecting the correct interval length by Property 2, which precludes the inclusion of τ_j's execution requirement in the interval (and other tasks with deadline D_k).

Algorithm 3 Threads-Per-Job (TPJ).

```
1:  SLACK($D_0$) $\leftarrow \infty$
2:  for $k \in \{D_1, D_2, D_3, ..., \}$ do
3:      if $D_k > T^*(\tau)$ then
4:          return *feasible*
5:      end if
6:      for $\hat{\tau}_j \in \{\tau_i \in \tau \mid (d_i = D_k)\}$ do
7:          if SLACK($D_{k-1}$) $< \hat{c}_j(1)$ then
8:              return *infeasible*
9:          end if
10:         $\Phi_j \leftarrow \{\hat{\tau}_j\}$
11:         if SLACK($D_{k-1}$) $< \hat{c}_j(\hat{m}_j)$ then          ▷ Jobs must be divided
12:             $\Phi_j \leftarrow$ DIVIDE($\hat{\tau}_j$,SLACK($D_{k-1}$))
13:             $\tau \leftarrow \tau \setminus \hat{\tau}_j$          ▷ Anterior task $\hat{\tau}_j$ is represented by $\Phi_j$
14:             $\tau \leftarrow \tau \cup \Phi_j$          ▷ Partial tasks include all threads of $\hat{\tau}_j$
15:         end if
16:         for $\tau_j \in \Phi_j$ do
17:             $q_j \leftarrow c_j(m_j)$
18:         end for
19:     end for
20:     SLACK($D_k$) $\leftarrow \min\left(\text{SLACK}(D_{k-1}), D_k - \sum_{\tau_i \in \tau} \text{DBF}(\tau_i, D_k)\right)$
21:     if SLACK($D_k$) $< 0$ then
22:         return *infeasible*
23:     end if
24: end for
```

The Threads per Job scheduling Algorithm 3, is a modification of BNC from limited-preemption EDF (EDF-LP) scheduling to non-preemptive EDF (EDF-NP). Input to the schedulability test is a task set specification τ, if TPJ returns a *feasibile* result there exists a posterior task set which can be scheduled by non-preemptive EDF and the posterior task set is returned as τ. An *infeasible* result indicates that TPJ could not guarantee τ would be schedulable by EDF-NP for any posterior task set. Since non-preemptive EDF is not optimal with respect to feasibility [12], TPJ is a sufficient test but cannot be necessary.

Algorithm 3 (TPJ) modifies BNC, the modifications are limited to Lines 6-19. An additional benefit of BNC removing the pessimism of each q_j, is that each q_j can be calculated without consideration of the current task τ_j and the demand at D_k. Chunk values depend on the demand of D_{k-1} instead. This permits an efficient implementation of TPJ by moving the slack calculation of the current interval to the end of each iteration. Otherwise, if slack were calculated earlier in each iteration, the changes to demand resulting from Lines 6-19 would force the demand and slack of D_k to be recalculated.

The first notable change to BNC is introduced on Line 7, comparing the available slack to the WCET of a single thread of $\hat{\tau}_j$. If there is insufficient slack to execute just one thread of $\hat{\tau}_j$ to completion, the task cannot be executed non-preemptively for any number of threads and the task set is infeasible non-preemptively.

Lines 11-15 introduce several subtle changes. For clarity, it is simpler to discuss the negative case ($\text{SLACK}(D_{k-1}) \geq \hat{c}_j(\hat{m}_j)$) before the positive. When there is sufficient slack for $\hat{m}_j$ threads to execute without preemption, $\hat{\tau}_j$ is given its full WCET ($\hat{c}_j(\hat{m}_j)$) as its non-preemptive chunk. In other words, no division of $\hat{\tau}_j$ is required and the posterior task set τ is unchanged (with respect to $\hat{\tau}_j$). Lines 11-15 are avoided, the algorithm progresses to the next task such that $d_i = D_k$.

However, in the positive case on Line 11 (when $\text{SLACK}(D_{k-1}) < \hat{c}_j(\hat{m}_j)$), $\hat{m}_j$ threads of $\hat{\tau}_j$ cannot feasibly execute without being preempted. Therefore, $\hat{\tau}_j$ must be divided. The DIVIDE procedure creates a partial task Φ_j set of $\hat{\tau}_j$, such that all tasks $\tau_p \in \Phi_j$ will complete within the available slack $c_p(m_p) \leq D_{k-1}$. The posterior task set τ has $\hat{\tau}_j$ removed, and is replaced by the partial set Φ_j maintaining the specified number of threads for $\hat{\tau}_j$.

For any task $\hat{\tau}_j$, the task is transformed into a partial task set Φ_j and assigned a non-preemptive chunk only once in the iteration where the absolute deadline D_k is equal to the relative deadline of the task: $D_k = \hat{d}_j$. Since tasks of τ are evaluated in strictly increasing absolute deadline order, the impact on demand and non-preemptive chunk sizes of processing $\hat{\tau}_j$ exclusively impacts demand for larger intervals $D_\ell > D_k$ and non-preemptive chunk sizes for tasks $\tau_\ell \in \tau$ with greater relative deadlines $d_\ell > \hat{d}_k$.

▶ **Property 3** (Divisions of $\hat{\tau}_j$ Exclusively Impacts Interval of Length $t \geq \hat{d}_j$). *Division of $\hat{\tau}_j$ into the partial set Φ_j, and replacing $\hat{\tau}_j$ in τ with Φ_j will impact demand exclusively for intervals of length $D_k \geq \hat{d}_j$, slack of absolute deadlines $D_k > \hat{d}_j$ and therefore non-preemptive chunk values q_ℓ for tasks $\tau_\ell \in \tau$ with relative deadlines $d_\ell \geq D_k$*

By definition of $\text{DBF}(\hat{\tau}_j,t)$, no task of Φ_j or $\hat{\tau}_j$ can impact the task set τ demand $\text{DBF}(\tau,t)$ when $t < d_j$. Thus replacing $\hat{\tau}_j$ in τ, only affects the demand of intervals with length $\hat{d}_j$ or greater. Slack over the interval D_k is calculated from exclusively shorter intervals. Since the demand of the current interval D_k does not influence the slack at D_k, replacing $\hat{\tau}_j$ in τ only affects the slack of intervals with length greater than D_k. Non-preemptive chunk sizes are assigned based on the available slack, and only those assigned for an interval of length greater than D_k can be affected by replacing $\hat{\tau}_j$ in τ.

Algorithm 4 DIVIDE.

```
1: procedure DIVIDE($\hat{\tau}_j$, $q$)
2:     $\Phi_j \leftarrow \{\}$
3:     $m \leftarrow \underset{m \in \mathbb{Z}^+}{\text{argmax}} (\hat{c}_i(m) \leq q)$
4:     $r \leftarrow \hat{m}_j$
5:     while $r > 0$ do
6:         $m_p \leftarrow \min(r, m)$
7:         $\tau_p \leftarrow (\hat{p}_j, \hat{d}_j, m_p, \hat{c}_j)$      ▷ Posterior task, same period, deadline, WCET function.
8:         $\Phi_j \leftarrow \Phi_j \cup \tau_p$
9:         $r \leftarrow r - m_p$
10:    end while
11:    return $\Phi_j$
12: end procedure
```

On Line 12 of the TPJ Algorithm 3, the task $\hat{\tau}_j$ is divided into Φ_j by the DIVIDE procedure. Pseudocode of DIVIDE is given by Algorithm 4. The number of tasks in Φ_j are determined by the maximum number of threads m of $\hat{\tau}_j$ that can execute non-preemptively within q time units. Each task $\tau_k \in \Phi_j$ is assigned m threads of $\hat{\tau}_j$ or however many remain, whichever is less. The result is that each task set has the following properties.

▶ **Property 4** (Partial Task Sets Returned from DIVIDE). *The partial task set Φ_j of an anterior task $\hat{\tau}$ for a specific q value (and related maximum threads assigned per job m such that $c_j(m) \leq q$) contains posterior tasks where:*

1. *The exact number of posterior tasks is $|\Phi_j| = \lceil \frac{\hat{m}_j}{m} \rceil$*
2. *Exactly $\lfloor \frac{\hat{m}_j}{m} \rfloor$ tasks of Φ_j are assigned m threads per job.*
3. *There is at most one task $\tau_g \in \Phi_j$ with exactly $m_g = \hat{m}_j \mod m$ threads.*

4.4 Non-Preemptive Feasibility of TPJ and DIVIDE

The DIVIDE Algorithm 4 creates a partial task set Φ_j for an anterior task $\hat{\tau}_j$, assigning as many threads to each task in Φ_j as possible. Upon returning Φ_j to TPJ, $\hat{\tau}_j$ is replaced in the task set τ. Algorithm 4 is one method of dividing of $\hat{\tau}_j$ which TPJ could employ when creating the posterior task set τ. This section justifies DIVIDE's method by demonstrating the effect on schedulability and optimality of TPJ.

This section's ultimate objective is to clearly convey Theorem 5; concluding that TPJ is optimal with respect to task-level non-preemptive multi-threaded feasibility. The theorems that precede Theorem 5 establish minimal demand and WCET sums for partial sets created by DIVIDE necessary to illustrate TPJ's optimality.

Non-preemptive EDF scheduling of jobs of multiple threads ordered by a thread-level scheduler (such as `BUNDLE` or `BUNDLEP`) allows preemptions between threads of the same job but precludes preemptions between jobs. Each task benefits from the advantages of thread-level scheduling by the exclusive use of the processor and shared resources. Since task set specifications may be divided, a specification is feasible when threads of the specification $\hat{\tau}$ may be assigned to tasks such that the posterior task set τ is feasible by EDF-NP.

▶ **Definition 11** (npm-feasible). *A task set specification $\hat{\tau}$ is task-level non-preemptive multi-threaded feasible (*npm-feasible*) if there exists a posterior task set τ of $\hat{\tau}$ such that all multi-threaded jobs scheduled by EDF-NP will always meet their deadlines.*

For the theorems that follow, unless necessary to discriminate between anterior and posterior tasks, the anterior task $\hat{\tau}_i$ will be written τ_i. The sum of the demand of the partial tasks of τ_i for an interval of length t is $\sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k,t)$.

▶ **Theorem 1** (Minimal Demand of Partial Task Sets Over All Intervals). *For a partial task set Φ_i of an anterior task τ_i with m_i threads, minimizing $\sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k,d_i)$ minimizes $\sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k,t)$ for all $t \geq 0$.*

Proof. Provided into two parts, when $t < d_i$ and $t \geq d_i$. The first portion is a simple direct argument. The second portion is by contradiction.

Part 1: When $t < d_i$, $0 = \sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k,t)$. By definition of the demand bound function (Equation 4) the execution requirement of a task is zero before the first possible deadline. All tasks $\tau_k \in \Phi_i$ share the same relative deadlines $d_k = d_i$ and absolute deadlines because $p_k = p_i$. These follow from the definition of division (Definition 1) and partial tasks (Definition 2). Since $t < d_i$, $\text{DBF}(\tau_k,t) = 0$ for all $\tau_k \in \Phi_i$. Therefore, $\sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k,t)$ will be minimal (exactly zero) when $t < d_i$, regardless of $\sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k,d_i)$.

Part 2: When $t \geq d_i$, assume $\sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k,d_i)$ is minimal and $\sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k,t)$ is not minimal. Since all partial tasks $\tau_k \in \Phi_i$ share absolute deadlines (as described in Part 1), demand for each task $\text{DBF}(\tau_k,t)$ increases only for values of t that equal absolute deadlines. Furthermore, the execution requirement of every τ_k increases exactly by $c_k(m_k)$ for each absolute deadline of $\tau_i = \{D_1, D_2, ...\}$:

$$\begin{aligned}
\textsc{dbf}(\tau_k,D_1) &= 1 \cdot c_k(m_k)\\
\textsc{dbf}(\tau_k,D_2) &= 2 \cdot c_k(m_k)\\
&...\\
\textsc{dbf}(\tau_k,D_z) &= z \cdot c_k(m_k)
\end{aligned}$$

Utilizing Property 2, for $t \geq d_i$ and D_z, where D_z is the greatest absolute deadline of τ_i less than or equal to t ($D_z \leq t$):

$$\sum_{\tau_k \in \Phi_i} \textsc{dbf}(\tau_k,t) = \sum_{\tau_k \in \Phi_i} z \cdot \textsc{dbf}(\tau_k,d_i) \quad = z \cdot \sum_{\tau_k \in \Phi_i} \textsc{dbf}(\tau_k,d_i)$$

Because z depends on t (and is completely independent of the division of the partial task set), if $\sum_{\tau_k \in \Phi_i} \textsc{dbf}(\tau_k,t)$ were not minimal then $\sum_{\tau_k \in \Phi_i} \textsc{dbf}(\tau_k,d_i)$ could not be minimal, contradicting the assumption.

Combining Parts 1 and 2, when the demand for the partial tasks of τ_i is minimized for the interval d_i, the demand of partial tasks of τ_i is minimized for all intervals of length $t \geq 0$. ◀

▶ **Corollary 2** (Minimal WCET Sum of Φ_i Minimizes Demand Over the Interval d_i). *The demand of Φ_i over the interval d_i is minimized when the sum of WCET of Φ_i is minimized.*

Proof. Following directly from Theorem 1, where the demand over the interval d_i of each task $\tau_k \in \Phi_i$ is given by $\textsc{dbf}(\tau_k,d_i) = 1 \cdot c_k(m_k) = c_k(m_k)$. Then,

$$\sum_{\tau_k \in \Phi_i} \textsc{dbf}(\tau_k,d_i) = \sum_{\tau_k \in \Phi_i} c_k(m_k)$$

Thus, minimizing $\sum_{\tau_k \in \Phi_i} c_k(m_k)$ minimizes $\sum_{\tau_k \in \Phi_i} \textsc{dbf}(\tau_k,d_i)$ ◀

▶ **Corollary 3** (Minimal WCET Sum of Φ_i Minimizes Demand Over all Intervals $t \geq 0$). *The demand of Φ_i over alls interval $t \geq 0$ is minimized when the sum of WCET of Φ_i is minimized.*

Proof. Following directly from Theorem 1 and Corollary 2. ◀

▶ **Definition 12** (Assumptions of Theorem 4). *For the following theorem, there are several assumptions that must be upheld for the result to be valid. These assumptions are consequences of the non-preemptive setting and requirements of the task set specification.*

1. *All tasks τ_i must be characterized by strictly increasing discrete concave WCET function $c_i(m_i)$.*
2. *Any task $\tau_i \in \tau$ where $c_i(m_i) > q_i$ is not schedulable non-preemptively. Consequently, no assignment of m_i may cause $c_i(m_i) > q_i$ or the task set is infeasible.*
3. *The greatest number of threads assigned to a task τ_i such that $c_i(m_i) \leq q_i$ is named $m = \underset{m \in \mathbb{Z}^+}{\text{argmax}}\,(c_i(m) \leq q_i)$.*

▶ **Theorem 4** (Minimal Sum of WCET of Φ_i for any q by DIVIDE). *For an anterior task $\hat{\tau}_i$ and non-preemptive chunk size q,* DIVIDE *will produce a partial task set Φ_i with minimum WCET sum among all possible partial task sets of $\hat{\tau}_i$.*

Proof. To illustrate a contradiction, assume Φ_i returned from DIVIDE does not have the minimal WCET sum for a specific q and task $\hat{\tau}_i$. There must exist a partial task set Φ_k of $\hat{\tau}_i$ that differs, ie. $\Phi_i \neq \Phi_k$ and

$$\sum_{\tau_k \in \Phi_k} c_k(m_k) < \sum_{\tau_j \in \Phi_i} c_j(m_j)$$

By Property 4 of partial tasks created by DIVIDE, Φ_i will have at most one task with less than m threads assigned to it. For Φ_k to differ, it must have at least two tasks with less than m threads assigned to them. Call these two tasks with less than m threads $\tau_x, \tau_y \in \Phi_k$. Select τ_x as the task with the greater number of threads $m_x \geq m_y$.

Consider the impact on $\sum_{\tau_k \in \Phi_k} c_k(m_k)$ of moving one thread of τ_y to τ_x, as the operation of adding the difference of WCET values for $c_x(m_x+1)$ and $c_y(m_y-1)$ to the sum.

$$\begin{aligned}&\left(\sum_{\tau_k \in \Phi_k} c_k(m_k)\right) - c_x(m_x) + c_x(m_x+1) - c_y(m_y) + c_y(m_y-1)\\ =\ &\left(\sum_{\tau_k \in \Phi_k} c_k(m_k)\right) + (c_x(m_x+1) - c_x(m_x)) - (c_y(m_y) - c_y(m_y-1))\end{aligned}$$

By the concave growth Property 1 and virtue of $m_y \leq m_x$, the quantity $(c_x(m_x+1) - c_x(m_x))$ is less than or equal to $(c_y(m_y) - c_y(m_y-1))$ so the difference must be less than or equal to zero. Therefore:

$$\left(\sum_{\tau_k \in \Phi_k} c_k(m_k)\right) + (c_x(m_x+1) - c_x(m_x)) - (c_y(m_y) - c_y(m_y-1)) \leq \sum_{\tau_k \in \Phi_k} c_k(m_k)$$

The WCET sum of Φ_k can be reduced by moving one thread of τ_y to τ_x. When $m_x = m$ no more threads may be assigned to τ_x or the system will be infeasible by Definition 12. While there are two (or more) tasks of $\tau_x, \tau_y \in \Phi_k$ with fewer than m threads assigned, moving one thread from τ_y to τ_x will reduce the WCET sum. By repeatedly moving tasks to reduce the WCET sum, Φ_k will satisfy all aspects of Property 4 of partial task sets created by DIVIDE, ie. $\Phi_i = \Phi_k$ after all moves have been completed. This contradicts the assumption that $\Phi_i \neq \Phi_k$ and the relationship of their WCET sums, therefor Φ_i is minimal. ◀

▶ **Theorem 5** (TPJ is Optimal with Respect to npm-feasibility). *For a task set specification $\hat{\tau}$, TPJ returns feasible if and only if there exists an npm-feasible posterior task set τ of $\hat{\tau}$.*

Proof. *Forward Direction* (TPJ returns feasible for $\hat{\tau} \implies \exists$ a posterior task set $\tau \mid \tau$ is npm-feasible): The TPJ algorithm returned a posterior task set τ where the infeasibility condition (Equation 9) is never satisfied across intervals of length $0 \leq t \leq T^*(\tau)$ and every job of $\tau_i \in \tau$ executes non-preemptively for $c_i(m_i) \leq q_i$ time units. Therefore, τ is npm-feasible.

Reverse Direction ($\exists$ a posterior task set $\tau \mid \tau$ is npm-feasible $\implies$ TPJ returns feasible for $\hat{\tau}$): For the purpose of demonstrating a contradiction, assume TPJ returns infeasible for an npm-feasible task set $\hat{\tau}$. Name the absolute deadline which TPJ returned infeasibility for D_x from the set ordered deadlines $\{D_1, D_2, ...\}$ and the task which generated D_x, $\hat{\tau}_x$. Name the set of tasks with relative deadlines smaller than $\hat{d}_x$, $\bar{\tau}$.

For any task $\tau_k \in \bar{\tau}$ and partial task set Φ_k of τ_k included in the posterior set τ, the number of tasks and threads assigned to each Φ_k cannot be affected by $\hat{\tau}_x$ due to $\hat{d}_x > d_k$ and Property 3. The combined set of posterior tasks of $\bar{\tau}$ in τ is denoted $\dot{\tau} = \cup_{\tau_k \in \bar{\tau}} \Phi_k$.

There are two cases where TPJ will return infeasible for $\hat{\tau}$, on Line 8 and Line 22. Both illustrate a contradiction with the respect to demand.

Line 8: If TPJ returns infeasible for $\hat{\tau}$ on Line 8 there is insufficient slack q_x to execute any one-thread job of $\hat{\tau}_x$ non-preemptively. Since slack is inversely related to demand, the demand of $\dot{\tau}$ is too great to allow any thread of τ_x as part of a feasible task set.

Line 22: If TPJ returns infeasible for $\hat{\tau}$ on Line 22, there is insufficient supply for Φ_x (the set of partial tasks of $\hat{\tau}_x$). By Corollary 2 and Theorem 4 the demand of Φ_x is minimal over all intervals for the available slack q_x. Due to Property 3 only tasks with shorter relative deadlines i.e. $\dot{\tau}$, can impact the demand of Φ_x by affecting q_x. In this case, the demand of $\dot{\tau}$ is too great for the demand of Φ_x to be included as part of a feasible task set.

By assumption $\hat{\tau}$ is npm-feasible, the infeasibility conditions on Lines 8 and 22 of TPJ indicate the demand of $\dot{\tau}$ is too great. However, TPJ adds each partial set Φ_k to $\dot{\tau}$ in increasing deadline order. By Property 3, every Φ_k added to $\dot{\tau}$ exclusively impacts the demand of larger deadlines. Every Φ_k increases the demand of $\dot{\tau}$ minimally starting with D_1, maximizing the slack available for partial task sets with greater deadlines; thus the demand of $\dot{\tau}$ is minimal and cannot be reduced. For $\hat{\tau}$ to be npm-feasible, there must be another partial task set that reduces $\dot{\tau}$'s demand, which is a direct contradiction. Therefore, TPJ must return feasible. ◀

5 Evaluation

Evaluation [28] of TPJ and the non-preemptive multi-threaded task model presented in this work focuses on the schedulability ratio of synthetic task sets and a case study based upon the evaluation of `BUNDLEP` [29]. The ratio of task set specifications deemed schedulable by TPJ for EDF-NP will be compared to NP-CHUNKS in both limited and fully preemptive settings for EDF. What follows is a description of the parameters to task set specification generation, the prescribed evaluation metrics, and analysis of the results.

5.1 Generating Task Sets

A specified task set τ is generated with four parameters, M the total number of threads of execution, U the target utilization, a maximum growth factor $\mathbb{F}$, and m the maximum number of threads per task. The number of threads M may be one of $\{3, 5, 7, 10, 25, 50, 100\}$ with dependent m values of $\{2, 2, 3, 4, 8, 16, 32\}$. Utilization varies from [0.1, 0.9] and the growth factor varies from [0.1, 0.9] independently by increments of 0.1.

Each task $\tau_i \in \tau$ is assigned m_i threads from a random uniform integer distribution over $[1, m]$, such that the sum of all threads is equal to $M = \sum_{\tau_i \in \tau} m_i$. A task's period p_i is from a uniform integer distribution over [10, 1000]. Utilization u_i of each task τ_i is calculated using the UUniFast(n, U) [9] algorithm, where $n = |\tau|$.

A task's WCET is assigned for m_i threads, $c_i(m_i) = \lceil p_i \cdot U_i \rceil$. Tasks are given a growth factor $\mathbb{F}_i$ in a uniform real distribution over $[0.1, \mathbb{F}]$. The remaining $m_i - 1$ WCET values are determined by substituting $\mathbb{F}_i$ into Equation 3. The relative deadline of τ_i, d_i is taken from a uniform integer distribution over $[\max(c_i(m_i), p_i/2), 1000]$.

For each combination of $(M, m, U, \mathbb{F})$, 1000 task sets specifications are generated. Table 2 summarizes the parameters of task set generation. The smaller values of M are taken from [7] and the dependent m values were selected to avoid one task consuming more than half of the threads in the task set specification (where possible).

Table 2 Task Set Generation Parameters.

U	[0.1, 0.9]	M	{3,	5,	7,	10,	25,	50,	100}
$\mathbb{F}$	[0.1, 0.9]	m	{2,	2,	3,	4,	8,	16,	32}

5.1.1 Applicability of Parameters

To avoid favoring TPJ, the task set generation parameters m and $\mathbb{F}$ were carefully selected. For the threads per task m, a large m favors TPJ. Therefore, no single task my be assigned more than half the total threads: $m \leq \lfloor \frac{M}{2} \rfloor$ (except for $M = 3$).

The growth factor $\mathbb{F}$ is informed by previous results for BUNDLEP [29]. In [29], multi-threaded tasks are constructed from the Mälardalen WCET benchmarks [16]. Task analysis in [29] yields growth factors below 0.1 for several benchmarks. A lower bound (0.1) on $\mathbb{F}$ greater than observed values is pessimistic, resulting in less favorable results for TPJ.

5.2 Case Study

BUNDLEP's evaluation covers 18 benchmarks for distinct architecture configurations. An architecture configuration includes the block reload time (BRT), cycles per instruction (CPI), and number of cache lines. One of the least favorable in terms of the analytical benefit of BUNDLE is a BRT of 100, CPI of one, and 32 cache lines. From this configuration, the WCET values and growth factors were extracted, growth factors ranging in the range $[0.08, 3.02]$.[1]

From these results of BUNDLEP 1000 task sets with 18 tasks (one per benchmark) and a total 100 threads were generated per utilization target. The utilization target ranged from 0.1 to 1.0 increments of 0.1. Threads were assigned to each task τ_i from a distribution over $m_i \in [2, 8]$. Each tasks utilization, period, and deadline, $c_i(m_i)$ were assigned using the same method as synthetic tasks. The WCET values for fewer threads $1 \leq k < m_i$, were scaled such that the value of $c_i(k)/c_i(m_i)$ remained constant after the $c_i(m_i) = \lceil p_i \cdot U_i \rceil$ assignment.

5.3 Evaluation Metrics

TPJ is compared with the NP-CHUNKS schedulability test in non-preemptive (EDF-NP) and preemptive (EDF-P) settings. The focus of the evaluation is on the non-preemptive setting. The preemptive setting serves as a comparison to alternative scheduling strategies and the theoretical best case. For EDF-P, preemptions incur **no** penalty, CRPD or otherwise. In this highly advantageous setting for EDF-P, TPJ can still produce feasible non-preemptive task sets NP-CHUNKS deems infeasible in a preemptive setting!

To compare schedulability tests, each task set specification $\hat{\tau}$ is provided to TPJ without modification under EDF-NP scheduling. TPJ will transform the task set producing a posterior task set τ if a feasible one exists. A task set specification $\hat{\tau}$ cannot be provided directly to NP-CHUNKS, since NP-CHUNKS has no concept of threads per job.

To be suitable for analysis by NP-CHUNKS, a task set specification $\hat{\tau}$ is transformed into two posterior task sets. The first task set, τ^1 represents single-threaded tasks by including all threads of $\hat{\tau}$ as individual tasks. The second task set, τ^m represents the tasks of $\hat{\tau}$ as indivisible, executing all specified threads without preemption per job. Each task in τ^m benefits from the thread-level scheduler but does not expose the threaded nature of the task to the scheduling algorithm. This is achieved by modifying an anterior task $\hat{\tau}_j$ with $\hat{m}_j > 1$ and $\hat{c}_j(\hat{m}_j)$ to a posterior task τ_j with $m_j = 1$ and $c_j(1) = \hat{c}_j(\hat{m}_j)$.

The NP-CHUNKS schedulability test will produce results for τ^1 and τ^m in both preemptive and non-preemptive settings. For non-preemptive schedulability analysis, each task $\tau_i \in \tau^1$ or τ^m must have a non-preemptive chunk size $q_i \geq c_i(m_i)$. When evaluating preemptive EDF schedulability for τ^1 and τ^m, the results are labeled EDF-P:1 and EDF-P:M respectively. When evaluating non-preemptive EDF schedulability, the results are labeled EDF-NP:1 and

[1] Due to length restrictions the full listing of WECT and growth factors are omitted.

Table 3 Schedulability Test Combinations.

Test	Task Set	EDF-NP	EDF-P
TPJ	$\hat{\tau}$	EDF-TPJ	-
NP-CHUNKS	τ^1	EDF-NP:1	EDF-P:1
	τ^m	EDF-NP:M	EDF-P:M

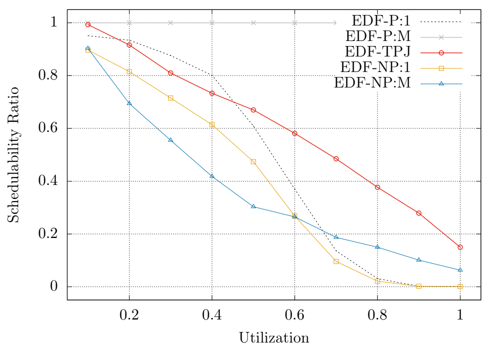
EDF-NP:M. Schedulability results for TPJ under EDF-NP scheduling are labeled EDF-TPJ. Table 3 gives a synopsis of the schedulability tests. Schedulability ratios for each of the combinations are calculated for every $(M, m, U, \mathbb{F})$ configuration.

It must be noted that EDF-P:M is an unrealistic schedulability test. It serves only as a theoretical limit to the benefits of concave growth. Concave growth is a result of scheduling threads of the same job without preemption by another job with a `BUNDLE`-based thread-level scheduler. However, current `BUNDLE` implementations require that an executing task cannot be preempted by a different task. Such a preemption would destroy the cache benefits and analysis of `BUNDLE` scheduling. Analysis of EDF-P:M assumes preemptions between jobs are allowed and have zero cost. It is included as a reference for TPJ's performance, as a ceiling for what is theoretically possible given ideal (but likely impossible) conditions.

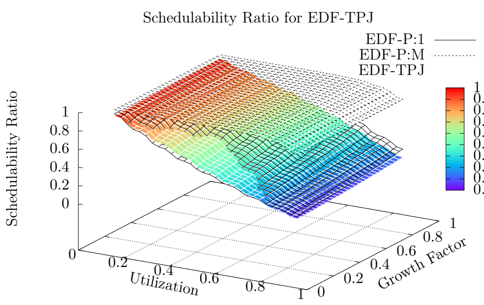
As a consequence of transforming multi-threaded task set specifications $\hat{\tau}$ to single-threaded task sets τ^1, some single threaded task sets may not be feasible. One reason for a task set τ^1 to become infeasible is the utilization exceeding one, while τ^m and $\hat{\tau}$ have utilization less than one. In this setting, EDF-TPJ is capable of scheduling task sets that preemptive EDF cannot.

For a task set specification configuration $(M, m, U, \mathbb{F})$, call S the set of all task set specifications $\hat{\tau}$ generated for the configuration. Call s the set of τ^1 task sets transformed from $\hat{\tau} \in S$ such that τ^1 has utilization greater than one. The set s^{TPJ} is the subset of s deemed feasible by the TPJ schedulability test. That is, s^{TPJ} is the set of all tasks TPJ could schedule, yet EDF-P:1 could not (even) when CRPD values are zero.

5.4 Results



(a) `BUNDLEP` Case Study.

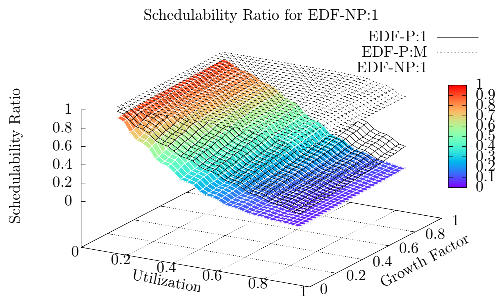


(b) EDF-TPJ Summary.

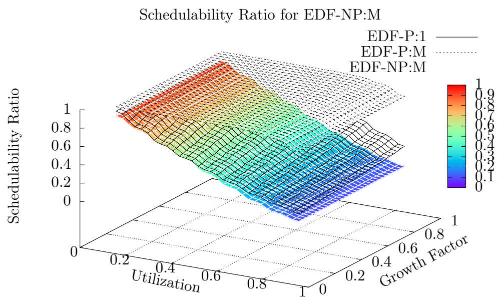
Figure 2 Case Study and EDF-TPJ Summary Results.

Schedulability ratios from the `BUNDLEP` case study are given in Figure 2a. For the target architecture and 18 benchmarks, EDF-TPJ consistently outperforms the other non-preemptive algorithms. For preemptive EDF-P:1 (with zero cost preemptions), EDF-TPJ has higher schedulability ratios for the majority of target utilization values. EDF-TPJ's comparative performance increases with the target utilization. This case study demonstrates the benefit of TPJ to non-preemptive and (potentially) preemptive approaches.

Figures 2b, 3a, and 3b, summarize the results for the synthetic task sets varied by the utilization and growth factor. Within each graph, the schedulability ratios provided by EDF-P:1 and EDF-P:M serve as references. The difference between EDF-P:1 and the subject of the graph illustrate the benefit of preemptive scheduling. Inclusion of EDF-PM highlights the theoretical limit of concave growth to schedulability.



(a) EDF-NP:1 Summary.



(b) EDF-NP:M Summary.

Figure 3 EDF-NP:1 and EDF-NP:M Summary.

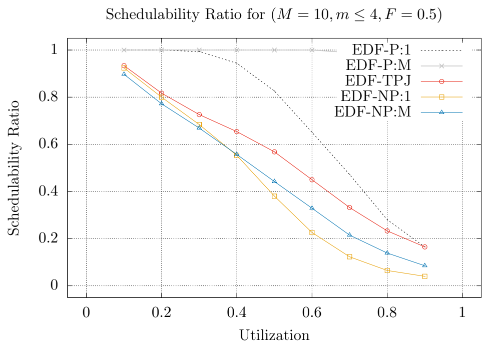
Including EDF-P:1 and EDF-P:M in each of the summary graphs eases the comparison between EDF-NP:1, EDF-NP:M, and EDF-TPJ. Comparing EDF-NP:1 (3a) to EDF-NP:M (3b), illustrates the benefits of the model and scheduling mechanism. EDF-NP:M has a consistently higher schedulability ratio for all utilizations and growth factors. EDF-TPJ (2b) outperforms EDF-NP:M, with higher schedulability ratios for all utilizations and growth factors due to the ability to transform task sets. EDF-TPJ performs best among the non-preemptive tests across all configurations. Additionally, EDF-TPJ is able to schedule task sets deemed infeasible for EDF-P:1.

Table 4 summarizes the infeasible utilization findings for the synthetic tasks. For moderate and larger values of $M(\geq 25)$, the number of infeasible by utilization task sets dominate the specifications. For 25, 50, and 100 total threads, the infeasible by utilization comprise 44, 59, and 74 percent of the task sets respectively, with EDF-TPJ finding 25, 34, and 45 percent feasible. This illustrates the large potential of the proposed model, in conjunction with concave growth WCET functions of thread-level schedulers (e.g. `BUNDLE` and `BUNDLEP`).

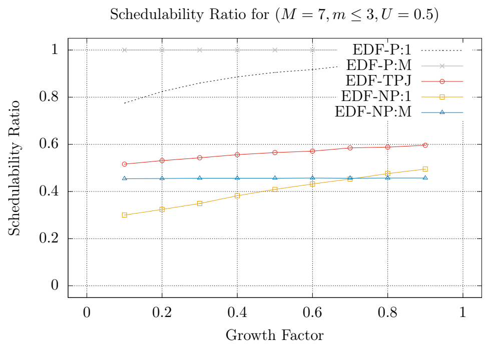
Table 4 $U > 1$ Feasibility.

(M, m)	$(3,2)$	$(5,2)$	$(7,3)$	$(10,4)$	$(25,8)$	$(50,16)$	$(100,32)$	Total
$\|S\|$	81000	81000	81000	81000	81000	81000	81000	567000
$\|s\|$	3131	4973	11744	18689	36565	49147	59412	183661
$\|s^{\text{TPJ}}\|$	465	291	1437	3065	9426	16912	25832	57428

There are two noteworthy trends within the schedulability results. The simpler of the two is the relationship between utilization and schedulability ratio for a fixed growth factor. Figure 4a illustrates the trend common among $M \leq 10$ total threads. The trend for preemptive and non-preemptive schedulability tests when utilization increases is for the schedulability ratio to decrease. However, EDF-TPJ always outperforms the other non-preemptive tests.



(a) $(M, m, U, \mathbb{F}) = (10, 4, *, 0.5)$.



(b) $(M, m, U, \mathbb{F}) = (7, 3, 0.5, *)$.

Figure 4 $M \leq 10$ Performance.

The second trend is slightly more complex. Figure 4b was selected for the smallest M and U values with visually distinct plots per schedulability test. The growth factor and the schedulability ratio are correlated. As the growth factor increases, so does the schedulability ratio. This is due to the utilization being held constant. When the growth factor is small, the WCET of the first thread of each task is larger. Larger WCET values are harder to schedule non-preemptively.

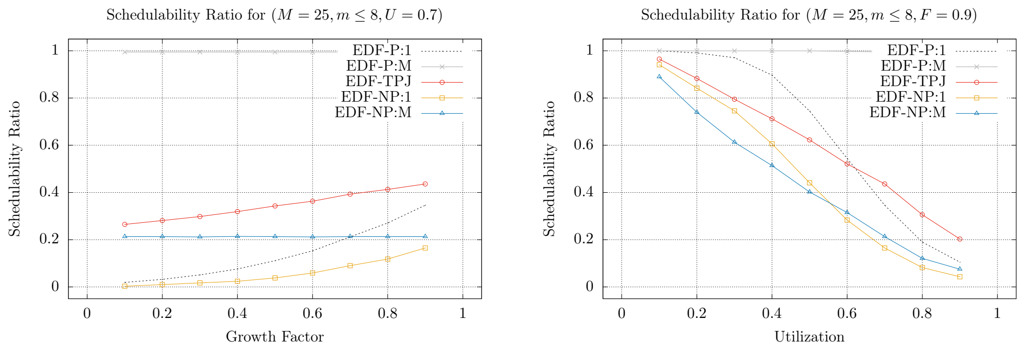


Figure 5 $M > 10$ EDF-TPJ Performance Above EDF-P:1.

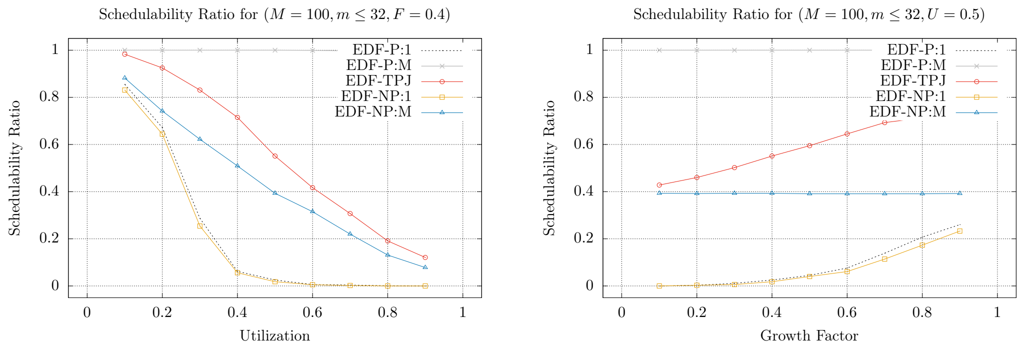


Figure 6 $M = 100$ EDF-TPJ Performance.

As M increases beyond 10 total threads, the number of infeasible by utilization task sets s grows. This contributes to the schedulability ratio of EDF-TPJ surpassing EDF-P:1 for threshold utilization and growth factor values. For $M = 25$, the threshold of utilization is between $[0.6, 0.7]$ shown in Figure 5.

For $M = 100$ and $\mathbb{F} \leq 0.4$, EDF-TPJ outperforms EDF-P:1. Figure 6 highlights the advantage of EDF-TPJ compared to EDF-P:1 by virtue of concave growth. It also highlights the benefit of dividing tasks, as the performance of EDF-NP:M is always below EDF-TPJ.

The comparative performance of EDF-TPJ is at its lowest for $M < 10$ threads and $U > .4$ utilization. In these ranges EDF-TPJ maintains the highest schedulability ratio among the non-preemptive methods, but the ratio is closer to EDF-NP:M or EDF-NP:1 than EDF-P:1. This suggests, the decrease in EDF-TPJ's performance is more likely due to the non-preemptive setting combined with larger WCET values for individual threads.

6 Conclusion

Motivation for this work stemmed from `BUNDLE`-based thread-level schedulers limitation of a single task and single job. The primary goal was to create a multi-task scheduling technique and schedulability test for those `BUNDLE`-based thread-level schedulers which leverages without decreasing the inter-thread cache benefit.

In addition to achieving the primary goal, the scheduling technique and schedulability test developed for the multi-task `BUNDLE`-based scheduler can be applied to any thread level scheduler with strictly increasing discrete concave WCET functions. This allows any compatible thread-level scheduling technique to benefit from the TPJ approach developed in this work. As a non-preemptive multi-threaded schedulability test TPJ is optimal with respect to npm-feasibility, always producing a feasible task set if one is schedulable by EDF-NP.

For future work, the primary focus is upon a fully or limited preemption scheduling algorithm that permits the inter-thread cache benefit of `BUNDLE`-based schedulers and other schedulers characterized by concave growth to retain their thread-level scheduling benefits.

References

1 S. Altmeyer, R. Davis, and C. Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real Time Systems*, 48(5), 2012.

2 S. Altmeyer, R. I. Davis, and C. Maiza. Cache Related Pre-emption Delay Aware Response Time Analysis for Fixed Priority Pre-emptive Systems. In *IEEE Real-Time Systems Symposium*, pages 261–271, November 2011. `doi:10.1109/RTSS.2011.31`.

3 Sebastian Altmeyer and Claire Maiza Burguière. Cache-related Preemption Delay via Useful Cache Blocks: Survey and Redefinition. *Journal of Systems Architecture*, 57(7):707–719, August 2011. `doi:10.1016/j.sysarc.2010.08.006`.

4 S. Bak, G. Yao, R. Pellizzoni, and M. Caccamo. Memory-Aware Scheduling of Multicore Task Sets for Real-Time Systems. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 300–309, August 2012. `doi:10.1109/RTCSA.2012.48`.

5 S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *[1990] Proceedings 11th Real-Time Systems Symposium*, pages 182–190, December 1990. `doi:10.1109/REAL.1990.128746`.

6 Sanjoy Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 137–144, July 2005. `doi:10.1109/ECRTS.2005.32`.

7 M. Bertogna and S. Baruah. Limited Preemption EDF Scheduling of Sporadic Task Systems. *IEEE Transactions on Industrial Informatics*, 6(4):579–591, November 2010. `doi:10.1109/TII.2010.2049654`.

8 M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazzo. Optimal Selection of Preemption Points to Minimize Preemption Overhead. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pages 217–227, July 2011. `doi:10.1109/ECRTS.2011.28`.

9 E. Bini and G. C. Buttazzo. Biasing effects in schedulability measures. In *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.*, pages 196–203, July 2004. doi:10.1109/EMRTS.2004.1311021.

10 R. Bril, S. Altmeyer, M. van den Heuvel, R. Davis, and M. Behnam. Fixed priority scheduling with pre-emption thresholds and cache-related pre-emption delays: integrated analysis and evaluation. *Real-Time Systems*, 53(4):403–466, July 2017.

11 A. Burns. *Advances in Real-Time Systems*, chapter Preemptive priority-based scheduling: an appropriate engineering approach, pages 225–248. Prentice Hall, Inc., 1995.

12 Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Springer Publishing Company, Incorporated, 3rd edition, 2011.

13 John Michael Calandrino. *On the Design and Implementation of a Cache-aware Soft Real-time Scheduler for Multicore Platforms.* PhD thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 2009.

14 J. Cavicchio, C. Tessler, and N. Fisher. Minimizing Cache Overhead via Loaded Cache Blocks and Preemption Placement. In *Proceedings of the Euromicro Conference on Real-Time Systems*, 2015.

15 Laurent George, Nicolas Rivierre, and Marco Spuri. Preemptive and Non-Preemptive Real-Time UniProcessor Scheduling. Research Report RR-2966, INRIA, 1996. Projet REFLECS. URL: https://hal.inria.fr/inria-00073732.

16 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In *International Workshop on Worst-Case Execution Time Analysis*, volume 15, pages 136–146, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

17 C.-G. Lee, J. Hahn, S.L. Min, R. Ha, S. Hong, C.Y. Park, M. Lee, and C.S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

18 C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 20(1):46–61, January 1973. doi:10.1145/321738.321743.

19 J. M. Marinho, V. Nelis, S.M. Petters, and I. Puaut. An Improved Preemption Delay Upper Bound for Floating Non-preemptive Region. In *Proceedings of IEEE International Symposium on Industrial Embedded Systems*, 2012.

20 H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate Estimation of Cache Related Preemption Delay. In *Proceedings of IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and Systems Synthesis (CODES)*, 2003.

21 R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A Predictable Execution Model for COTS-Based Embedded Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279, April 2011. doi:10.1109/RTAS.2011.33.

22 B. Peng, N. Fisher, and M. Bertogna. Explicit Preemption Placement for Real-Time Conditional Code. In *Proceedings of Euromicro Conference on Real-Time Systems*, 2014.

23 J. Simonson and J.H. Patel. Use of preferred preemption points in cache based real-time systems. In *Proceedings of IEEE International Computer Performance and Dependability Symposium*, 1995.

24 J. Staschulat and R. Ernst. Scalable Precision Cache Analysis for Real-Time Software. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(4), September 2005.

25 Y. Tan and V. Mooney. Integrated intra- and inter-task cache analysis for preemptive multi-tasking real-time systems. In *Proceedings of International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2004.

26 C. Tessler and N. Fisher. BUNDLE: Real-Time Multi-threaded Scheduling to Reduce Cache Contention. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 279–290, November 2016. doi:10.1109/RTSS.2016.035.

27 C. Tessler and N. Fisher. BUNDLEP: Prioritizing Conflict Free Regions in Multi-threaded Programs to Improve Cache Reuse. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 325–337, December 2018. `doi:10.1109/RTSS.2018.00048`.

28 Corey Tessler. `NPM-BUNDLE` Artifacts, 2019. URL: `http://www.cs.wayne.edu/~fh3227/npm-bundle/`.

29 Corey Tessler and Nathan Fisher. BUNDLEP: prioritizing conflict free regions in multi-threaded programs to improve cache reuse - extended results and technical report. *CoRR*, abs/1805.12041, 2018. `arXiv:1805.12041`.

30 Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and Precise WCET Prediction by Separated Cache and Path Analyses. *Real-Time Systems*, 18(2):157–179, May 2000. `doi:10.1023/A:1008141130870`.

31 H. Tomiyama and N. D. Dutt. Program Path Analysis to Bound Cache-Related Preemption Delay in Preemptive Real-Time Systems. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign (CODES)*, 2000.

32 Y. Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Proceedings of the International Conference on Real Time Computing Systems and Applications*, 1999.

33 B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson. Making Shared Caches More Predictable on Multicore Platforms. In *Euromicro Conference on Real-Time Systems*, pages 157–167, July 2013. `doi:10.1109/ECRTS.2013.26`.

34 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-case Execution-time Problem – Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008. `doi:10.1145/1347375.1347389`.

Scheduling Self-Suspending Tasks: New and Old Results

Jian-Jia Chen
TU Dortmund University, Germany
jian-jia.chen@tu-dortmund.de

Tobias Hahn
University of Bremen, Germany
tobiash4hn@gmail.com

Ruben Hoeksma
University of Bremen, Germany
hoeksma@uni-bremen.de

Nicole Megow
University of Bremen, Germany
nicole.megow@uni-bremen.de

Georg von der Brüggen
TU Dortmund University, Germany
georg.von-der-brueggen@tu-dortmund.de

Abstract

In computing systems, a job may suspend itself (before it finishes its execution) when it has to wait for certain results from other (usually external) activities. For real-time systems, such self-suspension behavior has been shown to induce performance degradation. Hence, the researchers in the real-time systems community have devoted themselves to the design and analysis of scheduling algorithms that can alleviate the performance penalty due to self-suspension behavior. As self-suspension and delegation of parts of a job to non-bottleneck resources is pretty natural in many applications, researchers in the operations research (OR) community have also explored scheduling algorithms for systems with such suspension behavior, called the *master-slave* problem in the OR community.

This paper first reviews the results for the master-slave problem in the OR literature and explains their impact on several long-standing problems for scheduling self-suspending real-time tasks. For frame-based periodic real-time tasks, in which the periods of all tasks are identical and all jobs related to one frame are released synchronously, we explore different approximation metrics with respect to resource augmentation factors under different scenarios for both uniprocessor and multiprocessor systems, and demonstrate that different approximation metrics can create different levels of difficulty for the approximation. Our experimental results show that such more carefully designed schedules can significantly outperform the state-of-the-art.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases Self-suspension, master-slave problem, computational complexity, speedup factors

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.16

Supplement Material ECRTS 2019 Artifact Evaluation approved artifact available at
`https://dx.doi.org/10.4230/DARTS.5.1.6`

Funding *Ruben Hoeksma*: Partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project Number 146371743 – TRR 89 Invasive Computing.
Nicole Megow: Partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project Number 146371743 – TRR 89 Invasive Computing.

Acknowledgements The authors thank Minming Li from City University of Hong Kong and Guillaume Sagnol from TU Berlin, for discussions in an early stage of this research. The authors also thank the organizing committee of MAPSP 2017 for planning a discussion session during the workshop, which initialized the study in this paper.

1 Introduction

Advanced embedded computing and information processing systems heavily interact with the physical world in which time naturally progresses. Due to this, *timeliness* of computation is an essential requirement of correctness. Thus, to ensure safe operations of such embedded systems, also called *real-time* embedded systems, worst-case timeliness needs to be verified.

In most real-time embedded systems, control tasks are executed recurrently, i.e., each task τ_i releases an infinite number of tasks instances, called *jobs*, either periodically [32] or sporadically [35], i.e., with a fixed period T_i or a minimum inter-arrival time T_i between two jobs. When a job with relative deadline D_i arrives at the system at time t, it must finish its execution no later than its *absolute deadline* $t + D_i$. If the relative deadline D_i of each task τ_i in the task set is equal to (no more than, respectively) the period T_i, the task set is called an *implicit-deadline* (constrained-deadline, respectively) task set. For real-time systems, two correlated problems exists: (1) *designing scheduling policies* to schedule the tasks and (2) *validating* whether the deadlines are always met in the resulting schedule. We term the former as the *scheduler design* problem and the latter as the *schedulability test* problem.

Most existing approaches to analyze real-time systems work under the important assumption that a job does not suspend itself, therefore allowing to exploit the widely-adopted critical instant theorem [32], the busy-window concept [30], etc. This assumption means that a job that starts executing on the computer either finishes its execution or is preempted by a higher priority job, i.e., the job currently executing is preempted and the processor is allocated to the new arriving job. If tasks can suspend themselves, most scheduling analysis for existing scheduling algorithms cannot be applied without modifications. Nevertheless, in real-world systems self-suspension behavior may occur for multiple reasons, for instance when: (1) external devices are used to accelerate computation, so called computation offloading [15, 25], (2) resources are shared in multiprocessor systems, i.e., when a job requests a resource currently held by a different job on another processor and cannot continue before the resource access is granted [23, 47], (3) direct memory access (DMA) is used to hide the latency of memory accesses [21], etc. In these situations, the execution efficiency of the system may be improved if a job suspends itself and releases the processor, i.e., allowing a lower priority job to run instead of spinning on the processor.

To model self-suspension behavior, three self-suspension task models have been explored in the literature as detailed in recent surveys [10, 12]. The *dynamic* self-suspension model allows a job of task τ_i to suspend itself at any moment before it finishes as long as the worst-case (or maximum) self-suspension time S_i is not violated. The *segmented* self-suspension model further characterizes the computation segments and suspension intervals as $(C_{i,1}, S_{i,1}, C_{i,2}, S_{i,2}, \ldots, S_{i,m_i-1}, C_{i,m_i})$, an array composed of m_i computation segments separated by $m_i - 1$ suspension intervals. The simplest segmented self-suspension model allows a task to have at most one self-suspension interval, i.e., $m_i \leq 2$. The *hybrid* self-suspension model [48] introduces some flexibility into the segmented suspension model by allowing certain combinations of $\sum_{j=1}^{m_i} C_{i,j}$. For instance, when considering two execution segments, the hybrid model is applicable for scenarios where $C_{i,1} + C_{i,2}$ is specified but the detailed information of $C_{i,1}$ and $C_{i,2}$ is not revealed until the job finishes its execution.

The investigation of the impact of self-suspension on timing predictability in real-time systems has started since 1988 by Rajkumar et al. [38]. The early research mainly focused on the schedulability test problem under the classical real-time scheduling algorithms, e.g., [38] in 1988, [34] in 1994, [27] in 1995, [17] in 1998, [33, p. 164-165] in 2000, [14, Section 4.5] in 2003, [1, 2] in 2004, and [5] in 2005.

For periodic segmented self-suspension real-time tasks, the first scheduling algorithm to alleviate the self-suspension behavior, called *period enforcer*, is due to Rajkumar [37] in 1991. In 2004, Ridouard et al. [39] showed that the scheduler design problem for the segmented self-suspension task model is $\mathcal{NP}$-hard in the strong sense. The proof by Ridouard et al. [39] only needs each segmented self-suspending task to have one suspension interval with two computation segments. In 2014, Chen and Liu [9] presented the fixed-relative deadline (FRD) strategy and provided a resource augmentation factor of 3 in uniprocessor systems for the segmented self-suspension task model with at most one self-suspension interval per implicit-deadline task. Since then, FRD has been applied in several results [20, 36, 47–49], and it has been shown by von der Brüggen et al. [49] that the speedup factor of 3 also holds for other FRD approaches. Chen and Brandenburg [8] have recently revisited the period enforcer algorithm and presented its underlying assumptions and limitations. Schönberger et al. [44] considered fixed-priority scheduling, combining suspension as computation and restarting inference for each computation interval.

For scheduling periodic dynamic self-suspension real-time tasks, Huang et al. [22] in 2015 provided a priority assignment scheme which achieves a resource augmentation factor of 2, compared to the optimal fixed-priority scheduling strategy. In 2016, Chen [7] showed that the speedup factor for any fixed-priority preemptive scheduling, compared to the optimal schedules, is not bounded by a constant *if the suspension time cannot be reduced by speeding up*. An unbounded speedup factor has also been proved for the earliest-deadline-first (EDF), the least-laxity-first (LLF), and the earliest-deadline-zero-laxity (EDZL) scheduling algorithms.

Nevertheless, most of the theoretical results regarding speedup factors are byproducts of the construction of scheduling algorithms and a thorough theoretical analysis in this direction has never been performed. Therefore, we focus ourselves on the fundamental analysis of the most basic recurrent setting, i.e., frame-based implicit deadline tasks sets, to provide some theoretical ground work that leads to a deeper understanding of the underlying problem and algorithms to handle the setting efficiently. For instance, a large number of flaws were found in the literature [10] and in our opinion fundamental theoretical results will help to avoid such flaws in the future. Furthermore, we hope that the provided algorithms can be extended to cover more general settings like periodic tasks, especially when harmonic or semi-harmonic task sets are considered, for instance in automotive systems [19, 28, 50].

Our Contribution. In light of the increasing importance of self-suspending behavior in many applications in real-time systems, we examine the fundamental difficulty of the scheduler design problem. The contribution of this paper is as follows:

- We provide a survey of several results in the operations research (OR) community for the *master-slave* problem, which is shown $\mathcal{NP}$-hard in the strong sense by Yu et al. [51] in 2004 even for a very simple setting. This concludes that the computational complexity of the scheduler design problem is **NOT** due to the recurrence of real-time jobs, and that removing the periodicity and non-uniform execution times of the computation segments does **NOT** make the problem easier with respect to the computational complexity. Details can be found in Section 3.
- We provide a systematic study to quantify the resource augmentation (speedup) factors of several heuristic algorithms that can be applied for different self-suspension models. Motivated by the necessity for a fundamental exploration detailed above and the fact that Yu et al. [51] showed that the complexity of self-suspension can be observed even in simple settings, we focus our work on the frame-based task model. Two types of speedup factors are explored in this paper. The *suspension-coherent speedup factor* defines the resource

Table 1 Summary of speedup factors for uni- and multiprocessor systems.

Uniprocessor	segmented (one suspension)	hybrid (one suspension)	dynamic (multiple suspen.)
coherent speedup	1.5 ([42])	1.5 (Cor. 4.5)	2 (Theorem 4.7)
speedup only the processor	2 (Theorem 4.12)	2 (Theorem 4.13)	-

Multiprocessor	segmented (one suspension)	hybrid (one suspension)	dynamic (multiple suspen.)
coherent speedup	2 ([42])	2 ([42] & Thm. 5.5)	2 (Theorem 5.5)
speedup only the processors	$3 - 1/m$ (Thm. 5.9)	$3-1/m$ (Thm. 5.10)	-

augmentation factor by reducing the suspension time and execution time of a job at the same time. The *speedup factor* defines the resource augmentation factor by reducing only the execution time of a job. In addition to providing upper bounds on the speedup factors in the uniprocessor and the multiprocessor setting, we provide lower bounds that show that these two types of factors are very different. Constant *suspension-coherent speedup factors* can be achieved easily by using work-conserving scheduling algorithms. However, speedup factors without reducing the suspension time are much more difficult to achieve. Table 1 summarizes these resource augmentation factors for uniprocessor and multiprocessor systems from the literature and in Sections 4 and 5, respectively, where a "-" denotes the cases where the speedup factor is unknown.

2 Model, Terminology, and Assumptions

In this section we explain the basic task models and terminology used in this paper. For a self-suspending task τ_i, we consider three different models:

- Segmented self-suspension with only one suspension interval: task τ_i is defined by the triple $(C_{i,1}, S_i, C_{i,2})$, where $C_{i,1}$ and $C_{i,2}$ are execution times on a processor and S_i is the suspension time, which for the segmented model is also called the length of the suspension interval.[1] A job of task τ_i suspends itself for S_i amount of time after it is executed for $C_{i,1}$ time units, i.e., the execution of the first computation segment is finished. The second computation segment is released when the job returns from self-suspension. For notational brevity, we denote $C_i = C_{i,1} + C_{i,2}$.
- Dynamic self-suspension: task τ_i is defined by (C_i, S_i), where a job of task τ_i can suspend itself at any moment and several times if needed before it finishes as long as the total self-suspension time of the job is not more than S_i.
- Hybrid self-suspension with only one suspension interval: the tuple (C_i, S_i) defines task τ_i, where a job of τ_i suspends only once for S_i amount of time and the sum of the execution times of the two computation segments is at most C_i, i.e., the individual segments $C_{i,1}$ and $C_{i,2}$ are unknown but the sum of their length $C_{i,1} + C_{i,2} = C_i$ is known.[2]

In this paper, we will implicitly consider *frame-based real-time task systems*. The given tasks release their jobs at the same time, have the same period D, and a uniform relative deadline D. Let $\mathbf{T}$ be the set of the n given tasks. As a result, we do not have to consider

[1] In most task models in the literature, the suspension and execution time are both upper bounds. Here, we consider them to be exact in the segmented self-suspension model to give rigorous worst-case bounds.
[2] In general, the hybrid model assumes $C_{i,1} + C_{i,2} \leq C_i$. In this case, some lemmas have to be revised but the corresponding factors remain the same. Furthermore, in [48] multiple hybrid models are provided that take advantage of additional information about the tasks if available.

the periodicity of the tasks while scheduling frame-based real-time task systems. That is, we assume that each task releases a job at time 0 and the schedule starts always at time 0. We consider both uniprocessor and homogeneous multiprocessor platforms, i.e., m identical processors, to schedule the given self-suspending tasks. When considering the multiprocessor setting, we assume that no intra-task parallelization is possible, i.e., each task can be executed on at most one processor at any given time.

The two problems that we consider are: (1) the scheduler design problem, where we design scheduling policies to schedule the tasks, and (2) the schedulability test problem, where we validate whether the deadlines are always met in the resulting schedule.

A schedule is *work-conserving* if the processor in a uniprocessor system (or a processor in a multiprocessor system) never idles whenever a computation segment is available. A scheduling instance is called *feasible* or *schedulable*, if there exists a schedule on a uniprocessor (multiprocessor, respectively) system of unit speed (on a set of m unit speed processors, respectively) in which all jobs complete before their deadline.

We say that a computation segment is *available* if it could be scheduled. To be precise, the first computation segment of a task is available from the beginning of the frame until it finishes its execution, and the second computation segment becomes available S_i time units after the first computation segment finishes its execution, i.e., after the suspension interval of the task is finished.

Speedup Factors. As it is the case for many interesting problems, the problems that we consider here are $\mathcal{NP}$-hard. Therefore, we cannot hope for exact polynomial time algorithms unless $\mathcal{P} = \mathcal{NP}$. Hence, the metrics of the resource augmentation bound or the speedup factor are widely used to quantify the imperfectness of the scheduling algorithms [24]. Assume that the input task set can be feasibly scheduled on a unit-speed processor by a (not necessarily known) optimal scheduling algorithm. An algorithm $\mathcal{A}$ has a *speedup factor* $\rho \geq 1$ when it can be guaranteed that the schedule derived from algorithm $\mathcal{A}$ is always feasible by running the processor at speed ρ. The speedup factor of a (sufficient) schedulability test can be defined accordingly. When considering a multiprocessor platform, all m processors are assumed to be sped up by ρ.

While for non-suspending task sets any computation is assumed to be sped up by ρ, regarding self-suspension the questions remains weather only the computation segments are sped up or if the suspension interval is sped up as well. Both possibilities are meaningful, depending on the analyzed system. If the suspension interval can be *coherently* reduced by changing the local execution platform, we talk about a *suspension-coherent speedup factor*. For instance, it can be assumed that the suspension interval can be reduced as well when the self-suspension behavior is due to resource access and multiprocessor synchronization on the same platform. On the other hand, if the suspension length cannot be *coherently* reduced by changing the local execution platform the general term speedup factor is used, e.g., if the suspension behaviour is due to computation offloading to an external device.

Please note that the speedup factor should only be considered to analyze the worst-case behavior of an algorithm, since algorithms with similar speedup factors may differ largely regarding their performance. This fact and how considering speedup factors during the algorithm design can lead to reduced performance has been recently elaborated by Chen et al. [11]. To the best of our knowledge, the algorithms presented in this paper do not suffer from any of the potential pitfalls pointed out in [11].

Clairvoyant Schedules. In the hybrid and dynamic self-suspension models, the scheduling algorithm is supposed to be unaware of the exact moment when a job suspends itself. Therefore, the scheduling algorithm works in the on-line fashion. However, according to the self-suspension models, there are upper bounds of the suspension time and the execution time of a job. To analyze the speedup factors and suspension-coherent speedup factors for the hybrid and dynamic self-suspension models, we have to essentially compare to *clairvoyant schedules* that know exactly when a job suspends and plan the best possible schedules.

Approximation guarantee. A polynomial-time algorithm is called ρ-approximation algorithm if it guarantees to derive a feasible solution with an objective value that is within a factor ρ of the optimal objective value for every input instance. The factor ρ is also called approximation factor or guarantee.

3 Master-Slave Problem and Complexity

As self-suspension is pretty natural in many applications, researchers in the operations research (OR) community have also explored scheduling algorithms for systems with such suspension behavior. In 1991, Kern and Nawijn [26] introduced the scheduling of multi-operation jobs with time lags on a single machine. Their problem definition is:

> “There are jobs to be processed on a single machine. Each job requires two operations to be processed in a given order. The time between the start of the second operation and the completion of the first operation cannot be less than a pre-specified time constant, i.e., there is a minimal time lag between the two operations of a job. Our aim is to minimize the makespan, i.e., the completion time of the second operation of the last job in the schedule.”

The two operations defined by Kern and Nawijn [26] are identical to the two computation segments in our segmented suspension model and the *lag* is identical to the self-suspension time. Therefore, the problem studied by Kern and Nawijn [26] is in fact identical to the scheduler design problem for frame-based segmented self-suspending real-time task systems with a single suspension interval per job in uniprocessor platforms. They proved that the decision version of the problem, i.e., whether there exists a schedule to meet the uniform deadline D, is $\mathcal{NP}$-complete in the weak sense (by reduction from the 2-Partition problem).

Kern and Nawijn [26] also explored some special cases that can be solved in polynomial time. Specifically, they concluded that there are polynomial-time scheduling algorithms to derive optimal schedules for a single suspension on a uniprocessor for the following two cases:

- All jobs have the same lag, i.e., uniform suspension time.
- All jobs have only the first operation, i.e., $C_{i,2} = 0$.

As a third special case, they analyzed the case where $C_{i,1} = C_{i,2} = 1$ for all the tasks (jobs), but the computational complexity was left as an open problem [26].

In 1995, Sahni [40, 41] termed the above problem as the *master-slave* scheduling model. It assumes a given number of master devices and a sufficiently large number of slave devices. A job is associated with three activities: preprocessing on the master device (i.e., first computation segment), slave work (i.e., self-suspension interval), and postprocessing on the master device (i.e., second computation segment). It is assumed that the number of slaves is sufficient, i.e., there is always a slave device available if needed, and that the slave device starts working without any delay. Hence, if there is only one master, this problem is identical to the scheduler design problem for the segmented self-suspension task model in uniprocessor systems, and if there are multiple masters, this problem is identical to the scheduler design problem for the segmented self-suspension task model in multiprocessor systems. Sahni [40] proved that the makespan problem for only one master is also $\mathcal{NP}$-hard in the weak sense for the scheduling algorithms that have certain limited capabilities, e.g., the order of the jobs in the preprocessing must be identical to the order in the postprocessing.

In 1996, Sahni and Vairaktarakis [42] proposed several heuristic algorithms to minimize the makespan, i.e., the completion time of the last job, for the master-slave scheduling model under single-master and multiple-master systems. Specifically, an approximation algorithm with an approximation ratio of 3/2 was given for single-master systems and an approximation algorithm with an approximation ratio of 2 was given for multiple-master systems. In 1997, Vairaktarakis [46] further considered variants when there are m_1 masters for preprocessing and m_2 masters for postprocessing. He gave approximation algorithms with an approximation ratio of $2 - 1/\max\{m_1, m_2\}$ for such scenarios. Different configurations for multiple masters in the master-slave scheduling model, including preemptive and non-preemptive constraints and job migration constrains, were further studied by Leung and Zhao [31]. A survey article on the master-slave scheduling model can be found in [43].

The master-slave scheduling model can be viewed as a special case of the two-stage flowshop problem with transfer lags. The two-stage flow shop problem is defined as follows: There are two machine stages each of which has one machine. Each job has to be first processed on the first machine stage and then on the second stage, in which the operation time on each stage is specified as the input. A job cannot be processed on the second stage unless it has finished on the first stage. In classical scheduling theory, scheduling problems are typically described shortly in the three-field notion $\alpha|\beta|\gamma$ [16], where

- α characterizes the processor environment,
- β the job/task parameters, and
- γ gives the objective function.

In this compact notation, the makespan minimization for the two-stage flowshop problem is termed as $F2||C_{\max}$. The transfer lag of a job is defined as the minimum separation time between the start of its operation on the second machine stage and the completion of its operation on the first stage. This problem is termed as $F2|l_j|C_{\max}$. If the transfer lags are long enough such that all of the operations on the first machine stage finish before the second-stage machine starts any operation, then those special cases of $F2|l_j|C_{\max}$ are equivalent to the master-slave scheduling model for one master.

In 1996, Dell'Amico [13] showed that the problem $F2|l_j|C_{\max}$ is $\mathcal{NP}$-hard in the strong sense for both preemptive and non-preemptive settings. In 2004, Yu et al. [51] further proved that the problem $F2|l_j, p_{ij} = 1|C_{\max}$ is $\mathcal{NP}$-hard in the strong sense, where the condition $p_{ij} = 1$ implies that all jobs only need unit time operations in both machines. Specifically, Yu et al. [51, Theorem 24] concluded that the open problem left by Kern and Nawijn [26] mentioned above, i.e., the master-slave scheduling model for a single master with $C_{i,1} = C_{i,2} = 1$ for all the tasks (jobs), is $\mathcal{NP}$-hard in the strong sense.

Therefore, the computational complexity of the master-slave scheduling model as well as the scheduler design problem of segmented self-suspension task systems is in fact mainly due to the non-uniform *self-suspension time.* Removing the periodicity and non-uniform execution times of the computation segments does not make the problem easier with respect to the computational complexity. This result regarding the computational complexity of the scheduler design problem for self-suspending real-time tasks is in fact stronger than the $\mathcal{NP}$-hardness by Ridouard et al. [39] for periodic real-time task systems in 2004.

The above research line has unfortunately been ignored in the real-time systems community while exploring self-suspension task models. The recent survey papers by Chen et al. [10, 12] also did not refer to these results. Although most of these results cannot be applied to generic periodic or sporadic real-time task systems, they have provided solid fundamental results regarding computational complexity and approximation algorithms for the scheduler design problem for self-suspension task models.

4 Speedup Factors: Uniprocessor

This section presents new and old algorithms that have bounded speedup factors on a uniprocessor for scheduling recurrent frame-based task sets, where the given tasks release their jobs at the same time, have the same period D, and a uniform relative deadline D. We will first discuss the suspension-coherent speedup factors and then the speedup factors for the case that the suspension time is not reduced, summarized in Table 1. We provide lower and upper bounds that clearly separate both models in terms of achievable speedup factors.

4.1 Suspension-Coherent Speedup Factors

Let $\mathbf{J}$ be the set of the jobs released by the task set $\mathbf{T}$ at time 0. As mentioned in Section 3, Sahni and Vairaktarakis [42] developed a 3/2-approximation algorithm for the single-master master-slave scheduling model. The algorithm is detailed in Algorithm 1.

Algorithm 1 Sahni-Vairaktarakis' Algorithm (SV).

```
Input: $\mathbf{J}$ on one processor;
1: Classify the jobs generated by $\mathbf{T}$ into two sets: $\mathbf{J}_1$ and $\mathbf{J}_2$, where
   $\mathbf{J}_1 = \{J_i \mid C_{i,1} \leq C_{i,2}, \tau_i \in \mathbf{T}\}$ and $\mathbf{J}_2 = \{J_i \mid C_{i,1} > C_{i,2}, \tau_i \in \mathbf{T}\}$.
2: Order the jobs in $\mathbf{J}_1$ according to a non-decreasing order of $S_i$, i.e., shortest suspension first.
3: Order the jobs in $\mathbf{J}_2$ according to a non-increasing order of $S_i$, i.e. longest suspension first.
4: Schedule the jobs in $\mathbf{J}_1$ first and then in $\mathbf{J}_2$ according to the above orders, and always prioritize
   the first computation segments of the jobs.
```

Note that the schedule is work-conserving, i.e., the uniprocessor always executes a computation segment whenever a computation segment is available.

▶ **Theorem 4.1** ([42]). *SV is a 3/2-approximation algorithm for the single-master master-slave problem.*

While SV is a simple algorithm, Hahn [18] shows that the even simpler Longest-suspension time first algorithm (LSF) displayed in Algorithm 2 also is a 3/2-approximation algorithm.

Algorithm 2 Longest-suspension first (LSF).

```
Input: $\mathbf{J}$ on one processor; jobs are indexed in non-increasing order of $S_j$;
1: Schedule the first computation segments of the jobs in increasing order of index;
2: Then schedule the second computation segments of the jobs as early as possible (when they
   become available) in a work-conserving manner (i.e., first-come-first serve (FCFS));
```

▶ **Theorem 4.2** ([18]). *LSF is a 3/2-approximation algorithm for the single-master master-slave problem.*

For completeness, we provide another proof for Theorem 4.2. Yet, before we do, we give the following straightforward bound on the makespan of any feasible schedule that directly follows from the definition. Hence, the proof is omitted.

▶ **Lemma 4.3.** *The makespan of any uniprocessor schedule for a given task set* $\boldsymbol{T}$ *is at least* $\max\left\{\max_{\tau_i\in\boldsymbol{T}}\{S_i\},\sum_{\tau_i\in\boldsymbol{T}}C_i\right\}$. *This lower bound holds for all the segmented, hybrid, and dynamic self-suspension models.*

Now, we prove Theorem 4.2.

Proof of Theorem 4.2. Let the jobs be indexed in non-increasing order of S_j. Consider the schedule produced by LSF and denote by Δ the time between the time at which LSF finishes the last first segment, i.e., $\sum_{j=1}^{n} C_{j,1}$, and the time at which the last suspension time finishes, i.e., $\max_{J_j\in\mathbf{J}}\sum_{k=1}^{j} C_{k,1}+S_j$. Moreover, let $\mathcal{C}_2$ be the processing volume that is processed after the last suspension time finishes. Then, the following three lower bounds on the makespan of the optimal solution, denoted Opt, hold.

$$\textsc{Opt} \geq \sum_{j=1}^{n} C_j \tag{1}$$

$$\textsc{Opt} \geq \Delta + \sum_{j=1}^{n} C_{j,1} \tag{2}$$

$$\textsc{Opt} \geq \Delta + \mathcal{C}_2 \tag{3}$$

Note that if (1)–(3) hold, then the makespan of LSF is at most

$$\sum_{j=1}^{n} C_{j,1}+\Delta+\mathcal{C}_2 \leq \frac{1}{2}\left(\sum_{j=1}^{n} C_{j,1}+\sum_{j=1}^{n} C_{j,2}+\Delta+\sum_{j=1}^{n} C_{j,1}+\Delta+\mathcal{C}_2\right) \leq \frac{3}{2}\textsc{Opt}\,.$$

Thus, it is sufficient to show that (1)–(3) hold.

From Lemma 4.3 we have that (1) holds. To see why (2) holds, consider the relaxation of the instance where for all $J_j\in\mathbf{J}$, we have $C_{j,2}=0$. Then, the makespan is given by the latest finished suspension. For LSF this is equal to the right hand side of (2). LSF minimizes the makespan in this relaxation, as can be seen by the following simple interchange argument. Consider a non-LSF schedule σ and two jobs J_j and J_k, such that $j<k$ and $C_{j,1}$ is scheduled after $C_{k,1}$. Then, the suspension of J_j finishes after the suspension of J_k. Now, consider the schedule σ' where we reschedule $C_{k,1}$ directly after $C_{j,1}$ and shift all jobs originally scheduled after $C_{k,1}$ forward by that amount, such that there is no idle time. In σ', the time at which $C_{k,1}$ finishes is equal to the time at which $C_{j,1}$ finishes in σ, and $C_{j,1}$ finished earlier in σ' than in σ. This can be repeated until no jobs are scheduled in non-LSF order. Therefore, LSF minimizes the makespan for this relaxation and the right hand side of (2) is a lower bound on the makespan in the optimal schedule.

Now, to see why (3) holds, we consider a similar relaxation, where for all $J_j\in\mathbf{J}$, we have $C_{j,1}=0$. Then, for this relaxation, any work-conserving schedule is optimal, since it minimizes idle time. Compare an optimal solution for the relaxation to the LSF schedule starting from time $\sum_{j=1}^{n} C_{j,1}$. This LSF schedule schedules exactly the same computation segments that the relaxation needs to schedule. Moreover, it is work-conserving by definition, and, since no first segment finishes later than $\sum_{j=1}^{n} C_{j,1}$, no segment is available later than in the relaxation. Thus, the makespan of this LSF schedule, $\Delta+\mathcal{C}_2$, is at most the makespan of an optimal schedule for the relaxation and therefore also at most Opt. ◀

An approximation guarantee ρ for an algorithm for the master-slave problem translates directly to a suspension-coherent speedup factor of ρ for the scheduler design problem for the frame-based segmented self-suspension task model. Let $\textsc{Opt}$ denote the optimal makespan for an input instance I of the master-slave scheduling problem, and let $\textsc{Alg}$ denote the makespan of the ρ-approximation algorithm. By definition $\textsc{Alg} \leq \rho \cdot \textsc{Opt}$. Consider a task set in the frame-based segmented self-suspension task model that consists of the same set of jobs as in I with an additional deadline D. If the task set is feasible then the makespan $\textsc{Opt}$ satisfies $\textsc{Opt} \leq D$. If we speedup the computation *and* suspension with a factor of ρ, then the makespan obtained by the algorithm is $\rho \cdot \textsc{Opt}/\rho \leq D$, and thus, the algorithm is guaranteed to find a feasible schedule for a feasible task set.

▶ **Corollary 4.4.** *Both SV and LSF have a suspension-coherent speedup factor of* $3/2$ *for the scheduler design problem for the frame-based segmented self-suspension task model in uniprocessor systems.*

While SV crucially uses information about the length of $C_{j,1}$ and $C_{j,2}$ to classify job J_j, LSF does not need this information. Hence LSF is directly applicable to the hybrid model.

▶ **Corollary 4.5.** *LSF has a suspension-coherent speedup factor of* $3/2$ *for the scheduler design problem for the frame-based hybrid self-suspension task model in uniprocessor systems.*

In addition to SV, Sahni and Vairaktarakis [42] also showed that any canonical schedule (that starts from the first computation segments followed by the second computation segments) has an approximation ratio of 2 for minimizing the makespan. By the above argumentation, this translates to a suspension-coherent speedup factor of 2 for the hybrid suspension model. Here, we present a slightly stronger result. The suspension-coherent speedup factors for the hybrid and dynamic suspension models can be obtained by considering any arbitrary work-conserving schedule. Before presenting the suspension-coherent speedup factors, we first demonstrate the upper bound of the makespan of a work-conserving schedule.

▶ **Lemma 4.6.** *The makespan of a work-conserving schedule for all the segmented, hybrid, and dynamic self-suspension models is at most* $\max_{\tau_i \in \boldsymbol{T}}\{S_i\} + \sum_{\tau_i \in \boldsymbol{T}} C_i$.

Proof. Suppose that job J_j is the last job finished in the work-conserving schedule. Let f be the makespan of the work-conserving schedule. Since the schedule is work-conserving, from time 0 to f, the processor either idles or executes a job. If the processor idles at time t, since the schedule is work-conserving, job J_j must be suspended at time t; otherwise it should be executed. Therefore, from 0 to f, the maximum idle time is at most the suspension time S_j of job J_j. Since the amount of execution time is $\sum_{\tau_i \in \mathbf{T}} C_i$, we know that

$$f \leq S_j + \sum_{\tau_i \in \mathbf{T}} C_i \leq \max_{\tau_i \in \mathbf{T}} \{S_i\} + \sum_{\tau_i \in \mathbf{T}} C_i \,. \qquad ◀$$

▶ **Theorem 4.7.** *On a uniprocessor, the suspension-coherent speedup factor of any work-conserving scheduling algorithm is* 2 *for scheduling a frame-based task set* $\boldsymbol{T}$ *under both the hybrid self-suspension model and the dynamic self-suspension model. This factor is tight.*

Proof. By Lemma 4.3, if the input task set is feasible (i.e., there exists a feasible schedule), then both $\max_{\tau_i \in \mathbf{T}}\{S_i\} \leq D$ and $\sum_{\tau_i \in \mathbf{T}} C_i \leq D$ hold. By Lemma 4.6, under a suspension-coherent speedup factor of 2, we know that the makespan is at most

$$\frac{\max_{\tau_i \in \mathbf{T}}\{S_i\} + \sum_{\tau_i \in \mathbf{T}} C_i}{2} \leq D\,.$$

The analysis is tight as the following example shows. Consider two jobs: job J_1 with $(C_1, S_1) = (1, \varepsilon)$ and job J_2 with $(C_2, S_2) = (2\varepsilon, 1)$, for an infinitesimal $\varepsilon > 0$. A work-conserving algorithm may schedule J_1 from 0 to $1 - \varepsilon$ and J_2 from $1 - \varepsilon$ to 1, while J_1 suspends at time $1 - \varepsilon$ for ε time units and J_2 suspends at time 1 for 1 time units. The makespan of the above schedule is $2 + \varepsilon$, while scheduling the jobs in the reverse order provides a schedule with a makespan of $1 + 2\varepsilon$.

Since Lemma 4.6 holds for the dynamic suspension model, the proof of the dynamic case is identical to the hybrid case. The tightness example can be applied as well. ◀

4.2 Speedup Factors

In this section, we assume that only the processor speed can be changed. We firstly give a necessary condition that any feasible task set must satisfy. Then we consider a *preemptive* variant of LSF, called pmt-LSF, which may interrupt the processing of a job at any time and continues processing it at any time later. We show that pmt-LSF requires at most a speed of 2. Then we show that a preemptive schedule produced by our algorithm can be transformed into a non-preemptive schedule without increasing the makespan. Based on this, we can argue that also LSF requires a speedup factor of at most 2 – in both, the segmented and the hybrid suspension model.

▶ **Lemma 4.8.** *Let the jobs in* $\boldsymbol{J}$ *be indexed in non-increasing order of* S_j*. Any feasible instance with one suspension satisfies for any job* J_j*:*

$$\max\left\{\sum_{k=1}^{j} C_{k,1}, \sum_{k=1}^{j} C_{k,2}\right\} \leq D - S_j. \tag{4}$$

Proof. Consider a feasible instance with a feasible schedule. Suppose there is at least one job which does not satisfy (4). We distinguish two cases.

(a) Let J_j be the job with the smallest index and $\sum_{k=1}^{j} C_{k,1} > D - S_j$. In a feasible schedule, J_j completes its first computation segment by $D - S_j$. Since not all jobs in $\{J_1, \ldots, J_j\}$ can finish by $D - S_j$, there is a job $J_{k'} \in \{J_1, \ldots, J_{j-1}\}$ that finishes its first computation segment after $D - S_j$. The completion time of this first computation segment is later than $D - S_j \geq D - S_{k'}$ since $S_j \leq S_{k'}$, and thus, job $J_{k'}$ fails to meet the deadline. This contradicts the assumption that we have a feasible schedule. Hence, there cannot be a job J_j with $\sum_{k=1}^{j} C_{k,1} > D - S_j$.

(b) Similarly, let J_j be the smallest-index job with $\sum_{k=1}^{j} C_{k,2} > D - S_j$. In a feasible schedule, J_j does not start its second computation segment earlier than S_j. Since not all jobs in $\{J_1, \ldots, J_j\}$ can start their second computation segments at S_j or later, there must be some job $J_{k'} \neq J_j$, which starts its second computation segment earlier. This start time is strictly less than $S_j \leq S_{k'}$ which is infeasible and gives a contradiction. ◀

▶ **Lemma 4.9.** *Let jobs be indexed in non-increasing order of* S_j*. Any feasible instance for the hybrid suspension model (in which a job suspends at most once) satisfies for any job* J_j

$$\frac{\sum_{k=1}^{j} C_k}{2} \leq D - S_j.$$

Proof. Recall that the hybrid suspension model assumes to know $C_k = C_{k,1} + C_{k,2}$ but is unaware of the actual distribution of $C_{k,1}$ and $C_{k,2}$ before the first computation segment finishes. However, for a concrete distribution of $C_{k,1}$ and $C_{k,2}$ of the given jobs J_k's in $\mathbf{J}$, this set of jobs can be scheduled under the segmented self-suspension model. Therefore, we can directly apply the result in Lemma 4.8 for each given distribution of $C_{k,1}$ and $C_{k,2}$ for the jobs J_k's in $\mathbf{J}$. By the pigeon hole principle, we have

$$\frac{\sum_{k=1}^{j} C_k}{2} \le \max\left\{\sum_{k=1}^{j} C_{k,1}, \sum_{k=1}^{j} C_{k,2}\right\}.$$

Therefore, by the above inequality and Lemma 4.8, we reach the conclusion. ◀

For the analysis of LSF (Algorithm 2) in terms of speedup factors, we first consider the *preemptive* version (see Algorithm 3).

Algorithm 3 Preemptive longest-suspension first (pmt-LSF).

```
Input: J on one processor; jobs are indexed in non-increasing order of $S_j$;
1: At any time schedule the available computation segment with smallest job index. Preempt a
   running job if another lower-index (second) segment becomes available.;
```

▶ **Theorem 4.10.** *For any instance that satisfies Condition* (4) *and $C_j + S_j \le D$, for any job J_j, pmt-LSF finds a feasible schedule on a processor with speed* 2.

Proof. Consider an instance with jobs indexed in non-increasing order of S_j that satisfy (4). Let $\alpha \ge 2$ denote the speedup of the machine.

Consider some job $J_k \in \mathbf{J}$ and the time interval between time 0 and the completion time of the second computation segment of J_k. Whenever J_k is not being executed, then either some other, higher priority, job $J_j \in \mathbf{J}$ with $j < k$ is being executed or J_k is suspended. Thus, the completion time of J_k is bounded by the total computation volume of higher priority jobs in $\mathbf{J}$ processed at speed α and the suspension time S_k, that is, the completion time of job J_k is at most

$$\begin{aligned}\sum_{j=1}^{k} \frac{C_j}{\alpha} + S_k &= \sum_{j=1}^{k} \frac{C_{j,1}}{\alpha} + \sum_{j=1}^{k} \frac{C_{j,2}}{\alpha} + S_k \le \frac{2}{\alpha} \cdot \max\left\{\sum_{j=1}^{k} C_{j,1}, \sum_{j=1}^{k} C_{j,2}\right\} + S_k \\ &\le \frac{2}{\alpha}(D - S_k) + S_k \le D\,,\end{aligned}$$

where the last inequality holds by Lemma 4.9 and since $\alpha \ge 2$. Thus, we conclude that all jobs finish before the deadline D and therefore the schedule is feasible. ◀

Now we first show that there exists a non-preemptive schedule that has makespan equal to the makespan of the schedule produced by pmt-LSF.

▶ **Theorem 4.11.** *Any preemptive schedule produced by pmt-LSF can be transformed into a non-preemptive schedule without increasing the makespan.*

Proof. Let σ be the schedule produced by pmt-LSF and let $C_{j,i}$ be the first computation segment that is preempted. Let $\mathbf{C}$ be the set of computation segments that preempt $C_{j,i}$. First note that a preemption can only happen if the preempting segment became available after the preempted computation segment started to be processed. Thus, since first computation segments are available from time 0, all computation segments in $\mathbf{C}$ must be second computation segments. Then note that the completion time of a second computation segment does not influence the availability of any other computation segment. Lastly, between the start and completion of the processing of $C_{j,i}$ there cannot be idle time, since $C_{j,i}$ remains available. Therefore, the machine only processes $C_{j,i}$ and $\mathbf{C}$, between the start and completion of the processing of $C_{j,i}$.

Now, consider the schedule σ' that is constructed by finished the processing of $C_{j,i}$ before starting the processing of $\mathbf{C}$, where the latter is otherwise processed exactly as in σ. The new schedule is feasible, since none of the computation segments start their processing before the time that they start processing in σ. Clearly, in σ', segment $C_{j,i}$ finishes not later than in σ. Moreover, in σ', the segments in $\mathbf{C}$ finish processing exactly at the time that $C_{j,i}$ finishes processing in σ. Thus the makespan of σ' is not greater that the makespan of σ.

We repeat this process until there are no preempted segments left. ◀

Now we are ready to prove that LSF has a speedup factor of 2 as well.

▶ **Theorem 4.12.** *The speedup factor of LSF is 2 for scheduling a frame-based task set $\boldsymbol{T}$ under the segmented-suspension model on a single processor. This factor is tight.*

Proof. Note that, in the proof of Theorem 4.11, the computation segments in $\mathbf{C}$ are all second computation segments. Moreover, in the non-preemptive schedule, all these segments are processed consecutively without idle time between them, since all processing of the preempted job is shifted to the front. Thus, we can process the jobs in $\mathbf{C}$ in any order, that does not introduce idle time, without changing the makespan. Therefore, LSF computes one particular non-preemptive schedule that has a makespan equal to at most the makespan of the preemptive schedule produced by pmt-LSF.

The analysis for LSF is tight as the following example shows. Consider two jobs: job J_1 with $(C_{1,1}, S_1, C_{1,2}) = (0, 1, 1)$ and job J_2 with $(C_{2,1}, S_2, C_{2,2}) = (1, 1 + \varepsilon, 0)$ for an infinitesimal $\varepsilon > 0$. LSF schedules in a decreasing order of S_j and achieves a makespan of 2 only when given speed 2. The opposite order of scheduling gives an optimal solution of makespan 2 on a unit-speed processor. ◀

LSF only prioritizes the first computation segments of the jobs in $\mathbf{J}$ according to their suspension times. Therefore, they can be applied for the hybrid and dynamic suspension models as well. Interestingly, the knowledge of the exact values of $C_{j,1}$ and $C_{j,2}$ does not improve the speedup factors in such a case.

▶ **Theorem 4.13.** *The speedup factor of LSF is 2 for scheduling a frame-based task set $\boldsymbol{T}$ under the hybrid self-suspension model on a uniprocessor. This factor is tight.*

Proof. LSF does not require the knowledge of $C_{j,i}$. It relies only on the relative order of suspension times S_j and on observing when a segment is completed. Thus, the algorithm and its analysis apply to the hybrid model and the result follows from Theorem 4.12. ◀

In fact, LSF is speedup optimal for the hybrid self-suspension model, i.e., it has the best possible speedup factor, as the following lower bound proves.

▶ **Theorem 4.14.** *There is no deterministic algorithm which can achieve a speedup factor of $2 - \varepsilon$ for an infinitesimal $\varepsilon > 0$ for scheduling a frame-based task set $\boldsymbol{T}$ under the hybrid self-suspension model on a processor. This means that LSF is best possible algorithm with respect to speedup factors.*

Proof. Consider two jobs similar to the example in the proof of Theorem 4.12: job J_1 with $(C_{1,1}, S_1, C_{1,2}) = (0, 1, 1)$ and job J_2 with $(C_{2,1}, S_2, C_{2,2}) = (1, 1, 0)$. In the hybrid self-suspension model, an algorithm knows $C_j = C_{j,1} + C_{j,2}$ but not the individual values

$C_{j,i}$. Hence, in our example, jobs J_1 and J_2 are indistinguishable. W.l.o.g. we may assume that an algorithm schedules job J_2 before J_1 and achieves a makespan of 2 only when given speed 2. The opposite order, that is, jobs J_1 before J_2, gives an optimal solution of makespan 2 on a unit-speed processor. ◀

It is somewhat surprising that LSF is powerful for both the segmented and the hybrid model. It remains open, if another algorithm can improve on LSF in the segmented model by exploiting the exact values $C_{j,1}$ and $C_{j,2}$ for jobs j. However, we rule out that the previously known algorithm SV can improve on LSF.

▶ **Lemma 4.15.** *The speedup factor of SV is at least* 2.

Proof. Consider three jobs: J_1 and J_2 with $(C_{j,1}, S_j, C_{j,2}) = (1, 1, 1)$ for $j \in \{1, 2\}$ and J_3 with $(C_{3,1}, S_3, C_{3,2}) = (1 + \varepsilon, 4, 1 - \varepsilon)$ for an infinitesimal $\varepsilon > 0$. SV classifies $\mathbf{J}_1 = \{J_1, J_2\}$ and $\mathbf{J}_2 = \{J_3\}$ and achieves a makespan of 6 only when given speed 2. The opposite order of scheduling gives an optimal solution of makespan 6 on a unit-speed processor. ◀

4.3 Makespan and Schedulability Tests

As already mentioned in Section 1, the schedulability test problem is also important for real-time systems. After deriving the scheduling algorithms, we should also explore the schedulability conditions. In our model, it is rather straightforward. We simply need to check whether the resulting makespan is at most D. The time complexity of such a schedulability test is the same as the time complexity of the scheduling algorithm. However, for LSF, we can derive the following schedulability test.

▶ **Theorem 4.16.** *Let the jobs in $\boldsymbol{J}$ be indexed in non-increasing order of S_j. Let set $\boldsymbol{A}_j$ be*

$$\boldsymbol{A}_j = \left\{ J_\ell \mid J_\ell \in \boldsymbol{J} \text{ and } S_\ell + \sum_{k=1}^{\ell} C_{k,1} \geq S_j + \sum_{k=1}^{j} C_{k,1} \right\} .$$

If $\sum_{J_k \in \boldsymbol{J}} C_{k,1} + C_{k,2} \leq D$ and every job J_j satisfies

$$\sum_{k=1}^{j} C_{k,1} + \sum_{J_k \in \boldsymbol{A}_j} C_{k,2} \leq D - S_j$$

then LSF derives a feasible schedule for $\boldsymbol{J}$ under the segmented self-suspension model.

Proof. Suppose this is not the case and there is a job J_j that finishes its second computation segment after the deadline D. Then, there is some job j^* such that its second computation segment starts at time

$$r_{j^*} = \sum_{k=1}^{j^*} C_{k,1} + S_{j^*}$$

and there is no idle time between r_{j^*} and the time that J_j finishes. Now, note that $\mathbf{A}_{j^*}$ exactly describes the jobs of which the second computation segment becomes available later than r_{j^*}. Therefore, the time at which j finishes is not later than

$$\sum_{k=1}^{j^*} C_{k,1} + S_{j^*} + \sum_{J_k \in \mathbf{A}_{j^*}} C_{k,2} \leq D\,.$$ ◀

The schedulability condition in Theorem 4.16 can be further extended to a schedulability test of LSF for the hybrid self-suspension model.

▶ **Theorem 4.17.** *Let the jobs in $\boldsymbol{J}$ be indexed in non-increasing order of S_j. If*

1. $\sum_{J_k \in \boldsymbol{J}} C_k \leq D$, *and*
2. *for every combination of $C_{k,1}$ and $C_{k,2}$ such that $C_{k,1} + C_{k,2} = C_k$ for jobs J_k in $\boldsymbol{J}$, we have that for each job J_j,*

$$\sum_{k=1}^{j} C_{k,1} + \sum_{J_k \in \boldsymbol{A}_j} C_{k,2} \leq D - S_j$$

then LSF derives a feasible schedule for $\boldsymbol{J}$ under the hybrid self-suspension model, where

$$\boldsymbol{A}_j = \left\{ J_\ell \mid J_\ell \in \boldsymbol{J} \text{ and } S_\ell + \sum_{k=1}^{\ell} C_{k,1} \geq S_j + \sum_{k=1}^{j} C_{k,1} \right\}.$$

Proof. This is identical to the proof of Theorem 4.16. ◀

The schedulability test provided in Theorem 4.17 requires to consider all combinations of $C_{k,1} + C_{k,2} = C_k$ for every job J_k in $\mathbf{J}$. Tools like Satisfiability Modulo Theories (SMT) can be used to evaluate whether the condition holds (or is violated for unschedulability).

5 Speedup Factors: Multiprocessor Systems

This section presents new and known algorithms with bounded speedup factors for scheduling frame-based task sets in a homogeneous multiprocessor setting. Regarding suspension-coherent speedup factors, we show that a known algorithm from the master-slave scheduling literature has a speedup factor of 2. Then we provide a speedup factor of $3 - 1/m$ for the case that the suspension time is not reduced (see Table 1).

5.1 Suspension-Coherent Speedup Factors

For multiprocessor systems, Sahni and Vairaktarakis [42] developed a 2-approximation algorithm. The algorithm is described by Algorithm 4.

Algorithm 4 Sahni-Vairaktarakis' Algorithm for multiprocessor systems (Multi-SV).

```
Input: J on $m$ processors; jobs are indexed in decreasing order of $C_j$;
1: Schedule the first computation segments of the jobs in order of indices, scheduling each job on
   the first free processor and each of them directly followed by the suspension segment on the
   external source.
2: Then, when no first computation segments are left to schedule, schedule the second computation
   segments of the jobs as early as possible (after they are released) in a work-conserving manner
   on any free processor (i.e., first-come-first serve (FCFS)).
```

▶ **Theorem 5.1** ([42]). *Multi-SV is a 2-approximation algorithm for the multi-master master-slave problem.*

By the same argument as in the uniprocessor case, the theorem implies the following result.

▶ **Corollary 5.2.** *Multi-SV has a suspension-coherent speedup factor of 2 for the scheduler design problem for the frame-based segmented and hybrid self-suspension task models in multiprocessor systems.*

▶ **Lemma 5.3.** *The makespan of a work-conserving schedule, which means that at least one of the m uniprocessors always executes a computation segment whenever a computation segment is available, is at most $\max_{\tau_i \in \boldsymbol{T}} \{S_i + C_i\} + \sum_{\tau_i \in \boldsymbol{T}} C_i/m$. This upper bound holds for all the segmented, hybrid, and dynamic self-suspension models.*

Proof. Suppose that the last job finished in the work-conserving schedule is due to job J_j. Let f be the makespan of the work-conserving schedule. Since the schedule is work-conserving, from time 0 to f, either all of the m processors idle or one (or more) of them executes a job.

- If all the m processors idle at time t, since the schedule is work-conserving, job J_j must be suspended at time t. Therefore, from 0 to f, the maximum idle time is at most the suspension time S_j of job J_j.
- Otherwise, job J_j should be executed or all the m processors are executing jobs. Therefore, from 0 to f, the amount of time that at least one processor is executing a job under work-conserving schedules is at most $C_j + \big((\sum_{\tau_i \in \mathbf{T}} C_i) - C_j\big)/m$.

Hence,

$$f \leq S_j + C_j + \frac{(\sum_{\tau_i \in \mathbf{T}} C_i) - C_j}{m} \leq \max_{\tau_i \in \mathbf{T}} \left\{ S_i + C_i - \frac{C_i}{m} \right\} + \frac{\sum_{\tau_i \in \mathbf{T}} C_i}{m}.$$ ◀

▶ **Lemma 5.4.** *The makespan of any schedule for a given task set $\boldsymbol{T}$ on m homogeneous multiprocessors is at least $\max \left\{ \max_{\tau_i \in \boldsymbol{T}} \{S_i + C_i\}, \sum_{\tau_i \in \boldsymbol{T}} C_i/m \right\}$. This lower bound holds for all the segmented, hybrid, and dynamic self-suspension models.*

Proof. This follows directly from the definition. ◀

▶ **Theorem 5.5.** *On m identical processors, the suspension-coherent speedup factor of any work-conserving scheduling algorithm is 2 for scheduling a frame-based task set $\boldsymbol{T}$ under the hybrid self-suspension model and the dynamic suspension model. The factor is tight.*

Proof. By Lemma 5.4, if the input task set is feasible (i.e., there exists a feasible schedule), then both $\max_{\tau_i \in \mathbf{T}} \{S_i + C_i\} \leq D$ and $\sum_{\tau_i \in \mathbf{T}} C_i/m \leq D$ hold. By Lemma 5.3, under a suspension-coherent speedup factor of 2, we know that the makespan is at most D.

The analysis is tight as a special case when $m = 1$ is tight in Theorem 4.7. Since Lemma 5.3 can be applied also for the dynamic self-suspension model, the proof of this theorem is identical to the proof of Theorem 5.5. The tightness example can be applied as well. ◀

5.2 Speedup Factors

In this section, we present an algorithm with speedup factor $3 - 1/m$. After giving a necessary condition that any feasible task set must satisfy, we consider again first a preemptive scheduling algorithm (pmtn-Multi-LSF, Algorithm 5) which is a list scheduling algorithm prioritizing tasks in decreasing order of suspension times. We show a speedup factor of $3 - 1/m$. Then, we can apply the uniprocessor results (Theorem 4.11) and we argue that any preemptive schedule produced by pmt-Multi-LSF can be transformed into a non-preemptive schedule without increasing the makespan. This gives a non-preemptive algorithm (Multi-LSF, detailed in Algorithm 6) with speedup factor $3 - 1/m$.

We first generalize the necessary condition in Lemma 4.8 to multiprocessors.

▶ **Lemma 5.6.** *Let jobs be indexed in non-increasing order of S_j. Any feasible instance for the multiprocessor model with one suspension satisfies for any job J_j that*

$$\max \left\{ \sum_{k=1}^{j} C_{k,1}, \sum_{k=1}^{j} C_{k,2} \right\} \leq m \left(D - S_j \right). \tag{5}$$

Algorithm 5 Multi-processor preemptive longest-suspension first (pmt-Multi-LSF).

```
Input: J on $m$ processors; jobs are indexed in non-increasing order of $S_j$
1: Assign jobs to machines as follows: consider jobs in order of indices and assign a job to the
   machine that has currently the least total computation time (first and second segments) assigned.
2: On each machine, consider only the jobs assigned to it and at any time schedule the available
   computation segment with the smallest index. Preempt a running job if another lower-index
   (second) segment becomes available.
```

Proof. Consider a feasible instance with a feasible schedule. Suppose there is at least one job which does not satisfy (5). The proof is similar to the one for the uniprocessor case. Again, we distinguish two cases.

(a) Let J_j be the job with the smallest index and $\sum_{k=1}^{j} C_{k,1} > m \cdot (D - S_j)$. In a feasible schedule, J_j completes its first computation segment by $D - S_j$. By time $D - S_j$, the m processors process at most a total load of $m \cdot (D - S_j)$. Thus, not all jobs in $\{J_1, \dots, J_j\}$ can finish by time $D - S_j$, so there is a job $J_{k'} \in \{J_1, \dots, J_{j-1}\}$ that finishes its first computation segment after time $D - S_j$. The completion time of this first computation segment is later than $D - S_j \geq D - S_{k'}$ since $S_j \leq S_{k'}$, and thus, $J_{k'}$ misses the deadline. This contradicts the assumption that we have a feasible schedule. Hence, there cannot be a job J_j with $\sum_{k=1}^{j} C_{k,1} > m \cdot (D - S_j)$.

(b) Similarly, let J_j be the smallest-index job with $\sum_{k=1}^{j} C_{k,2} > m \cdot (D - S_j)$. Between time S_j and D, the m processors process at most $m \cdot (D - S_j)$ total load. Thus, not all jobs in $\{J_1, \dots, J_j\}$ can start their second computation segments at S_j or later. In a feasible schedule, J_j does not start its second computation segment earlier than S_j. Since not all jobs in $\{J_1, \dots, J_j\}$ can start their second computation segments at S_j or later, there must be some $J_{k'} \neq J_j$ among them, which starts its second computation segment earlier. This start time is smaller than $S_j \leq S_{k'}$ which is infeasible and gives a contradiction. ◀

Now we can analyze Algorithm 5, a preemptive list scheduling algorithm that prioritizes tasks in decreasing order of suspension time, Multi-processor preemptive longest-suspension first (pmt-Multi-LSF).

▶ **Theorem 5.7.** *For any instance that satisfies* (5) *and* $C_j + S_j \leq D$*, for any job* J_j*, pmt-Multi-LSF finds a feasible schedule on* m *processors of speed* $3 - 1/m$*.*

Proof. Consider an instance with jobs indexed in decreasing order of S_j that satisfy (5). Let $\alpha \geq 3 - 1/m$ denote the speedup of the machines. Consider some processor i and let A_i denote the set of jobs assigned to i. Notice that for any job $J_j \in A_i$, the total computation volume of higher priority jobs in A_i is at most $\sum_{j=1}^{k-1} C_j/m$ due to the greedy assignment in Step 1 of the algorithm.

Now, consider some job $J_k \in A_i$ and the time interval between time 0 and the completion time of the second computation segment of J_k. Whenever J_k is not being executed on processor i, then either some other, higher priority, job $J_j \in A_i$ with $j < k$ is being executed or J_k is suspended. Thus, the completion time of J_k is bounded by the total computation volume of higher priority jobs in A_i processed at speed α and the suspension time S_k, that is, the completion time is at most

$$\sum_{j=1}^{k-1} \frac{C_j}{\alpha m} + \frac{1}{\alpha} C_k + S_k = \sum_{j=1}^{k} \frac{C_j}{\alpha m} + \frac{1}{\alpha}\left(1 - \frac{1}{m}\right) C_k + S_k \,.$$

Algorithm 6 Multi-processor longest-suspension first (Multi-LSF).

```
Input: J on $m$ processors; jobs are indexed in non-increasing order of $S_j$
1: Assign jobs to machines as follows: consider jobs in order of indices and assign a job to the
   machine that has currently the least total computation time (first and second segments) assigned.
2: On each machine, consider only the jobs assigned to it and at any time schedule non-preemptively
   the available computation segment with the smallest index.
```

Notice that we can bound the first time using (5) as follows

$$\sum_{j=1}^{k-1} \frac{C_j}{m} \leq 2 \cdot \max \left\{ \sum_{j=1}^{k-1} \frac{C_{j,1}}{m}, \sum_{j=1}^{k-1} \frac{C_{j,2}}{m} \right\} \leq 2(D - S_k)\,.$$

Thus, the completion time of J_k is at most

$$\begin{aligned} &\frac{2}{\alpha}(D - S_k) + \frac{1}{\alpha}\left(1 - \frac{1}{m}\right) C_k + S_k = \frac{2}{\alpha} D + \frac{1}{\alpha}\left(1 - \frac{1}{m}\right) C_k + \left(1 - \frac{2}{\alpha}\right) S_k \\ &\leq \frac{2}{\alpha} D + \left(1 - \frac{2}{\alpha}\right)(C_k + S_k) \leq D\,. \end{aligned}$$

In the first inequality, we use that $\alpha \geq 3 - 1/m$, which implies that $\frac{1}{\alpha}(1 - \frac{1}{m}) \leq 1 - \frac{2}{\alpha}$. In the last inequality, we use that in any feasible instance holds $C_k + S_k \leq D$. We conclude that any job completes before the deadline when $\alpha = 3 - 1/m$. ◀

Our findings in the uniprocessor case, imply the following result.

▶ **Theorem 5.8.** *Any preemptive schedule produced by pmt-Multi-LSF can be transformed into a non-preemptive schedule without increasing the makespan.*

Proof. Notice that pmt-Multi-LSF runs on each uniprocessor the Algorithm pmt-LSF (Algorithm 3). Thus, we can directly apply Theorem 4.11 on each processor separately which concludes the proof. ◀

Now, consider the non-preemptive algorithm in Algorithm 6.

▶ **Theorem 5.9.** *The speedup factor of Multi-LSF is $3 - 1/m$ for scheduling a frame-based task set $\boldsymbol{T}$ under the segmented self-suspension model on m processors.*

Proof. The same argumentation of the proof for Theorem 4.12 holds for each machine, separately. Thus, we conclude that Multi-LSF computes a non-preemptive schedule that has makespan equal to at most the makespan of the preemptive schedule produced by pmt-Multi-LSF. ◀

Notice that also Multi-LSF prioritizes jobs in **J** according to their suspension times. When assigning jobs to processors, only total execution times C_j play a role. Therefore, this algorithm can be applied again for both, the hybrid and dynamic self-suspension model.

▶ **Theorem 5.10.** *The speedup factor of Multi-LSF is $3 - 1/m$ for scheduling a frame-based task set $\boldsymbol{T}$ under the hybrid self-suspension model on m processors.*

6 Evaluation

We analyzed the performance in a uniprocessor frame-based setting for the algorithms considered in this paper by evaluating how the algorithm by Sahni & Vairaktarakis (SV) and the Largest Suspension First (LSF) Algorithm perform compared to SEIFDA [49], the state-of-the-art for scheduling period segmented self-suspending tasks with one suspension interval. SEIFDA [49] (short for Shortest Execution Interval First Deadline Assignment) considers the tasks in increasing order of their execution interval, i.e., $T_i - S_i$, which for a frame-based setting is identical to the LSF order. For each task, a virtual deadline is set for both computation segments and after such a deadline is set for all segments of all tasks the segments are scheduled using EDF. The metric to compare the performance is the *acceptance ratio*, i.e., percentage of accepted task sets, with respect to the task set utilization. 100 synthetic task sets were randomly generated for each setting and utilization level, ranging from 0% to 100% system utilization with steps of 1%.

In our evaluations we focused on the impact that the number of tasks and the length of the suspension interval has on the acceptance ratio, considering 3 different values for the cardinality of the task set, i.e., 10, 20, and 50 tasks. For a given cardinality, first a set of utilization values with the same size was generated adopting the UUniFast method [4], ensuring that the total utilization was identical to the currently considered system utilization. The total execution time of the tasks was set accordingly to $C_i = T \cdot U_i$ where T is the length of the frame, set to 1000ms in all experiments, since $U_i = C_i/T$. We generated $C_{i,1}$ as a percentage of C_i, chosen based on a uniform distribution from $[0.1, 0.9]$, and set $C_{i,2}$ accordingly. The suspension length was determined as a random fraction of $T - C_i$, based on a factor x uniformly drawn from an interval of possible values. We considered 3 settings for this interval:

- short suspension: $x \in [0.01, 0.1]$
- moderate suspension: $x \in [0.1, 0.3]$
- long suspension: $x \in [0.3, 0.6]$

Since the evaluations showed similar behaviour independent from the cardinality, in Figure 1 we only display the results for 20 tasks due to space limitations. SEIFDA is clearly outperformed by SV and LSF and the gap enlarges if the suspension interval gets longer. Since SEIFDA is designed for periodic tasks, it considers all possible release patterns of segments. Specifically, it also considers the case that the second computation segment of an already evaluated task is released together with the first segment of the current task and the other way around. This introduces some pessimism, since in the frame-based setting the first segments are always released at the same time, which increases if the suspension intervals get longer. SV always performs better than LSF and here the gap increases as well with the length of the suspension interval. The reason is that, if the tasks with larger suspension intervals are scheduled first, it is likely that at some point after all first segments are executed the processor will idle for some time since all tasks are in their suspension phase at the same time. Since SEIFDA [49] can handle periodic and sporadic task sets and therefore is applicable to a wider range of problems, a performance gain of SV and LSF compared to SEIFDA was expected. Hence, the large performance gain of SV and LSF on the one hand shows that these algorithms perform well for the considered problem and on the other hand shows that an extension of SV and LSF to periodic settings may potentially lead to good results.

However, when analyzing SV it is clear that tasks with a long suspension interval that are in $\mathbf{J}_2$ could jeopardize the schedulability, since they are executed late and therefore could lead to a case where the second segment is released too late to be finished before the deadline.

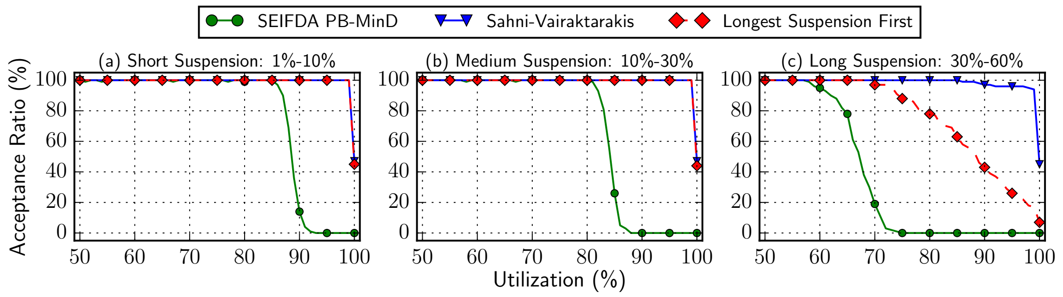


Figure 1 Comparison of Longest Suspension First (LSF) with the algorithm by Sahni & Vairaktarakis (SV) and SEIFDA (20 tasks per Set).

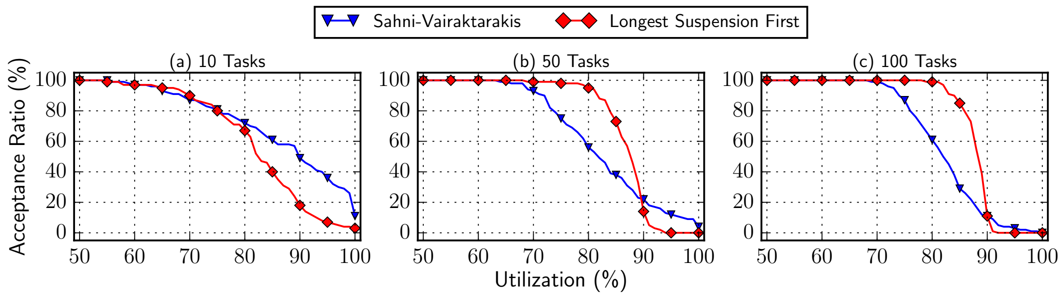


Figure 2 Displaying the case where Longest Suspension First (LSF) performs better than the algorithm by Sahni & Vairaktarakis (SV).

Therefore, this scenario should favor LSF. We conducted evaluations to enforce this case which are displayed in Figure 2. During the task generation process, the suspension intervals are randomly drawn from $[0.1, 0.8]$. Afterwards, the task with the longest suspension interval is placed in $\mathbf{J}_2$ while all other tasks are placed in $\mathbf{J}_1$ by exchanging the computation segments if necessary. Since the suspension intervals are still drawn randomly, the number of tasks plays a big factor to ensure that the suspension interval of the task in $\mathbf{J}_2$ is sufficiently large to create worse cases for SV as shown in Figure 2.

Since LSF and SV do not dominate each other and both have a low runtime complexity, we suggest to run both algorithms and take the better schedule. Furthermore, note that LSF can be used when considering the hybrid self-suspension model while SV is not applicable.

7 Conclusion and Discussions

We have demonstrated algorithms and analyses for different approximation metrics of different self-suspension models for uniprocessor and multiprocessor systems, as shown in Table 1.

In terms of possible speedup factors, we clearly separate the coherent speedup model, in which suspension and processing can be speeded up, from the model in which only the processor changed the speed. In contrast and somewhat surprising, we obtain the same speedup factors for the segmented and hybrid self-suspension models. This means that we have powerful LSF-based algorithms for general frame-based task scheduling, but we do not know how to exploit additional knowledge about the exact execution times of the first and second segment to obtain improved speedup factors in that case.

The dynamic self-suspension model is the most abstract and general self-suspension model. But, it also imposes great challenges to the scheduler design. We are not able to provide any upper bound and lower bound on the speedup factor, even for frame-based real-time task systems. A major difficulty lies in the fact that the speedup factor is defined to compare with a clairvoyant schedule, which knows the exact suspension and execution pattern. It should be mentioned that the analysis of preemptive LSF can be generalized for both, the uni- and the multiprocessor setting. It remains open if we can, and if so how to, convert a preemptive schedule into a non-preemptive one.

An important next step is to extend the results from frame-based real-time tasks to harmonic or nearly-harmonic real-time task systems. A task set is harmonic if the periods are integer multiples of each other. It has been formally proven in uniprocessor environments that scheduling tasks with (nearly-)harmonic periods is significantly better tractable than those with arbitrary periods [3, 6, 29, 50]. Such task systems are important in many industrial applications, e.g., avionic systems [45] and automotive systems [28]. We hope to reach better scheduling algorithms that can handle the studied self-suspension models well for (nearly-)harmonic task systems.

References

1 Neil C. Audsley and Konstantinos Bletsas. Fixed Priority Timing Analysis of Real-Time Systems with Limited Parallelism. In *16th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 231–238, 2004. doi:10.1109/ECRTS.2004.12.

2 Neil C. Audsley and Konstantinos Bletsas. Realistic Analysis of Limited Parallel Software / Hardware Implementations. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 388–395, 2004. doi:10.1109/RTTAS.2004.1317285.

3 Sanjoy K. Baruah and Joël Goossens. Scheduling Real-Time Tasks: Algorithms and Complexity. In Joseph Y.-T. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, chapter 28. CRC Press, 2003.

4 Enrico Bini and Giorgio C Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

5 Konstantinos Bletsas and Neil C. Audsley. Extended Analysis with Reduced Pessimism for Systems with Limited Parallelism. In *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 525–531, 2005. doi:10.1109/RTCSA.2005.48.

6 Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Nicole Megow, and Andreas Wiese. Polynomial-Time Exact Schedulability Tests for Harmonic Real-Time Tasks. In *Proceedings of the IEEE 34th Real-Time Systems Symposium, RTSS 2013, Vancouver, BC, Canada, December 3-6, 2013*, pages 236–245. IEEE Computer Society, 2013.

7 Jian-Jia Chen. Computational Complexity and Speedup Factors Analyses for Self-Suspending Tasks. In *Real-Time Systems Symposium (RTSS)*, pages 327–338, 2016.

8 Jian-Jia Chen and Björn B. Brandenburg. A Note on the Period Enforcer Algorithm for Self-Suspending Tasks. *LITES*, 4(1):01:1–01:22, 2017.

9 Jian-Jia Chen and Cong Liu. Fixed-Relative-Deadline Scheduling of Hard Real-Time Tasks with Self-Suspensions. In *Real-Time Systems Symposium (RTSS)*, pages 149–160, 2014.

10 Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil Audsley, Raj Rajkumar, Dionisio de Niz, and Georg von der Brüggen. Many suspensions, many problems: a review of self-suspending tasks in real-time systems. *Real-Time Systems*, September 2018.

11 Jian-Jia Chen, Georg von der Brüggen, Wen-Hung Huang, and Robert I. Davis. On the Pitfalls of Resource Augmentation Factors and Utilization Bounds in Real-Time Scheduling. In *29th Euromicro Conference on Real-Time Systems, ECRTS*, pages 9:1–9:25, 2017.

12 Jian-Jia Chen, Georg von der Brüggen, Wen-Hung Huang, and Cong Liu. State of the art for scheduling and analyzing self-suspending sporadic real-time tasks. In *23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA*, 2017.

13 Mauro Dell'Amico. Shop Problems With Two Machines and Time Lags. *Operations Research*, 44(5):777–787, 1996.

14 UmaMaheswari C. Devi. An Improved Schedulability Test for Uniprocessor Periodic Task Systems. In *15th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 23–32, 2003.

15 Zheng Dong, Cong Liu, Soroush Bateni, Kuan-Hsun Chen, Jian-Jia Chen, Georg von der Brüggen, and Junjie Shi. Shared-Resource-Centric Limited Preemptive Scheduling: A Comprehensive Study of Suspension-base Partitioning Approaches. In *Proceedings of the 24th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2018.

16 R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling : a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

17 José C. Palencia Gutiérrez and Michael González Harbour. Schedulability Analysis for Tasks with Static and Dynamic Offsets. In *Proceedings of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, December 2-4, 1998*, pages 26–37. IEEE Computer Society, 1998.

18 Tobias Hahn. Algorithms for scheduling with mandatory suspensions: worst-case and empirical analysis. Master's thesis, University of Bremen, 2019.

19 Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. Communication Centric Design in Complex Automotive Embedded Systems. In *29th Euromicro Conference on Real-Time Systems, ECRTS 2017*, pages 10:1–10:20, 2017.

20 Wen-Hung Huang and Jian-Jia Chen. Self-Suspension Real-Time Tasks under Fixed-Relative-Deadline Fixed-Priority Scheduling. In *Design, Automation, and Test in Europe (DATE)*, pages 1078–1083, 2016.

21 Wen-Hung Huang, Jian-Jia Chen, and Jan Reineke. MIRROR: symmetric timing analysis for real-time tasks on multicore platforms with shared resources. In *Proceedings of the 53rd Annual Design Automation Conference, DAC*, pages 158:1–158:6, 2016.

22 Wen-Hung Huang, Jian-Jia Chen, Husheng Zhou, and Cong Liu. PASS: Priority assignment of real-time tasks with dynamic suspending behavior under fixed-priority scheduling. In *Proceedings of the 52nd Annual Design Automation Conference (DAC)*, 2015.

23 Wen-Hung Huang, Maolin Yang, and Jian-Jia Chen. Resource-Oriented Partitioned Scheduling in Multiprocessor Systems: How to Partition and How to Share? In *Real-Time Systems Symposium (RTSS)*, pages 111–122, 2016.

24 Bala Kalyanasundaram and Kirk Pruhs. Speed is As Powerful As Clairvoyance. *Journal of ACM*, 47(4):617–643, July 2000.

25 W. Kang, S. Son, J. Stankovic, and M. Amirijoo. I/O-Aware Deadline Miss Ratio Management in Real-Time Embedded Databases. In *Proc. of the 28th IEEE Real-Time Systems Symp.*, pages 277–287, 2007.

26 W. Kern and W. Nawijn. Scheduling multi-operation jobs with time lags on a single machine. In *2nd Twente Workshop on Graphs and Combinatorial*, 1991.

27 In-Guk Kim, Kyung-Hee Choi, Seung-Kyu Park, Dong-Yoon Kim, and Man-Pyo Hong. Real-time scheduling of tasks that contain the external blocking intervals. In *RTCSA*, 1995. doi:10.1109/RTCSA.1995.528751.

28 Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

29 Tei-Wei Kuo and Aloysius K. Mok. Load Adjustment in Adaptive Real-Time Systems. In *IEEE Real-Time Systems Symposium*, pages 160–171, 1991.

30 John P. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *RTSS*, pages 201–209, 1990. doi:10.1109/REAL.1990.128748.

31 Joseph Y.-T. Leung and Hairong Zhao. Minimizing Sum of Completion Times and Makespan in Master-Slave Systems. *IEEE Trans. Computers*, 55(8):985–999, 2006.

32 C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

33 Jane W. S. Liu. *Real-Time Systems.* Prentice Hall PTR, 1st edition, 2000.

34 Li Ming. Scheduling of the Inter-Dependent Messages in Real-Time Communication. In *Proc. of the First International Workshop on Real-Time Computing Systems and Applications*, 1994.

35 Aloysius Ka-Lau Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1983.

36 Bo Peng and Nathan Fisher. Parameter Adaptation for Generalized Multiframe Tasks and Applications to Self-Suspending Tasks. In *International Conference on Real-Time Computing Systems and Applications (RTCSA)*, pages 49–58, 2016.

37 Ragunathan Rajkumar. Dealing with Suspending Periodic Tasks. Technical report, IBM T. J. Watson Research Center, 1991.

38 Ragunathan Rajkumar, Lui Sha, and John P. Lehoczky. Real-Time Synchronization Protocols for Multiprocessors. In *Proceedings of the 9th IEEE Real-Time Systems Symposium (RTSS '88)*, pages 259–269, 1988.

39 Frédéric Ridouard, Pascal Richard, and Francis Cottet. Negative Results for Scheduling Independent Hard Real-Time Tasks with Self-Suspensions. In *RTSS*, pages 47–56, 2004. doi:10.1109/REAL.2004.35.

40 Sartaj Sahni. Scheduling Master-Slave Multiprocessor Systems. In *Parallel Processing, First International Euro-Par Conference*, pages 611–622, 1995.

41 Sartaj Sahni. Scheduling Master-Slave Multiprocessor Systems. *IEEE Trans. Computers*, 45(10):1195–1199, 1996.

42 Sartaj Sahni and George L. Vairaktarakis. The master-slave paradigm in parallel computer and industrial settings. *J. Global Optimization*, 9(3-4):357–377, 1996.

43 Sartaj Sahni and George L. Vairaktarakis. The Master-Slave Scheduling Model. In *Handbook of Scheduling.* Chapman and Hall/CRC, 2004.

44 Lea Schönberger, Wen-Hung Huang, Georg von der Brüggen, Kuan-Hsun Chen, and Jian-Jia Chen. Schedulability Analysis and Priority Assignment for Segmented Self-Suspending Tasks. In *24th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2018, Hakodate, Japan, August 28-31, 2018*, pages 157–167, 2018.

45 Lui Sha. Real-Time Virtual Machines for Avionics Software Porting and Development. In *Real-Time and Embedded Computing Systems and Applications, 9th International Conference, RTCSA*, pages 123–135, 2003.

46 George L. Vairaktarakis. Analysis of scheduling algorithms for master–slave systems. *IIE Transactions*, 29(11):939–949, November 1997.

47 Georg von der Brüggen, Jian-Jia Chen, Wen-Hung Huang, and Maolin Yang. Release enforcement in resource-oriented partitioned scheduling for multiprocessor systems. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems, RTNS*, pages 287–296, 2017.

48 Georg von der Brüggen, Wen-Hung Huang, and Jian-Jia Chen. Hybrid Self-Suspension Models in Real-Time Embedded Systems. In *International Conference on Real-Time Computing Systems and Applications (RTCSA)*, 2017.

49 Georg von der Brüggen, Wen-Hung Huang, Jian-Jia Chen, and Cong Liu. Uniprocessor Scheduling Strategies for Self-Suspending Task Systems. In *International Conference on Real-Time Networks and Systems (RTNS)*, 2016.

50 Georg von der Brüggen, Niklas Ueter, Jian-Jia Chen, and Matthias Freier. Parametric utilization bounds for implicit-deadline periodic tasks in automotive systems. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems, RTNS*, 2017.

51 Wenci Yu, Han Hoogeveen, and Jan Karel Lenstra. Minimizing Makespan in a Two-Machine Flow Shop with Delays and Unit-Time Operations is NP-Hard. *J. Scheduling*, 7(5):333–348, 2004.

Impact of DM-LRU on WCET: A Static Analysis Approach

Renato Mancuso
Boston University, MA, USA
rmancuso@bu.edu

Heechul Yun
University of Kansas, Lawrence, KS, USA
heechul.yun@ku.edu

Isabelle Puaut
University of Rennes 1/IRISA, France
isabelle.puaut@irisa.fr

Abstract

Cache memories in modern embedded processors are known to improve average memory access performance. Unfortunately, they are also known to represent a major source of unpredictability for hard real-time workload. One of the main limitations of typical caches is that content selection and replacement is entirely performed in hardware. As such, it is hard to control the cache behavior in software to favor caching of blocks that are known to have an impact on an application's worst-case execution time (WCET).

In this paper, we consider a cache replacement policy, namely DM-LRU, that allows system designers to prioritize caching of memory blocks that are known to have an important impact on an application's WCET. Considering a single-core, single-level cache hierarchy, we describe an abstract interpretation-based timing analysis for DM-LRU. We implement the proposed analysis in a self-contained toolkit and study its qualitative properties on a set of representative benchmarks. Apart from being useful to compute the WCET when DM-LRU or similar policies are used, the proposed analysis can allow designers to perform WCET impact-aware selection of content to be retained in cache.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Theory of computation → Caching and paging algorithms

Keywords and phrases real-time, static cache analysis, abstract interpretation, LRU, deterministic memory, static cache locking, dynamic cache locking, cache profiling, WCET analysis

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.17

Funding This research is supported in part by NSF CNS 1718880. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

Acknowledgements We are especially grateful to Daniel Grund for making his research thesis [15] promptly available to us.

1 Introduction

Most modern embedded processors include cache(s) to improve average performance by reducing average memory access cost. However, a well-known downside of using caches is that it makes timing analysis difficult because software has little, if any, control over whether a certain memory block is in the cache or not, as it is determined by the hardware – the cache replacement policy and the state of the cache. This is problematic because analyzing precise and tight worst-case timing is necessary for real-time systems. While there are timing analysis techniques for well-known cache replacement policies [42], they cannot take advantage of programmer's insights (e.g., important data used in time-critical loops), potentially resulting in pessimistic timing.

On the other hand, a scratchpad memory is similar to a cache as it offers high-speed temporary storage for a processor, but the key difference is that it is entirely managed by software. For real-time systems, the fact that software, not hardware, has full control over its management is highly beneficial because accurate timing analysis is possible. However, the downside of scratchpad is that it is generally more difficult to use than cache due to its high programming complexity [3]. Alternatively, some cache designs support selective cache locking, which enables programmers to lock certain cache-lines in the cache at a fine-granularity (typically a cache line) [2, 7, 13]. A locked cache-line stays in the cache until it is explicitly unlocked by the programmer, which guarantees predictable timing. However, because the cache size is limited, the programmer must carefully select which cache-lines to be locked [5, 40]. Dynamic cache-locking techniques [39, 47] can help alleviate the size limitation problem of static cache-locking, but at the cost of increased complexity (for selecting locked cache lines) and overhead (to change cache contents dynamically).

In this paper, we consider a new cache architecture, which can leverage programmers' high-level insights on access frequency of memory blocks, and propose an abstract interpretation-based static analysis method to reason on the worst-case execution time (WCET) of applications. Our approach is based on a new memory abstraction, called Deterministic Memory (DM). Deterministic Memory enables classification of a program's address space into two distinct memory types – DM and non-DM [10], where the DM type indicates predictability is more important while the non-DM type indicates average performance is more important. The DM abstraction allows effective and extensible software/hardware co-designs, some of which are demonstrated in the context of providing efficient hardware isolation in multicore [10]. In this work, we instead focus on a single-core with a private cache, and study how static guarantees on cache hits/misses can be derived for a DM-aware LRU cache replacement policy, which we call DM-LRU.

We first describe the DM-LRU cache replacement algorithm, which is a single-core adaptation of the DM-aware cache initially proposed in [10]. Next, we generalize an abstract interpretation-based analysis for LRU caches to reason on the worst-case behavior of DM-LRU. We integrated DM-LRU support in Heptane [23], an academic static WCET analysis tool, in order to evaluate the effectiveness of DM-LRU in lowering tasks' WCET. Our results show that with DM-LRU WCET improvements up to 23.7% can be achieved, compared to vanilla LRU. The WCET improvements are comparable to static and dynamic cache locking techniques while significantly lowering programming complexity. Our contributions are as follows:

- We extend LRU abstract interpretation-based analysis to perform static WCET timing analysis for DM-LRU.
- We implement DM-LRU support in the Heptane static WCET analysis tool.
- We provide experimental evaluation results showing the WCET benefits and complexity reduction of the DM-LRU based approach.
- We propose a WCET-driven heuristic approach to select content to be preferentially cached using DM-LRU.

The remainder of the paper is organized as follows. Section 2 introduces necessary background on caches and the deterministic memory abstraction. Next, the DM-LRU policy is described in Section 3 and the proposed static timing analysis is described in Section 4. A comprehensive example on how to apply the proposed analysis is presented in Section 5.

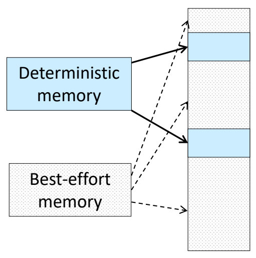


Figure 1 High-level application's memory view, where DM and BE memory coexist.

Comparison and differences with cache locking techniques are briefly highlighted in Section 6, while the WCET of a set of representative benchmarks is evaluated in Section 7. Section 8 discuss related work and we conclude in Section 9.

2 Background

In this section, we provide necessary background on memory abstractions, cache replacement algorithms, and cache timing analysis.

2.1 Deterministic Memory Abstraction

Traditionally, operating systems and hardware have provided a simple uniform memory abstraction to applications. While the simple abstraction is convenient for programmability, its downside is that programmer's insights on memory characteristics (e.g., time-criticality of certain data structures) cannot be explicitly expressed to enable better resource management.

Recently, a new memory abstraction, called Deterministic Memory abstraction, was proposed to explore the possibilities of more expressive memory abstractions [10]. In essence, the abstraction allows a programmer to associate (tag) a single bit of information to each memory block in the system, which classifies the memory block as either "deterministic memory" (DM) or "best-effort memory" (BE). Figure 1 shows an example address space of a task using both deterministic and best-effort memory. In [10], the memory tagging is implemented at the page granularity, although more fine-granularity tagging is also possible (e.g., [45]).

Once a task's memory blocks are tagged, the information can then be used by the operating system and the hardware to apply different resource management policies depending on the memory tag information. In [10], the DM abstraction is used to achieve hardware isolation among the cores in multicore, focusing on effective isolation of shared cache and DRAM.

2.2 DM-LRU Cache Replacement Policy

In this paper, we consider a deterministic memory-aware *private* cache design and show how such a design enables tighter static WCET cache timing analysis. We assume the cache controller has a mean to distinguish whether a certain cache-line corresponds to deterministic memory or best-effort one. This can be implemented as an additional bit in the auxiliary tag store of each cache-line, as in [10], or as a set of separately located architectural hardware range registers as in [27]. The cache implements an extended least recently used (LRU) cache replacement algorithm, which defines two *eviction classes* using the DM/BE abstractions and applies LRU-based replacement to DM lines and to BE lines separately. Allocation of a DM line can cause eviction of a BE line, but the opposite is not allowed. Note that prior work that implements a similar cache replacement policy exists [27]. In this paper, we call the extended LRU as Deterministic Memory-aware Least Recently Used, or DM-LRU for short. A more formal definition of DM-LRU is given in Section 3.

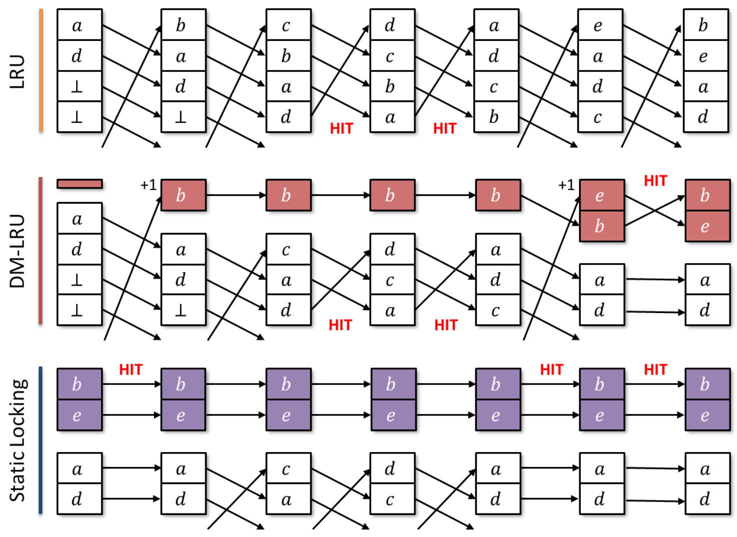


Figure 2 Comparison between traditional LRU, DM-LRU, and a statically locked LRU cache over the same access pattern b, c, d, a, e, b, where b and e are DM memory blocks, or statically locked.

Figure 2 illustrates the difference between traditional LRU, DM-LRU, and static locking. For simplicity, the example considers a single set of a 4-way set-associative cache. In the first step, only a and d are cached, and 0 lines are allocated for DM blocks under DM-LRU. Moreover, blocks b and e are set as DM blocks under DM-LRU, and pre-allocated in cache in case of static locking. The figure tracks the evolution of the cache state for the same access sequence b, c, d, a, e, b. A miss for a DM block triggers an increase of the number of ways allocated for the DM class. This is depicted in step 2 (miss on b) and 6 (miss on e). Traditional LRU simply ignores the DM/BE tag of the considered memory blocks. First, note that DM-LRU results in fewer misses compared to LRU, as the DM marked memory block b was not evicted by the best-effort memory accesses. Also note that while realizing the same number of hits in the example compared to static locking, two important remarks are required. First, the figure does not include the time spent to prefetch and lock the b and e blocks. Second, static locking causes additional misses for non-locked blocks compared to DM-LRU. This exemplifies the on-demand nature of DM-LRU, which is able to retain in cache blocks as they become needed during a task's execution. We discuss the analogies and differences between DM-LRU and static/dynamic locking more extensively in Section 6.

Intuitively, it is thanks to the on-demand allocation and differential treatment of DM memory blocks that DM-LRU enables tighter worst-case cache timing analysis, as we show in the rest of the paper.

2.3 Cache Analysis via Abstract Interpretation

In this work, we extend abstract interpretation-based analysis to reason on the hit/miss classification of memory accesses when a DM-LRU cache controller is implemented in hardware. Analysis via abstract interpretation was originally proposed for LRU caches [11] and better formalized and extended to FIFO and Pseudo-LRU in [41, 15]. An excellent survey on the topic was proposed in [32]. We reuse the notation in [15, 32], while some details are omitted due to space constraints. Since this work focuses on a DM-aware extension of LRU, we introduce some of the background related to abstract interpretation-based LRU analysis.

Imagine taking a snapshot of the cache state at a given point in time. In this case, one could highlight the *state* of the cache in terms of: (i) which blocks are currently in cache, and (ii) what is the *age* of each block. In LRU, the age of a block, say block a, captures the number of memory accesses (to other blocks than a) that were performed since the last access to a. For instance, in the six steps in Figure 2, the LRU age for a is in the following sequence: 0, 1, 2, 3, 0, 1, and 2. If a has an LRU age greater than or equal to the number of ways (4 in our example), then a is not cached.

If the ages of all the cached blocks are known, the cache is in a **concrete state**. From a concrete state, it is possible to produce a new concrete state that follows each new memory access (state update), as shown in Figure 2. In a typical program, however, execution may follow different paths. This means that at a given point in time, multiple concrete states are possible, depending on the execution path taken by the program in its control-flow graph (CFG).

Instead of keeping track of all the possible concrete states at any point of the CFG, abstract interpretation keeps track of two main pieces of information: (i) the upper-bound and (ii) the lower-bound on the age of any memory block among all the possible concrete states. Analysis on the age upper-bound and lower-bound is carried on separately. The former is referred to as *must-analysis*, while the latter goes under the name of *may-analysis*. A state that summarizes the upper-bound (resp., lower-bound) of each block in a set of possible concrete states is called an **abstract state**. For instance, consider a *must-analysis* abstract state of the form: $\bar{q} = [\{\}, \{a, b\}, \{\}, \{d, e\}]$. This corresponds to all the concrete states where blocks a, b have age at most 1, and d, e at most 3. The full *concretization* of $\bar{q}$ is the set: $\{[a, b, d, e], [b, a, d, e], [a, b, e, d], [b, a, e, d]\}$. Similarly, consider the *may-analysis* abstract state $\underline{q} = [\{\}, \{\}, \{a, b\}, \{\}]$. A concretization of $\underline{q}$ is the set $\{[\bot, \bot, \bot, \bot], [\bot, \bot, a, \bot], [\bot, \bot, \bot, a], [\bot, \bot, b, \bot], [\bot, \bot, \bot, b], [\bot, \bot, a, b], [\bot, \bot, b, a]\}$, where $\bot$ is a generic unknown block.

Given a *must*-analysis abstract state, it is possible to determine – i.e., *classify* – a memory access as *always-hit* (H). These are accesses that result in hits regardless of the path taken in the CFG. Similarly, given a *may*-analysis abstract state, it is possible to perform classification of *always-miss* memory accesses. If neither classification applies, the block is simply non-classified (NC). NC, often indicated as $\top$, represents the case in which some execution paths lead to a miss while others lead to a hit for the same memory access.

Note that for architectures without timing anomalies [31, 20], *must*-analysis is sufficient to safely compute the WCET of an application. In fact in this case NC accesses can be simply treated as misses. We developed and implemented both *must*- and *may*-analysis for DM-LRU, but we hereby focus in greater detail on *must*-analysis. Additional details about *may-analysis* are provided in the appendix.

3 Cache Model and Terminology

In this section we discuss the cache model adopted to represent the behavior of DM-LRU, and we introduce key concepts required to follow the proposed abstract interpretation analysis.

3.1 DM-LRU Model

Algorithm 1 shows the full pseudo-code of the DM-LRU cache replacement algorithm. The algorithm is defined for a generic A-way set-associative cache with S sets. The index of a set is indicated with $s \in \{0, \ldots, S-1\}$. In the algorithm, $DetMask_s$ denotes the bitmask of the set s's cache lines that contain deterministic memory. Consider a DM request ($DM = 1$) that resulted in a cache miss – see step 1 or 6 in Figure 2. The algorithm first tries to evict a BE cache line, if such a line exists (Line 3-4). This also causes an additional bit to be asserted in the $DetMask_s$ bitmap. If no BE can be evicted (i.e., all lines are deterministic ones), it chooses one of the deterministic lines (the older one in the LRU stack) as the victim (Line 6). On the other hand, consider the case where a BE memory block is requested ($DM \neq 1$), resulting in a miss – steps 1 and 2 in Figure 2. DM-LRU evicts one of the best-effort cache lines, but not any of the deterministic cache lines (Line 9).

Algorithm 1: Deterministic memory-aware cache line replacement algorithm.

```
   Input : $DetMask_s$ - deterministic ways of Set $s$
   Input : $A$ - cache associativity
   Output: $victim$ - the victim way to be replaced or NULL if no replacement possible
 1 if $DM == 1$ then
 2     if $(\neg DetMask_s) \neq NULL$ then
           // evict a best-effort line first
 3         $victim = LRU(\neg DetMask_s)$
 4         $DetMask_s \mid= 1 \ll victim$
 5     else
           // evict a deterministic line
 6         $victim = LRU(DetMask_s)$
 7     end
 8 else
 9     if $(\neg DetMask_s) \neq NULL$ then
           // evict a best-effort line
10         $victim = LRU(\neg DetMask)$
11     else
           // no BE line can be allocated
12         $victim = NULL$
13     end
14 end
15 return $victim$
```

We assume a single-core, single-level set-associative cache. We indicate with A the associativity of the cache. Since DM-LRU operates independently on each set, it is possible to describe our analysis on a single set without loss of generality. Hereafter, we consider a single cache set. At any point in time, D is the number of cache lines allocated to DM memory blocks for the considered cache set. D is the number of bits set to "1" in the $DetMask_s$ for the set under analysis. We indicate with B the number of lines that have not been allocated for DM memory. It holds that $D + B = A$. Note that if $D < A$, and a DM line that is currently not cached as a DM line is accessed, then the new DM line is allocated and D is increased by one. This may trigger the eviction of the least recently used BE block, as per Algorithm 1.

3.2 Terminology and notations

We indicate with $\mathcal{B}$ the set of memory blocks that map to the cache set under analysis. A generic memory block $b^{CL} \in \mathcal{B}$ is comprised of an address b and an *eviction class* $CL = \{DM, BE\}$. The set of all the possible concrete states of a DM-LRU cache is denoted as Q_{DM-LRU_A}, where each state $q \in Q_{DM-LRU_A}$ is defined as follows:

$$q := \{D, [b_0^{DM}, \dots, b_{D-1}^{DM}], [b_D^{CL}, \dots b_{A-1}^{CL}]\}, \tag{1}$$

where $D \in [0, A]$ and $b_i^{CL} \in \mathcal{B}$. Note that the first D cache lines are allocated as DM cache lines, hence these are necessarily DM memory blocks. The remaining $A - D$ blocks are currently allocated BE memory blocks. Throughout this paper we will use the shorthand notation $b_i \in \mathcal{B}$ for blocks whose eviction class is obvious from context or unimportant. For blocks allocated as BE, we assume BE class unless specified otherwise.

An important concept is the **age** of a memory block under DM-LRU, defined as follows.

▶ **Definition 1** (DM-LRU Age). *The age of a DM memory block a^{DM} is defined as the number of distinct DM blocks accessed since the last access to a^{DM}; the age of a BE memory block b is set to the current value of D whenever b^{BE} is accessed. It is then defined as $D+K$, where K is the number of misses to DM blocks, or accesses to distinct BE blocks since the last access to b^{BE}.*

Following Definition 1, the index of a given block $b_i^{CL} \in q$ is also the age of the block in DM-LRU. The age of a block b_i^{DM} allocated as DM can increase if: (1) a new DM line is allocated (with age 0); or (2) a line b_j^{DM} already allocated as DM with age greater than b_i is accessed. Conversely, the age of a BE block b_i^{BE} can increase if: (1) a new DM line is allocated (with age 0); (2) a new BE line is allocated (with age D); or (3) a line b_j^{BE} already in cache with age greater than b_i^{BE} is accessed.

Also note that Definition 1 remains consistent for the case in which a block b^{BE} is accessed but cannot be allocated because all the sets have been reserved for DM lines. This phenomenon goes under the name of *DM takeover*, and can be resolved by imposing a hard cap on the maximum number of DM lines that can be allocated. The analysis for a DM-LRU with an allocation cap is almost identical to an unrestricted DM-LRU, and only introduces uninteresting subcases. For simplicity, we hereby focus on the analysis for unrestricted DM-LRU. We demonstrate that preventing DM takeover is indeed necessary and beneficial in Section 7.

4 DM-LRU Analysis

In this section we detail our abstract interpretation-based analysis [15, 32] for DM-LRU, i.e. when the cache controller implements the policy defined in Algorithm 1. We discuss *must*-analysis in detail. As previously mentioned, *may*-analysis is not strictly required for architectures without timing anomalies. As such we only provide the intuition behind it and defer the details to the appendix. We do not provide a persistence analysis for DM-LRU. Persistence analysis is useful to determine if memory accesses inside loops can result in hits after the first iteration. Instead, for our evaluations, we unroll the first iteration of each loop, i.e., we perform *virtual unrolling, virtual inlining* (VIVU) [34, 32].

4.1 Must-analysis

Must-analysis is performed considering abstract cache states. In this case, *must-analysis* keeps track of the upper bound on the number of allocated DM blocks indicated with $D \in \{0, \ldots, A\}$, and the upper-bound on the DM-LRU age of each addressable memory block $b \in \mathcal{B}$. The abstract domain $DMLru_A^{\sqsubseteq}$ is defined as:

$$DMLru_A^{\sqsubseteq} := \{0, \ldots, A\} \times \mathcal{B} \to \{0, \ldots, A-1, \infty\}. \quad (2)$$

Intuitively, the domain associates a *current eviction class* (DM or BE) and an age upper bound $(0, \ldots, A$ or $\infty)$ to a memory block $b \in \mathcal{B}$ mapping to the set under analysis. We use the notation $\bar{q}(b)$ to indicate the upper-bound on the age of b in $\bar{q}$. To represent a generic abstract state $\bar{q} \in DMLru_{A}^{\sqsubseteq}$ we use a compact notation that highlights the distinction between DM and BE allocations. For instance, the notation

$$\bar{q} = [\{\}, \{a, b\}], [\{c\}, \{d\}] \in DMLru_{A}^{\sqsubseteq} \tag{3}$$

denotes an abstract state $\bar{q}$ where $D \leq 2, B \geq 2, A = 4$. Hence, blocks a and b have upper-bound $\bar{q}(a) = \bar{q}(b) = 1$ on their DM-LRU age. Similarly, c, d are BE blocks with $\bar{q}(c) = 2$ and $\bar{q}(d) = 3$, respectively.

Given an abstract state $\bar{q} \in DMLru_{A}^{\sqsubseteq}$, the Boolean operator $DM^{\sqsubseteq}(\bar{q}, b)$ returns *true* only if the block $b \in \mathcal{B}$ must exist as a DM-allocated block in $\bar{q}$. Formally

$$DM^{\sqsubseteq}(\bar{q}, b^{CL}) := \begin{cases} true & \text{if } CL = DM \wedge \bar{q}(b) < \infty \\ false & \text{otherwise.} \end{cases} \tag{4}$$

For instance, considering $\bar{q}$ defined as in Equation 3, we obtain $DM^{\sqsubseteq}(\bar{q}, a) = true$, $DM^{\sqsubseteq}(\bar{q}, d) = false$, and so on. We use the simpler notation $DM^{\sqsubseteq}(b)$ when the state is implicit. The operator $BE^{\sqsubseteq}(\bar{q}, b)$ is simply defined as $BE^{\sqsubseteq}(\bar{q}, b) := \neg DM^{\sqsubseteq}(\bar{q}, b)$. To prevent additional clutter in our notation, $DM^{\sqsubseteq}(\bar{q}, b^{DM})$ evaluates to *true* if and only if the DM block b^{DM} *must* be allocated in cache in $\bar{q}$. As such, if the generic DM block b^{DM} has an upper-bound on its DM-LRU age greater than $A - 1$, then $BE^{\sqsubseteq}(\bar{q}, b^{DM}) = true$.

An *abstract state transformer* for the $DMLru_{A}^{\sqsubseteq}$ domain is an operator that takes in input an abstract state $\bar{q} \in DMLru_{A}^{\sqsubseteq}$ and any number of additional parameters, and returns in output a transformed state $\bar{q}' \in DMLru_{A}^{\sqsubseteq}$. We consider and define two abstract transformers for $DMLru_{A}^{\sqsubseteq}$: an update transformer $U^{\sqsubseteq}(\bar{q}, a)$, and a join transformer $J^{\sqsubseteq}(\bar{q}, \bar{p})$. We use the operator $\lambda b.$ to represent an age update operation carried on each $b \in \mathcal{B}$ when considering a transformation from state $\bar{q}$ to $\bar{q}'$. This operator can be formally defined as:

$$\lambda b.\ f(\bar{q}(b)) := \forall b \in \mathcal{B}, \bar{q}'(b) \leftarrow f(\bar{q}(b)) \tag{5}$$

Must-analysis Update

The update abstract transformer for the *must*-analysis $U^{\sqsubseteq}(\bar{q}, a)$ is used to go from an initial abstract state, to a new abstract state after a new memory access has been performed. $U^{\sqsubseteq}(\bar{q}, a)$ takes in input an initial abstract state $\bar{q}$ and a memory block $a \in \mathcal{B}$, and returns the abstract state that results from accessing a. For ease of notation, we split the definition of $U^{\sqsubseteq}$ in two parts: the logic that corresponds to the update operation when a DM block a^{DM} is accessed, indicated with $U_{D}^{\sqsubseteq}$; and the update transformation when a BE block a^{BE} is accessed, namely $U_{B}^{\sqsubseteq}$. $U_{D}^{\sqsubseteq}$ is defined in Equation 6.

$$U_{D}^{\sqsubseteq}(\bar{q}, a^{DM}) :=$$

$$D' \leftarrow \begin{cases} D + 1 & \text{if } D < A \wedge BE^{\sqsubseteq}(a) & \text{(a.1)} \\ D & \text{if } D = A \vee DM^{\sqsubseteq}(a) & \text{(a.2)} \end{cases}$$

$$
\lambda b. \begin{cases}
0 & \text{if } b = a & \text{(b)} \\
\bar{q}(b) & \text{if } b \neq a \wedge \left\| \begin{array}{l} BE^{\sqsubseteq}(b) \wedge DM^{\sqsubseteq}(a) \\ DM^{\sqsubseteq}(b) \wedge \bar{q}(a) \leq \bar{q}(b) \\ BE^{\sqsubseteq}(b) \wedge BE^{\sqsubseteq}(a) \wedge \bar{q}(a) \leq \bar{q}(b) \end{array} \right. & \begin{array}{l} \text{(c.1)} \\ \text{(c.2)} \\ \text{(c.3)} \end{array} \\
\bar{q}(b)+1 & \text{if } b \neq a \wedge \bar{q}(a) > \bar{q}(b) \wedge \\
& \quad \left\| \begin{array}{l} DM^{\sqsubseteq}(b) \wedge \bar{q}(b) < D' - 1 \\ BE^{\sqsubseteq}(b) \wedge BE^{\sqsubseteq}(a) \wedge \bar{q}(b) < A - 1 \end{array} \right. & \begin{array}{l} \text{(d.1)} \\ \text{(d.2)} \end{array} \\
\infty & \text{if } b \neq a \wedge \bar{q}(a) > \bar{q}(b) \wedge \\
& \quad \left\| \begin{array}{l} DM^{\sqsubseteq}(b) \wedge \bar{q}(b) \geq D' - 1 \\ BE^{\sqsubseteq}(b) \wedge BE^{\sqsubseteq}(a) \wedge \bar{q}(b) \geq A - 1 \end{array} \right. & \begin{array}{l} \text{(e.1)} \\ \text{(e.2)} \end{array}
\end{cases} \tag{6}
$$

Here, D' (B', resp.) is the new value of D (B, resp.) after the update. The conditions following the $||$ operator are to be considered in logical "or" with each other.

The update abstract transformer $U_B^{\sqsubseteq}$ for a best-effort memory access a can be defined as follows:

$$
U_B^{\sqsubseteq}(\bar{q}, a^{BE}) :=
$$
$$
\lambda b. \begin{cases}
D & \text{if } b = a \wedge D < A & \text{(a)} \\
\bar{q}(b) & \text{if } b \neq a \wedge \left\| \begin{array}{l} DM^{\sqsubseteq}(b) \\ BE^{\sqsubseteq}(b) \wedge \bar{q}(a) \leq \bar{q}(b) \end{array} \right. & \text{(b)} \\
\bar{q}(b)+1 & \text{if } b \neq a \wedge BE^{\sqsubseteq}(b) \wedge \bar{q}(a) > \bar{q}(b) \wedge \bar{q}(b) < A - 1 & \text{(c)} \\
\infty & \text{if } \left\| \begin{array}{l} b = a \wedge D \geq A \\ b \neq a \wedge BE^{\sqsubseteq}(b) \wedge \bar{q}(a) > \bar{q}(b) \wedge \bar{q}(b) \geq A - 1 \end{array} \right. & \text{(d)}
\end{cases} \tag{7}
$$

To clarify the update operation, consider the abstract state $\bar{q} = [\{\}, \{b, f\}], [\{c\}, \{d\}]$, where $D = 2$. Assume that deterministic block a^{DM} is accessed, which has age upper-bound ∞ in $\bar{q}$, to obtain $\bar{q}' = U^{\sqsubseteq}(\bar{q}, a) = U_D^{\sqsubseteq}(\bar{q}, a)$. First, the value of D' is computed as $D' = D + 1 = 3$ (a.1); next, b, f both satisfy the condition $\bar{q}(a) > \bar{q}(b) = \bar{q}(f) = 1$. Moreover, we have that $DM^{\sqsubseteq}(b) = DM^{\sqsubseteq}(f) = \mathit{true}$, and that $\bar{q}(b) = \bar{q}(f) = 1 < D' - 1 = 2$. This corresponds to condition (d.1) in Equation 6. Hence, the age of b, f in the resulting state is $\bar{q}'(b) = \bar{q}'(f) = 2$. Similarly, block c and d satisfy condition (d.2) and (e.2), respectively. The resulting updated abstract state is: $\bar{q}' = [\{a\}, \{\}, \{b, f\}], [\{c\}]$.

An example for the abstract transformer $U_B^{\sqsubseteq}$ defined in Equation 7 is provided in Section 5.

▶ **Theorem 2** (Correctness of *must*-analysis update). *Consider a generic abstract state $\bar{p} = U^{\sqsubseteq}(\bar{q}, a^{CL})$ obtained from the* must-*analysis update state transformer when accessing a generic block a^{CL} from an initial abstract state $\bar{q}$. Then for any block $b \in \mathcal{B}$, $\bar{p}(b)$ is an upper-bound on the DM-LRU age of b.*

Proof Sketch. A proof can be constructed by considering two main sub-cases: (1) when $CL = DM$ for the block being accessed; and (2) the case when $CL = BE$. Due to space constraints, we provide an intuition for the former case, as the latter follows from the same reasoning. When considering $CL = DM$, the new state $\bar{p}$ is obtained as $\bar{p} = U_D^{\sqsubseteq}(\bar{q}, a^{DM})$, as per Equation 6.

First let us consider the rule on the update of D. If $\bar{q}(a) = \infty$ then a is not necessarily in cache and accessing a increases the upper-bound on the number of allocated DM blocks, as long as the associativity A has not been exceeded, i.e. $D < A$. In this case, note that $BE^{\sqsubseteq}(\bar{q}, a) = true$ and condition Equation 6 (a.1) applies. D does not change in any other case (a.2). After the update, block a will have age upper-bound equal to 0 (b).

Next, consider all the blocks $b \neq a$ that had age upper-bound of infinity in $\bar{q}$ – i.e. $\bar{q}(b) = \infty$, and $BE^{\sqsubseteq}(\bar{q}, b) = true$. When a is accessed, their age upper-bound should not change. If $\bar{q}(a) = \infty$ then condition (c.3) applies. If $\bar{q}(a) \neq \infty$ then $DM^{\sqsubseteq}(\bar{q}, a) = true$ and condition (c.1) applies.

Furthermore, consider all the blocks $b^{DM}, b \neq a$ that must be allocated as DM blocks in $\bar{q}$, i.e. such that $DM^{\sqsubseteq}(\bar{q}, b) = true$. If $\bar{q}(a) = \infty$, the upper-bound on their DM-LRU age will have to increase by 1 (d.1). If however the value of $\bar{q}(b) + 1$ exceeds the updated value of D, namely D', then the block may be evicted and the new upper-bound on its DM-LRU age $\bar{p}(b) = \infty$ (e.1). The same cases apply when $\bar{q}(a) < \infty$ and $\bar{q}(a) > \bar{q}(b)$.

On the other hand, if a has an age upper-bound that is same as or lower than b's, i.e. $\bar{q}(a) \leq \bar{q}(b)$, then a concrete state where DM-LRU age of a is strictly larger than that of b cannot exist. As such, the upper-bound on the DM-LRU age of b will not change, as per condition (c.2).

Lastly, consider all the blocks $b^{BE}, b \neq a$ that must be allocated as BE blocks in $\bar{q}$, i.e. such that $BE^{\sqsubseteq}(\bar{q}, b) = true$ and $\bar{q}(b) < \infty$. The only case in which $\bar{q}(a) > \bar{q}(b)$ is if $\bar{q}(a) = \infty$. When a is accessed, the upper-bound on the age of b will have to increase by 1 (d.2), unless by doing so the associativity A is exceeded. In the latter case, $\bar{p}(b) = \infty$ (e.2). ◀

Must-analysis Join

The join abstract transformer $J^{\sqsubseteq}(\bar{q}, \bar{p})$ is used to compute a new abstract state at the merging point of two or more execution paths. There are strong similarities between the transformer defined hereby and what used in traditional LRU *must*-analysis [15]. At a high level, the joined state will consider as *must*-cached only those blocks in the intersection of the joining states, each with the maximum age in any of the two states. For the new state, D is taken as the maximum between the value of D in the joining states. Equation 8 formalizes the $J^{\sqsubseteq}(\bar{q}, \bar{p})$ abstract transformer:

$$J^{\sqsubseteq}(\bar{q}, \bar{p}) := D \leftarrow \max\{D_{\bar{q}}, D_{\bar{p}}\}, \lambda b. \max\{\bar{q}(b), \bar{p}(b)\} \tag{8}$$

If we were to join $\bar{q} = [\{\}, \{b, f\}], [\{c\}, \{d\}]$ with $\bar{q}' = [\{a\}, \{\}, \{b, f\}], [\{c\}]$, the resulting state would be $\bar{q}'' = J^{\sqsubseteq}(\bar{q}, \bar{q}') = [\{\}, \{\}, \{b, f\}], [\{c\}]$.

▶ **Theorem 3** (Correctness of *must*-analysis join). *Consider an abstract state $\bar{s} = J^{\sqsubseteq}(\bar{q}, \bar{p})$ obtained from the* must-*analysis join state transformer from two initial abstract states $\bar{q}$ and $\bar{p}$. Then for any block $b \in \mathcal{B}$, $\bar{s}(b)$ is an upper-bound on the DM-LRU age of b.*

Proof Sketch. A proof can simply follow from the definition of the $J^{\sqsubseteq}$ operator in Equation 8. By hypothesis $\bar{q}$ and $\bar{p}$ carry the upper-bound on the age of a generic block b along two disjoint execution sub-paths. After the two sub-paths join, the maximum between $\bar{q}(b)$ and $\bar{p}(b)$ is a safe upper-bound on the DM-LRU age of b in the resulting abstract state $\bar{s}$. Moreover, an upper-bound on the number of allocated DM blocks in $\bar{s}$ is the maximum between $D_{\bar{q}}$ and $D_{\bar{p}}$. ◀

Must-analysis Classification

Every time an access is performed, it is possible to classify a memory access using a classification function that will either return M for cache miss, H for cache hit, or $\top$ in case neither M nor H classification can be made given the current abstract state. In order to

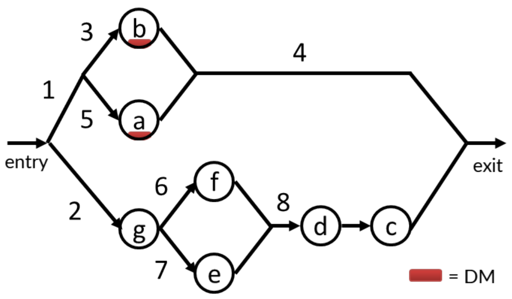


Figure 3 Fragment of process CFG. At the end of the fragment, all the cache blocks in the figure *may* be cached.

classify memory accesses, for a given $\bar{q}$ abstract state we define two helper sets $\bar{\mathcal{D}}$ and $\bar{\mathcal{B}}$ representing the deterministic and best-effort memory blocks that have finite upper bound on their DM-LRU age:

$$\begin{aligned}&\bar{\mathcal{D}} := \{b^{CL} \in \mathcal{B} \mid CL = DM \wedge \bar{q}(b) < \infty\}\\ &\bar{\mathcal{B}} := \{b^{CL} \in \mathcal{B} \mid CL = BE \wedge \bar{q}(b) < \infty\}\end{aligned} \tag{9}$$

The classification function of the *must* analysis is defined as:

$$C^{\sqsubseteq}(\bar{q}, a^{CL}) := \begin{cases} H & \text{if } \bar{q}(a) < \infty & \text{(a)} \\ M & \text{if } \left\| \begin{array}{l} CL = DM \wedge a \not\in \bar{\mathcal{D}} \wedge |\bar{\mathcal{D}}| = D \\ CL = BE \wedge a \not\in \bar{\mathcal{B}} \wedge |\bar{\mathcal{B}}| = B \end{array} \right. & \text{(b)} \\ \top & \text{otherwise} & \text{(c)} \end{cases} \tag{10}$$

We provide a complete step-by-step example on how *must*-analysis can be applied to an application's CFG in Section 5.

4.2 May-analysis

The complete *may*-analysis is provided in the appendix (Section A). We hereby provide a sketch of the approach followed in the analysis.

The goal of *may*-analysis is to track the lower-bound on the age of memory blocks. Given a *may*-analysis abstract state it is possible to classify a memory access as always leading to a miss. Let us consider the example in Figure 3 and reason on the lower bound on the age of each block for a 4-way fully associative cache. For block a^{DM}, the best case is represented by the execution pattern 1-5-4. In this case, the block has DM-LRU age 0. A similar situation occurs for block b^{DM} and path 1-3-4. For blocks f and g, the best-case is represented by the paths 2-6-8, and 2-7-8, respectively. This leads the two blocks to have a lower-bound of 2 on their DM-LRU age. Similarly, blocks c, d, and g have lower-bound 0, 1, and 3, respectively.

We can represent the resulting *may*-analysis state obtained following the derivation above as: $[\{a, b\}], [\{c\}, \{d\}, \{f, e\}, \{g\}]$. What happens if another access to a occurs after path 4 and 8 join? Then the best-case for block b is still 1-3-4, but its age lower-bound will be 1. At the same time, because at least one DM block was allocated regardless of the taken path, the minimum lower-bound on the age of any BE block has to be 1. Also note that regardless of the execution path taken, block g will be evicted. The result is the following *may*-analysis abstract state: $[\{a\}, \{b\}], [\{\}, \{c\}, \{d\}, \{f, e\}]$.

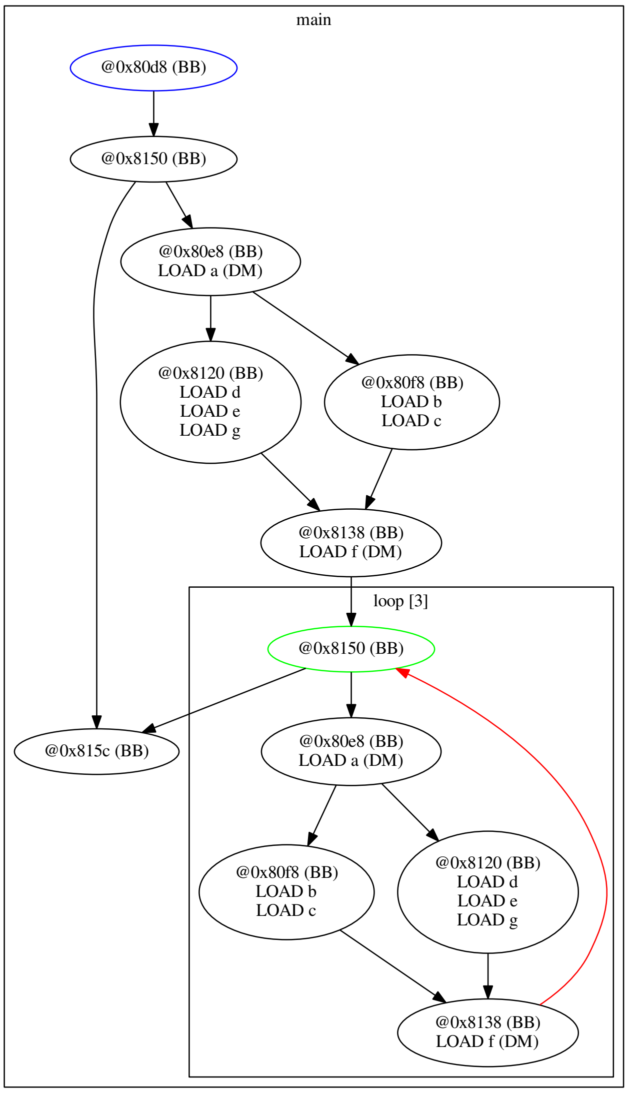


Figure 4 Original CFG of considered example as rendered by the Heptane tool, with annotated memory accesses (LOAD). Note that VIVU has been performed on the loop.

5 Analysis Example

In this section we provide a description of how DM-LRU *must-analysis* can be applied to a CFG once the target of each memory access is known. The original CFG of the considered C program code generated by the Heptane tool is shown in Figure 4. The program consists of a single loop with four iterations, where the first iteration has been unrolled. The program accesses 7 memory locations. These are $\mathcal{B} = \{a, b, c, d, e, f, g\}$ and are visible in the various basic blocks as operands of load/store instructions.

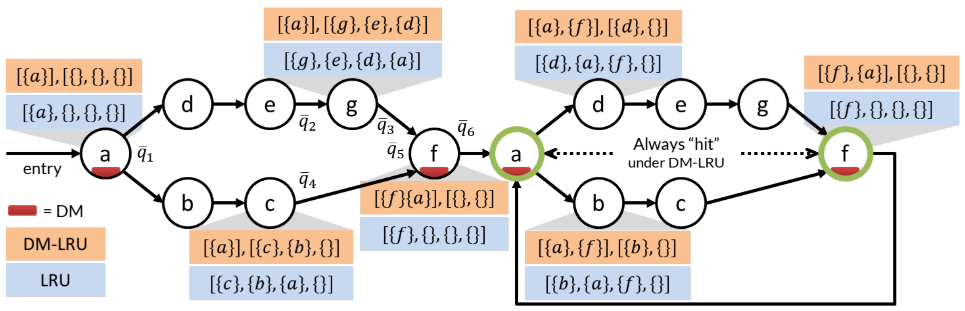


Figure 5 An example of *must* analysis under DM-LRU (orange states), compared to traditional LRU (blue states). If a and f are marked as DM, their accesses inside the loop can be classified as always hits.

Figure 5 shows the same CFG as in Figure 4, but where only basic blocks in which memory accesses are being performed are kept. Moreover, basic blocks with multiple memory accesses are depicted as sequences of blocks, each with a single memory access. The nodes are annotated with their corresponding abstract states. We apply *must*-analysis starting from the entry node a. We compare the behavior of traditional LRU analysis and DM-LRU when blocks a and f have been declared as DM. We consider a fully-associative cache with 4 ways. For DM-LRU analysis, the cache state before the first access $\bar{q}_0 = [], [\{\}, \{\}, \{\}, \{\}]$ ($D_0 = 0$); for LRU analysis it is $[\{\}, \{\}, \{\}, \{\}]$. Under DM-LRU, when block a^{DM} is accessed, the performed operation is $\bar{q}_1 = U^{\sqsubseteq}(\bar{q}_0, a_{DM}) = U^{\sqsubseteq}_{\bar{D}}(\bar{q}_0, a)$. Following Equation 6, we have $D_1 = D_0 + 1$, then condition (a) is satisfied by a, all the other blocks $b \in \mathcal{B}$ satisfy condition (d.2). As such, we have $\bar{q}_1 = [\{a\}], [\{\}, \{\}, \{\}]$, as reported in the figure.

Let us now follow the upper branch with access sequence $d \to e \to g$ (all of them are best-effort memory accesses). For each memory access, we apply Equation 7 to obtain a new abstract state. After accessing e, the resulting abstract state is: $\bar{q}_2 = [\{a\}][\{e\}, \{d\}, \{\}]$. Let us now show more clearly how we obtain $\bar{q}_3 = U^{\sqsubseteq}(\bar{q}_2, g^{BE}) = U^{\sqsubseteq}_{\bar{B}}(\bar{q}_2, g)$, when we next access g. Considering all blocks in $\mathcal{B}$ and using Equation 7 we know: block a satisfies condition (b.1) and its age remains the same; b and c satisfy (d.2) and their age remains ∞; block e satisfies (c) and its age increases by 1, from 1 to 2; the age of block d increases from 2 to 3; and finally, block g (being accessed) satisfies condition (a) and its age is set to $D_2 = 1$. The final state is $\bar{q}_3 = [\{a\}], [\{g\}, \{e\}, \{d\}]$, as shown in the figure above node g. The same procedure applies to the lower branch of the CFG, and we obtain the state $\bar{q}_4 = [\{a\}], [\{c\}, \{b\}, \{\}]$, after we access c.

Before accessing f, we need to join states $\bar{q}_3$ and $\bar{q}_4$ derived above. In this case, we apply Equation 8 to obtain $\bar{q}_5 = J^{\sqsubseteq}(\bar{q}_3, \bar{q}_4)$. It follows that $D_5 = 1$. Moreover, the only block present in both states $\bar{q}_3$ and $\bar{q}_4$ is a. All other blocks in $\mathcal{B}$ will have age ∞ in $\bar{q}_5$. As such we have $\bar{q}_5 = [\{a\}], [\{\}, \{\}, \{\}]$. Next, accessing f^{DM} yields $\bar{q}_6 = U^{\sqsubseteq}_{\bar{D}}(\bar{q}_5, f) = [\{f\}, \{a\}], [\{\}, \{\}]$, as shown in the figure. This is because $D_6 = D_5 + 1$, and because a satisfies condition (c.1) in Equation 6. The same reasoning can be applied to obtain the remaining states depicted in the figure.

Consider now the state $\bar{q}_6$ and apply the *must*-analysis classifier before accessing a^{DM}, i.e. compute $C^{\sqsubseteq}(\bar{q}_6, a)$ as in Equation 10. First, the sets $\bar{\mathcal{D}}_6$ and $\bar{\mathcal{B}}_6$ can be computed using Equation 9 as $\bar{\mathcal{D}}_6 = \{a, f\}$, and $\bar{\mathcal{B}}_6 = \{\}$. Hence condition (a) is satisfied and access to a is classified as H (hit). Conversely, no access can be classified as hit under LRU.

6 Analogies and Differences with Cache Locking

Cache locking refers to a technique where cache blocks that are deemed important for an application's timing are *pinned* (locked) in cache. Similar to DM-LRU, cache locking represents a way to partially override the best-effort replacement strategy offered by the hardware. And like DM-LRU, specialized hardware support is required to perform locking. With respect to WCET analysis, the big advantage provided by cache locking is that all those accesses for locked cache blocks can be immediately classified as hits. While cache locking was commonly supported in previous-generation embedded systems, the current trend in embedded SoCs is toward cache controllers that offer little or no management primitives.

Despite the strong similarities, some profound differences exist between cache locking and DM-LRU. Leveraging cache locking implies injection of additional logic – in either the application, the compiler, and/or the OS – to perform a series of prefetch&lock operations. On the contrary, a system featuring a DM-LRU cache only requires that memory blocks are tagged appropriately at task load time.

In case of static locking, prefetch&lock can be performed at initialization time. As such, the extra logic required to perform locking does not impact the task's WCET. Conversely, with dynamic cache locking, the locked cache content is modified at runtime. Depending on the available hardware support, this operation may not be directly possible in user-space, requiring instead a costly switch to kernel-space. Regardless, an online prefetch&lock routine can pollute the rest of the cache, resulting in overheads that may largely offset any benefit. In other words, additional system-level assumptions are required to make a meaningful comparison with dynamic locking. For this reason, we do not compare DM-LRU against dynamic locking.

Interestingly enough, however, the proposed DM-LRU analysis can be re-used to analyze dynamic locking schemes if additional system parameters are available. In a nutshell, consider a 4-way fully associative cache. Next, assume that the locked content is switched whenever a given branch in the CFG is taken. Then, consider the case where the new content to be locked is comprised of blocks a, b, c. A special node on the branch can be added with associated a modified update abstract transformer $Lock^{\sqsubseteq}$. This is such that the resulting *must*-analysis abstract state $\bar{q}$ after the update is: $\bar{q} = Lock^{\sqsubseteq}(\{a, b, c\}) = [\{a\}, \{b\}, \{c\}][\{\}]$.

7 Evaluation

The DM-LRU analysis presented in the previous sections provides a way to understand how the WCET of applications varies as memory blocks addressed in applications are declared as DM. We now apply DM-LRU analysis to a set of realistic embedded benchmarks. In this section, we first briefly describe our implementation. Next, we investigate three main aspects: (1) what is the degree of WCET improvement that can be achieved via DM-LRU when compared to LRU? (2) Is there an advantage in imposing a limit to the number of DM lines that can be simultaneously allocated, i.e. in preventing DM takeover? (3) how does DM-LRU compares to static cache locking?

In our evaluation we focus on the degree of WCET improvement that DM-LRU can provide compared to LRU. However, because supporting DM-LRU implies changes to the hardware cache memory and controller, it is also important to determine if a DM-LRU implementation can be efficiently carried out. In short, only one additional bit to distinguish between DM and BE lines is required per cache line. Additionally, compact changes[1] are required to the cache controller to restrict victim selection based on the classification (DM or BE) of a new line being allocated. No additional logic is required to appropriately set the DM bits at line fetch. Additional considerations on the incurred hardware costs to support DM memory are also provided in [10].

7.1 Implementation

We have implemented support for DM-LRU inside a state of the art static WCET analysis tool, namely Heptane [23]. Heptane implements Implicit Path Enumeration Technique (IPET) [29] and performs analysis for many cache architectures: e.g., LRU, FIFO, Pseudo-LRU, multi-level non-inclusive caches, and shared caches. In our setup, we consider a single-level of cache, divided into an instruction (I) cache, and a data (D) cache. For simplicity, we assume in all our experiments that both caches are selected to have the same number of sets and ways. Heptane supports two architectures: ARMv7 and MIPS. The ARMv7 target was used for this paper.

We have modified the Heptane tool to support two variants of DM-LRU, as well as static locking. Most importantly, we have extended the support for abstract interpretation-based cache analysis to implement the *must*- and *may*-analysis presented in the previous sections. The performed changes allow backward compatibility with the original set of policies supported by the tool. Next, we have integrated the logic to differentiate between BE memory and DM memory. For this purpose, we have added a table of DM addresses – the DM Table – that can be specified by an external tool, mimicking the selection of DM blocks by the OS at binary load time. Furthermore, we have added appropriate logic in Heptane to output per-memory-block statistics in terms of references, hits, and misses, as computed during WCET analysis. These statistics are then used to build a DM-block selection heuristic. Finally, we have modified Heptane's CFG extraction routines to perform VIVU – i.e., to recursively peel the first iteration of every loop.

We have developed and employed a simple heuristic to determine which memory blocks/addresses should be marked as DM and inserted into the DM Table. The heuristic initially performs WCET analysis without selecting any DM line. Next, it analyzes the per-memory-block statistics and selects as DM the block with the largest number of misses. At this point, WCET analysis is performed again with the new DM Table containing a single entry. Using the per-memory-block statistics of the latest run, a new DM block is selected in addition to the previously selected block. The same steps are performed until no more addresses can be selected as DM. Note that when no lines are selected as DM, the behavior of the cache is indistinguishable from vanilla LRU. Similarly, when the entire memory of an applications is selected as DM, no differences exist with LRU. In practice, however, we saw no differences between DM-LRU and LRU when more than $3 \times S \times A$ lines are selected as DM, where S and A is the number of sets and ways of the cache, respectively. In our experiments, we acquired analytical results for a number of DM lines in the range $[1, 3 \times S \times A]$.

[1] Whenever a line eviction has to occur, the DM/BE bits of all the lines in the considered set form a bitmask. Victim selection for a BE access is then performed by excluding all those lines that have a bit set to 1 in the DM bitmask.

7.2 Setup

We compare two variants of DM-LRU and static cache locking against LRU. A description of the three scenarios follows.

1. **Unrestricted DM-LRU ("DM-nolim")**: in this variant, no restriction is imposed on the maximum number of cache sets that can be reserved for DM lines. It follows that the only constraint for the allocation of DM lines is the cache associativity itself. The analysis for unrestricted DM-LRU is the one presented in the previous sections.
2. **Limited DM-LRU ("DM-cap")**: in this variant, a hard cap in the maximum number of ways is imposed on the expansion of DM lines. This represents a solution to the aforementioned problem of *DM takeover*. Imposing a cap of 0 makes DM-cap to be identical to vanilla LRU. Similarly, imposing a *cap* of A makes DM-cap to be identical to DM-nolim. In our experiments, we explore all the possible values of *cap* in the range $[1, A]$.
3. **Static locking ("Static")**: this case is used to draw a comparison between the considered DM-LRU variants and static locking. In case of static locking, selection of lines to statically allocate is performed following the same heuristic used for DM lines selection. Similar to DM-cap, we impose how many ways can be dedicated to statically locked content (locked ways). The maximum number of allocated line is then $S \times locked$. Note that the main performance difference between DM-cap and Static lies in the additional flexibility that DM-cap provides. In DM-cap, in fact, more lines than $S \times cap$ can be selected, while it is not allowed in static locking.

For all the considered variants, we explore a number of cache configurations. Specifically, we vary the associativity A of the I/D caches in the set $\{2, 4, 8, 16\}$. We vary the number of cache sets S such that $S \in \{2, 4, 8, 16, 32\}$. As previously mentioned, for DM-nolim and DM-cap, we progressively select up to $3 \times S \times A$ DM lines following the heuristic described above. In each system instance, we perform WCET analysis using the modified Heptane tool. Then, we keep track of which configuration – S, A, DM-lim *cap*, *locked* ways, number of DM lines – for each of the three scenarios leads to the best reduction in WCET compared to the vanilla LRU case.

For our benchmarks, we use a subset of realistic benchmarks from the Mälardalen suite [19]. Unfortunately, vanilla HEPTANE is not able to perform WCET analysis for some of the benchmarks in the suite. As such, our evaluation only includes those benchmarks that are correctly handled by HEPTANE. Notably, the aforementioned changes to implement DM-LRU analysis did not impact the set of benchmarks that can be correctly analyzed by the tool.

7.3 Results

Figure 6 provides an overview of the obtained results. A cluster of bars is provided for each of the considered benchmarks. Reading the plot from top to bottom, the first bar corresponds to the WCET under LRU. All the results in the figure are normalized to the LRU case. The second bar represents the best WCET improvement that was observed under DM-nolim. The WCET improvement is calculated as: $\frac{WCET_{\text{DM-nolim}}}{WCET_{\text{LRU}}}$, where the WCETs under DM-nolim and under LRU are obtained in the same system configuration. A similar calculation was performed to derive the remaining two bars, i.e. for the DM-lim and Static cases.

What emerges from the plot is that in 16 out of 20 cases, DM-nolim is able to achieve WCET reductions compared to vanilla LRU. Notably, in case of `bsort100` and `prime`, it is possible to achieve a WCET reduction of around 23.73% and 23.47%, respectively. It can

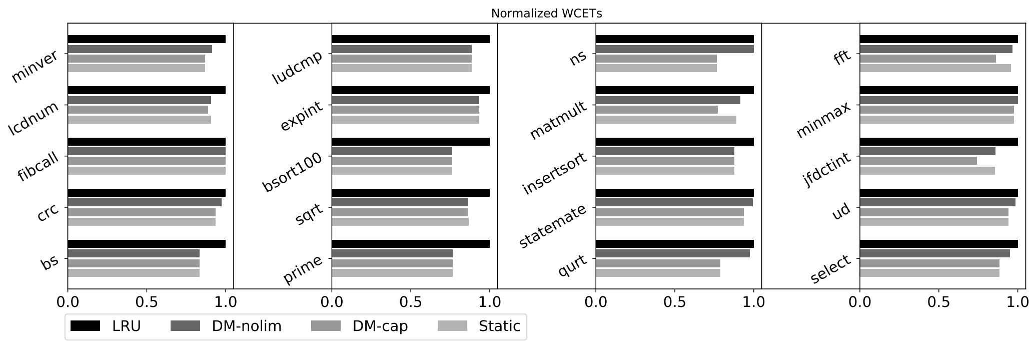


Figure 6 Computed WCETs for vanilla LRU (LRU), unrestricted DM (DM-nolim), DM limited to a subset of ways (DM-cap), and static locking (Static).

also be noted that DM-cap outperforms DM-nolim. Moreover, DM-cap performs generally better than static locking. For instance, the best WCET reduction achieved via static locking for the `jfdctint` benchmark is 26.09% under DM-cap (with a $S = 4$, $A = 8$, 19 DM lines, $cap = 1$). But the best WCET reduction under static locking is only 14.64%, which is achieved for a cache with parameters $S = 2$, $A = 16$ and 15 ways occupied by statically locked lines. Similar results can be observed for the benchmarks `matmult` and `fft`.

The reason for the performance improvement that can be obtained with DM-cap is twofold. On the one hand, the problem of DM takeover is solved. This prevents the case that all the accesses to BE lines result in misses. On the other hand, for applications that exhibit changes in working sets, static locking can be sub-optimal. Conversely, under DM-cap, is is possible to mark lines belonging to different working sets as DM. In this case, at working set changes over time, those DM lines belonging to a previous working set will be naturally evicted, without suffering pollution from BE lines.

A more detailed overview of the obtained experimental results is provided in Table 1. In the table, the first column reports the name of the benchmark under analysis. If multiple configurations are of interest, multiple rows are shown for a given benchmark. The second column reports the cache configuration in terms of sets S and ways W for the results on each row. Next, the WCET obtained with LRU is reported in the following column, followed by the best WCET obtained for the same configuration under DM-nolim and the relative improvement (due to the space limitation, the number of DM lines that were selected has been omitted in the table.) Similarly, the best result obtained under DM-cap is reported next, and the value of *cap* under which the result was achieved is reported in the adjacent column. Finally, the last two columns report the WCET (and the relative improvement) for static locking with the given cache configuration and number of locked ways reported in the last column.

8 Related Work

Memory Tagging and Hardware Support. In this work, we assume that hardware allows us to encode (tag) extra information (e.g., importance) on memory locations at a fine-granularity. The basic idea of memory tagging has first explored in the security community, to prevent memory corruption (e.g., buffer overflow) [6, 38] and to enforce data flow integrity [45] and capability protection [51]. Efficient hardware designs for word-granularity single-bit and multi-bit memory tagging have been investigated [24] and several real SoC prototypes have been built [45, 4], demonstrating the feasibility. In the real-time systems community, several

Table 1 Summary of notable experimental results under four strategies: (1) vanilla LRU ("LRU"); (2) unrestricted DM-LRU ("DM-nolim"); (3) restricted DM-LRU ("DM-cap"); and (4) static locking ("Static").

Benchmark	*S*×*A*	**LRU**	**DM-nolim**	**DM-cap**	*cap*	**Static**	*locked*
bs	2×2	6613	5513 (-16.63%)	5513 (-16.63%)	2	5513 (-16.63%)	2
crc	4×2	2492320	2425920 (-2.66%)	2330620 (-6.49%)	1	2330620 (-6.49%)	1
fibcall	2×2	14191	14191 (-0.00%)	14191 (-0.00%)	1	14191 (-0.00%)	1
lcdnum	4×2	16291	14791 (-9.21%)	14791 (-9.21%)	2	14791 (-9.21%)	2
	2×4	16191	16191 (-0.00%)	14391 (-11.12%)	2	15291 (-5.56%)	2
minver	4×2	126558	115758 (-8.53%)	109958 (-13.12%)	1	109958 (-13.12%)	1
prime	2×4	611425	467925 (-23.47%)	467925 (-23.47%)	3	467925 (-23.47%)	3
sqrt	2×4	54983	47552 (-13.52%)	47252 (-14.06%)	3	52552 (-4.42%)	2
	2×4	54983	47552 (-13.52%)	47252 (-14.06%)	3	47583 (-13.30%)	6
bsort100	2×2	12434700	9484580 (-23.72%)	9484580 (-23.72%)	1	9484580 (-23.72%)	1
expint	2×4	759551	709651 (-6.57%)	709651 (-6.57%)	4	709651 (-6.57%)	4
ludcmp	16×2	638233	564633 (-11.53%)	564633 (-11.53%)	2	564633 (-11.53%)	2
qurt	2×8	217555	212160 (-2.48%)	173755 (-20.13%)	6	173755 (-20.13%)	6
	4×4	217555	220355 (–1.29%)	171155 (-21.33%)	3	171155 (-21.33%)	3
statemate	2×2	616218	612918 (-0.54%)	576718 (-6.41%)	1	576718 (-6.41%)	1
	8×8	383718	382818 (-0.23%)	359118 (-6.41%)	6	359118 (-6.41%)	6
insertsort	2×2	80126	70126 (-12.48%)	70126 (-12.48%)	1	70126 (-12.48%)	1
matmult	2×2	7191620	6568220 (-8.67%)	5555520 (-22.75%)	1	6391620 (-11.12%)	1
ns	4×2	193481	193481 (-0.00%)	193481 (-0.00%)	1	193481 (-0.00%)	1
	2×2	530781	534781 (–0.75%)	406681 (-23.38%)	1	406681 (-23.38%)	1
select	4×2	170766	162266 (-4.98%)	157966 (-7.50%)	1	157966 (-7.50%)	1
	2×4	170766	162866 (-4.63%)	150966 (-11.59%)	3	150966 (-11.59%)	3
ud	4×2	226843	223243 (-1.59%)	223243 (-1.59%)	2	225243 (-0.71%)	2
	2×2	302443	354143 (–17.09%)	283843 (-6.15%)	1	283843 (-6.15%)	1
jfdctint	2×16	150234	128734 (-14.31%)	111034 (-26.09%)	2	128234 (-14.64%)	15
	4×8	150234	130334 (-13.25%)	111134 (-26.03%)	1	147134 (-2.06%)	1
	4×8	150234	130334 (-13.25%)	111134 (-26.03%)	1	130334 (-13.25%)	7
minmax	2×2	4034	4034 (-0.00%)	4034 (-0.00%)	1	4034 (-0.00%)	1
	2×4	4034	4034 (-0.00%)	3934 (-2.48%)	1	4034 (-0.00%)	1
fft	32×2	1683830	1623930 (-3.56%)	1623430 (-3.59%)	1	1623430 (-3.59%)	1
	4×4	2488230	2494360 (–0.25%)	2140830 (-13.96%)	1	2443230 (-1.81%)	1
	4×4	2488230	2494360 (–0.25%)	2140830 (-13.96%)	1	1716630 (-4.62%)	2

works explored the use physical memory address based differentiated hardware designs (mostly cache) in a more coarse-grained manner (i.e., memory segments, page, and task granularity). Kumar et al, proposed a criticality-aware cache design, called Least Critical (LC), which includes a memory criticality-aware extension to LRU replacement policy [27]. The LC cache's replacement policy is similar to the replacement policy we assumed in this work (Algorithm 1), while its memory tagging mechanism, which uses a fixed number of specialized range registers, does not allow flexible and fine-grained memory tagging. Therefore, our static analysis method can be directly applicable to analyze the LC cache. PRETI [28] also proposes a criticality-aware cache design but it focuses on shared cache for SMT hardware, while we focus on private caches. More recently, OS-level page-granularity memory tagging and supporting multicore architecture designs (including a new cache design) have been explored to provide efficient hardware isolation (incl. cache isolation) in multicore [10].

Static Cache Analysis. There exists a broad literature on static cache analysis [32, 50]. With respect to existing literature, this work is closely related to approaches that propose abstract interpretation-based cache analysis. This approach was initially proposed in [1, 12]. These works illustrate LRU analysis and hit/miss classification using *may*- and *must*-analysis.

The work in [12] also proposes a persistence analysis based on abstract states, which was found to be unsafe and for which a fix was proposed in [8, 25]. We base our DM-LRU extension on the *may*- and *must*-analysis proposed in [12], but use the improved formalization in [15]. In order to perform access classification in case of loops we use an approach similar to *virtual inlining & virtual unrolling* (VIVU) originally proposed in [34]. A large body of works has considered cache replacement policies other than LRU. These include FIFO [15, 14, 18], MRU [17], Pseudo-LRU [16]. Comparatively less work has been produced to analyze non-inclusive [36, 21] as well as inclusive [22] multi-level caches. With respect to these works, the proposed methodology set itself apart because it focuses on the impact on the WCET of designer-driven selection of frequently accessed memory blocks. In this sense, proposed approach can be used to analyze caches that support the definition of touch-and-lock cache lines, under the assumption that no more than A blocks are simultaneously locked on any set, where A is the associativity of the cache.

Cache Locking and Scratchpad Memory. Some COTS cache designs [2, 7, 13] support selective cache locking, which prevents evictions for certain programmer selected cache-lines. Exploting the feature, various static and dynamic cache locking schemes for both instruction and data caches have been investigated [5, 40, 39, 47, 35]. In [48, 47], for instance, cache locking statements are inserted in the task's execution flow at compilation time, when the uncertainty about the memory locations being accessed negatively impacts the static WCET analysis. Some recent works combined cache locking with cache partitioning to improve task WCET in multicore [30, 43, 33]. As an alternative to cache, scratchpad memory has received significant attention in the real-time systems community for its predictability benefits [46, 9, 49, 44]. More recently, a technique called invalidation-driven allocation (IDA) [26] was proposed to achieve the same level of determinism of a locked cache in spite of lack of hardware-assisted locking primitives. IDA can be used as long as precise invariants on the size of an application's working set hold. To overcome its high programming complexity, however, many researchers proposed various compiler-based techniques. In [44], for instance, a sophisticated compiler-based technique is proposed to break each task into intervals and at the beginning of each interval, the required memory blocks of the interval are prefetched onto a scratchpad memory via a DMA controller without blocking the task execution. Dividing a task into a sequence of well-defined memory and computation phases was originally proposed in [37, 52]. In both cache locking and scratchpad memory based techniques, a common limitation is the overhead of explicitly executing additional instructions (prefetch, lock/unlock, or data movement to/from scratchpad). Furthermore, these additional instructions are context sensitive in the sense that they must be executed before actual accesses occur, and if they are executed too early, they can negatively impact both performance and WCET. In contrast, our approach is context insensitive in the sense that, once DM blocks are flagged, actual allocation and replacement are automatically performed by the hardware (cache controller) without additional instruction execution overhead.

9 Conclusion

In this paper, we presented the DM-LRU cache replacement policy and proposed an abstract interpretation-based analysis for DM-LRU. We implemented the proposed analysis and DM-LRU support in the Heptane static WCET analysis tool. Using the Heptane, we evaluated the WCET impacts of our DM-LRU based approach on a number of benchmarks. The results show that our DM-LRU approach can provide lower task WCETs with less performance overhead and programming complexity, compared to the standard LRU and cache locking based approaches.

References

1 Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache Behavior Prediction by Abstract Interpretation. In *Proceedings of the Third International Symposium on Static Analysis*, SAS '96, pages 52–66, Berlin, Heidelberg, 1996. Springer-Verlag.

2 ARM. *PL310 Cache Controller Technical Reference Manual, Rev: r0p0*, 2007.

3 R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Int. Symp. Hardware/Software Codesign (CODES+ISSS)*, pages 73–78. ACM, 2002.

4 Alex Bradbury, Gavin Ferris, and Robert Mullins. Tagged memory and minion cores in the lowRISC SoC. *Memo, University of Cambridge*, 2014.

5 Marti Campoy, A Perles Ivars, and JV Busquets-Mataix. Static use of locking caches in multitask preemptive real-time systems. In *Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium)*, pages 1–6, 2001.

6 Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *Dependable Systems and Networks (DSN)*, pages 378–387. IEEE, 2005.

7 NVIDIA Corp. Variable SMP – A Multi-Core CPU Architecture for Low Power and High Performance. Technical report, Nvidia, 2011.

8 Christoph Cullmann. Cache Persistence Analysis: Theory and Practice. *ACM Trans. Embed. Comput. Syst.*, 12(1s):40:1–40:25, March 2013.

9 Jean-Francois Deverge and Isabelle Puaut. WCET-directed dynamic scratchpad memory allocation of data. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 179–190. IEEE, 2007.

10 Farzad Farshchi, Prathap Kumar Valsan, Renato Mancuso, and Heechul Yun. Deterministic Memory Abstraction and Supporting Multicore System Architecture. In *Euromicro Conf. Real-Time Syst. (ECRTS)*. IEEE, 2018.

11 Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache Behavior Prediction by Abstract Interpretation. *Sci. Comput. Program.*, 35(2-3):163–189, November 1999.

12 Christian Ferdinand and Reinhard Wilhelm. Efficient and Precise Cache Behavior Prediction for Real-TimeSystems. *Real-Time Syst.*, 17(2-3):131–181, December 1999.

13 Freescale. *e500mc Core Reference Manual*, 2012.

14 D. Grund and J. Reineke. Precise and Efficient FIFO-Replacement Analysis Based on Static Phase Detection. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 155–164, July 2010.

15 Daniel Grund. *Static Cache Analysis for Real-Time Systems: LRU, FIFO, PLRU*. epubli, 2012.

16 Daniel Grund and Jan Reineke. Toward Precise PLRU Cache Analysis. In *International Workshop on Worst-Case Execution Time Analysis (WCET)*, pages 23–35, 2010.

17 N. Guan, M. Lv, W. Yi, and G. Yu. WCET Analysis with MRU Caches: Challenging LRU for Predictability. In *Real Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, April 2012.

18 Nan Guan, Xinping Yang, Mingsong Lv, and Wang Yi. FIFO Cache Analysis for WCET Estimation: A Quantitative Approach. In *Design, Automation and Test in Europe (DATE)*, pages 296–301, San Jose, CA, USA, 2013. EDA Consortium.

19 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks – Past, Present and Future. In Björn Lisper, editor, *Procedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET'2010)*, pages 137–147, Brussels, Belgium, July 2010. OCG.

20 Sebastian Hahn and Jan Reineke. Design and Analysis of SIC: A Provably Timing-Predictable Pipelined Processor Core. In *Real-Time Systems Symposium (RTSS)*, pages 469–481. IEEE, 2018.

21 D. Hardy and I. Puaut. WCET Analysis of Multi-level Non-inclusive Set-Associative Instruction Caches. In *Real-Time Systems Symposium (RTSS)*, pages 456–466, November 2008.

22 Damien Hardy and Isabelle Puaut. WCET Analysis of Instruction Cache Hierarchies. *J. Syst. Archit.*, 57(7):677–694, August 2011.

23 Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. The Heptane Static Worst-Case Execution Time Estimation Tool. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 8 of *International Workshop on Worst-Case Execution Time Analysis*, page 12, Dubrovnik, Croatia, June 2017. `doi:10.4230/OASIcs.WCET.2017.8`.

24 Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W Moore, Alex Bradbury, Hongyan Xia, Robert NM Watson, David Chisnall, Michael Roe, Brooks Davis, et al. Efficient Tagged Memory. In *International Conference on Computer Design (ICCD)*, pages 641–648. IEEE, 2017.

25 Lei Ju, Samarjit Chakraborty, and Abhik Roychoudhury. Accounting for Cache-related Preemption Delay in Dynamic Priority Schedulability Analysis. In *Design, Automation and Test in Europe (DATE)*, pages 1623–1628, San Jose, CA, USA, 2007. EDA Consortium.

26 T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna. Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Montreal, Canada, April 2019.

27 NG Chetan Kumar, Sudhanshu Vyas, Ron K Cytron, Christopher D Gill, Joseph Zambreno, and Phillip H Jones. Cache design for mixed criticality real-time systems. In *Computer Design (ICCD)*, pages 513–516. IEEE, 2014.

28 Benjamin Lesage, Isabelle Puaut, and André Seznec. PRETI: Partitioned REal-TIme shared cache for mixed-criticality real-time systems. In *Real-Time and Network Systems (RTNS)*, pages 171–180. ACM, 2012.

29 Y. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(12):1477–1487, December 1997. `doi:10.1109/43.664229`.

30 T. Liu, Y. Zhao, M. Li, and C. J. Xue. Task Assignment with Cache Partitioning and Locking for WCET Minimization on MPSoC. In *2010 39th Int. Conf. Parallel Processing*, pages 573–582, September 2010.

31 Thomas Lundqvist and Per Stenström. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, RTSS '99, pages 12–, Washington, DC, USA, 1999. IEEE Computer Society. URL: `http://dl.acm.org/citation.cfm?id=827271.829103`.

32 Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. A survey on static cache analysis for real-time systems. *Leibniz Transactions on Embedded Systems*, 3(1):05–1, 2016.

33 R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-Time Cache Management Framework for Multi-core Architectures. In *Real-Time and Embedded Technology and Applicat. Symp. (RTAS)*. IEEE, 2013.

34 Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of Loops. In *International Conference on Compiler Construction (CC)*, pages 80–94, London, UK, UK, 1998. Springer-Verlag.

35 Sparsh Mittal. A Survey of Techniques for Cache Locking. *Transactions on Design Automation of Electronic Systems (TODAES)*, 21(3):49:1–49:24, May 2016. `doi:10.1145/2858792`.

36 Frank Mueller. Timing Predictions for Multi-Level Caches. In *In ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 29–36, 1997.

37 R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for COTS-based embedded systems. In *Real-Time and Embedded Technology and Applicat. Symp. (RTAS)*, pages 269–279. IEEE, 2011.

38 Krerk Piromsopa and Richard J Enbody. Secure bit: Transparent, hardware buffer-overflow protection. *Transactions on Dependable and Secure Computing*, 3(4):365–376, 2006.

39 Isabelle Puaut. WCET-centric software-controlled instruction caches for hard real-time systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 10–pp. IEEE, 2006.

40 Isabelle Puaut and David Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Real-Time Systems Symposium (RTSS)*, pages 114–123. IEEE, 2002.

41 Jan Reineke. *Caches in WCET analysis: predictability, competitiveness, sensitivity*. epubli, 2008.

42 Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing Predictability of Cache Replacement Policies. *Real-Time Syst.*, 37(2):99–122, November 2007. doi:10.1007/s11241-007-9032-3.

43 A. Sarkar, F. Mueller, and H. Ramaprasad. Static Task Partitioning for Locked Caches in Multicore Real-Time Systems. *ACM Trans. Embed. Comput. Syst.*, 14(1):4:1–4:30, January 2015.

44 M. R. Soliman and R. Pellizzoni. WCET-Driven dynamic data scratchpad management with compiler-directed prefetching. In *Euromicro Conference on Real-Time Systems (ECRTS)*, volume 76, pages 24:1–24:23, 2017.

45 Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. HDFI: hardware-assisted data-flow isolation. In *Symposium on Security and Privacy (SP)*, pages 1–17. IEEE, 2016.

46 Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. WCET centric data allocation to scratchpad memory. In *Real-Time Systems Symposium (RTSS)*, pages 10–pp. IEEE, 2005.

47 X. Vera, B. Lisper, and J. Xue. Data Cache Locking for Tight Timing Calculations. *ACM Trans. Embed. Comput. Syst.*, 7(1):4:1–4:38, December 2007.

48 Xavier Vera, Björn Lisper, and Jingling Xue. Data Cache Locking for Higher Program Predictability. *SIGMETRICS Perform. Eval. Rev.*, 31(1):272–282, June 2003. doi:10.1145/885651.781062.

49 Jack Whitham and Neil Audsley. Studying the applicability of the scratchpad memory management unit. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 205–214. IEEE, 2010.

50 R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst. (TECS)*, 7(3), 2008.

51 Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *International Symposium on Computer Architecture (ISCA)*, 2014.

52 G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Syst.*, 48(6):681–715, 2012.

A Appendix: May Analysis

In the DM-LRU analysis framework, *may*-analysis is once again performed by considering abstract cache states. Recall that *may*-analysis keeps track of the lower-bound on the age of each addressable memory block. There are a number of differences compared to the analytical tools used for *must* analysis. In *may*-analysis it is necessary to keep track of both $D \in \{0, \dots, A\}$ and $B \in \{0, \dots, A\}$. Here, the meaning of D and B changes. In this case, D represents the maximum lower-bound of any possibly cached DM block. Conversely, B captures the minimum lower-bound on the DM-LRU age of any BE block. It may be the case

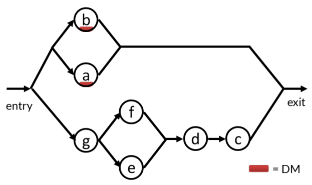


Figure 7 Fragment of process CFG that leads to abstract DM-LRU state $\underline{q} = [\{a, b\}], [\{c\}, \{d\}, \{e, f\}, \{g\}]$.

that $B + D > A$ in order to correctly abstract the age lower-bound resulting from multiple execution paths. It must hold however that $A \leq D + B \leq 2A$. It follows that the abstract domain for *may*-analysis $DMLru^{\sqsupseteq}_{A}$ is defined as:

$$DMLru^{\sqsupseteq}_{A} := \{0, \ldots, A\} \times \{0, \ldots, A\} \times \mathcal{B} \rightarrow \{0, \ldots, A-1, A\}. \tag{11}$$

An abstract state $\underline{q} \in DMLru^{\sqsupseteq}_{A}$ is then represented as two sets of memory blocks, for instance: $\underline{q} = [\{a, b\}], [\{c\}, \{d\}, \{e, f\}, \{g\}] \in DMLru^{\sqsupseteq}_{A}$. In this example, we have $D = 1, B = 4, A = 4$. It follows that the upper-bound on the number of DM memory blocks is 1, and that blocks a and b have at least DM-LRU age 0, and *may* be marked as deterministic blocks. On the other hand, c is a best-effort memory block with DM-LRU age at least 0. It should not come as a surprise that in some states $D + B > A$. Consider the execution depicted in Figure 7 that produces $\underline{q}$. When execution reaches the end of the figure, there could be 0 or 1 DM blocks allocated in cache. Hence the upper-bound on the number of DM blocks has to be $D = 1$. On the other hand, the upper bound on the number of BE blocks is $B = 4$.

The operator $DM^{\sqsupseteq}(\underline{q}, a)$ takes an abstract state $\underline{q}$ and a block a, and returns *true* if a *may* be allocated as a DM block in $\underline{q}$. For ease of notation, we simply use $DM^{\sqsupseteq}(a)$ when the considered abstract state is obvious. We define the operator $DM^{\sqsupseteq}(\underline{q}, a)$ as follows:

$$DM^{\sqsupseteq}(\underline{q}, b^{CL}) := \begin{cases} \mathit{true} & \text{if } CL = DM \wedge \underline{q}(b) < A \\ \mathit{false} & \text{otherwise.} \end{cases} \tag{12}$$

The update abstract transformer $U^{\sqsupseteq}_{D}$ for a DM memory access a can be defined as follows:

$$U^{\sqsupseteq}_{D}(\underline{q}, a) :=$$

$$D' \leftarrow \begin{cases} D + 1 & \text{if } D < A \wedge BE^{\sqsupseteq}(a) \\ D + 1 & \text{if } D < A \wedge \exists x^{DM} \neq a \;:\; \underline{q}(x) = \underline{q}(a) = D - 1 \\ D & \text{otherwise} \end{cases}, \tag{13}$$

$$B' \leftarrow \begin{cases} B - 1 & \text{if } B > 0 \wedge \underline{q}(a) \geq A - B \\ B & \text{if } B = 0 \vee \underline{q}(a) < A - B \end{cases}, \tag{14}$$

$$\lambda b. \begin{cases} 0 & \text{if } b = a & \text{(a)} \\[1em] \underline{q}(b) & \text{if } b \neq a \wedge & \\ & \left\| \begin{array}{l} DM^{\sqsupseteq}(b) \wedge \underline{q}(a) < \underline{q}(b) \\ BE^{\sqsupseteq}(b) \wedge \underline{q}(a) < A - B \end{array} \right. & \text{(b)} \\[1em] \underline{q}(b) + 1 & \text{if } b \neq a \wedge & \\ & \left\| \begin{array}{l} DM^{\sqsupseteq}(b) \wedge \underline{q}(a) \geq \underline{q}(b) \wedge \underline{q}(b) < D' - 1 \\ BE^{\sqsupseteq}(b) \wedge \underline{q}(a) \geq A - B \wedge \underline{q}(b) < A - 1 \end{array} \right. & \text{(c)} \\[1em] A & \text{if } b \neq a \wedge & \\ & \left\| \begin{array}{l} DM^{\sqsupseteq}(b) \wedge \underline{q}(a) \geq \underline{q}(b) \wedge \underline{q}(b) \geq D' - 1 \\ BE^{\sqsupseteq}(b) \wedge \underline{q}(a) \geq A - B \wedge \underline{q}(b) \geq A - 1 \end{array} \right. & \text{(d)} \end{cases} \tag{15}$$

Where D' (B', resp.) is the new value of D (B, resp.) after the update. Similarly, the update abstract transformer $U_{\underline{B}}^{\sqsupseteq}$ for a best-effort memory access a can be defined as follows:

$$U_{\underline{B}}^{\sqsupseteq}(\underline{q}, a) := \tag{16}$$

$$\lambda b. \begin{cases} A - B & \text{if } b = a & \text{(a)} \\ \underline{q}(b) & \text{if } b \neq a \wedge \left\| \begin{array}{l} DM^{\sqsupseteq}(b) \\ BE^{\sqsupseteq}(b) \wedge \underline{q}(a) < \underline{q}(b) \end{array} \right. & \text{(b)} \\[1em] \underline{q}(b) + 1 & \text{if } b \neq a \wedge BE^{\sqsupseteq}(b) \wedge \underline{q}(a) \geq \underline{q}(b) \wedge \underline{q}(b) < A - 1 & \text{(c)} \\ A & \text{if } b \neq a \wedge BE^{\sqsupseteq}(b) \wedge \underline{q}(a) \geq \underline{q}(b) \wedge \underline{q}(b) \geq A - 1 & \text{(d)} \end{cases} \tag{17}$$

To clarify how the $U^{\sqsupseteq}$ operation transforms a given state , consider the abstract state $\underline{q} = [\{a, b\}], [\{c\}, \{d\}, \{e, f\}, \{g\}]$, where $D = 1, B = 4$. Assume that DM block h is accessed, whose DM-LRU age is currently A or higher. First, the value of D' (B', resp.) is computed as $D' = D + 1$ ($B' = B - 1$, resp.); next, $\{a, b\}$ both satisfy the fourth condition in $U_{\underline{D}}^{\sqsupseteq}$ – Equation 15, first case of (c); block c, d, e and f satisfy the fifth condition; g the seventh. The resulting updated abstract state is: $\underline{q}' = [\{h\}, \{a, b\}], [\{\}, \{c\}, \{d\}, \{e, f\}]$. Note that in the resulting state $B = 3$, hence the least lower-bound on any BE block is $A - B = 1$.

***May*-analysis Join.** The join abstract transformer for DM-LRU *may*-analysis is symmetric to the join abstract transformer used for DM-LRU *must*-analysis. The joined state will contain all the blocks in the union of the joining states, each with the minimum age in any of the two states. Furthermore, D is taken as the maximum between the value of D in the joining states. Similarly, B is taken as the maximum between the value of B in the joining states. As such, after a join, it always holds that $D + B \leq 2A$. Equation 18 formalizes the $J^{\sqsupseteq}(\bar{q}, \bar{p})$ abstract transformer:

$$J^{\sqsupseteq}(\bar{q}, \bar{p}) := \; D \leftarrow \max\{D_{\bar{q}}, D_{\bar{p}}\}, B \leftarrow \max\{B_{\bar{q}}, B_{\bar{p}}\}, \lambda b. \min\{\bar{q}(b), \bar{p}(b)\}. \tag{18}$$

To clarify the join operation, consider the state $\underline{q} = [\{a, b\}], [\{c\}, \{d\}, \{e, f\}, \{g\}]$ obtained in Figure 7, and the state $\underline{q}' = [\{h\}, \{a, b\}], [\{\}, \{c\}, \{d\}, \{e, f\}]$ obtained as $\underline{q}' = U^{\sqsupseteq}(\underline{q}, h^{DM})$ (i.e. by accessing the DM block h). If we were to join $\underline{q}$ with $\underline{q}'$, the resulting state would be $\underline{q}'' = [\{a, b, h\}, \{\}], [\{c\}, \{d\}, \{e, f\}, \{g\}]$.

***May*-analysis Classification.** It is possible to classify a memory access using a classification function that will either return M for cache miss, or $\top$ in case access to a memory block cannot be guaranteed to be a miss given the current abstract state. The classification function of the *may* analysis is defined as:

$$C^{\sqsupseteq}(\underline{q}, a^{CL}) := \begin{cases} M & \text{if } \underline{q}(a) = A \\ \top & \text{otherwise.} \end{cases} \tag{19}$$

Modeling Cache Coherence to Expose Interference

Nathanaël Sensfelder
ONERA, Toulouse, France

Julien Brunel
ONERA, Toulouse, France

Claire Pagetti
ONERA, Toulouse, France

Abstract

To facilitate programming, most multi-core processors feature automated mechanisms maintaining coherence between each core's cache. These mechanisms introduce interference, that is, delays caused by concurrent access to a shared resource. This type of interference is hard to predict, leading to the mechanisms being shunned by real-time system designers, at the cost of potential benefits in both running time and system complexity.

We believe that formal methods can provide the means to ensure that the effects of this interference are properly exposed and mitigated. Consequently, this paper proposes a nascent framework relying on timed automata to model and analyze the interference caused by cache coherence.

2012 ACM Subject Classification Computer systems organization → Multicore architectures; Computer systems organization → Real-time systems

Keywords and phrases Real-time systems, multi-core processor, cache coherence, formal methods

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.18

Supplement Material ECRTS 2019 Artifact Evaluation approved artifact available at
`https://dx.doi.org/10.4230/DARTS.5.1.7`
`https://github.com/nsensfel/phylog-cache-coherence/`

Acknowledgements We would like to thank Mamoun Filali-Amine (IRIT-CNRS) for his helpful insights on how to validate our model and related works suggestions.

1 Introduction

The next generation of aircrafts will embed multi-core processors. Indeed, it will be more and more difficult to find mono-core processors on the market and, when correctly programmed, multi-core processors offer huge opportunities to reduce the amount of equipment required to host multiple applications compared to *federated* or single-core IMA (*Integrated Modular Avionics*) architectures. However, multi-core processors come with several drawbacks, among which is the lack of predictability [26, 27], one of the key elements of certification expectations. This lack of predictability is caused by *interference*, a delay inherent to the concurrent access to a shared resource.

Cache Coherence. In most multi-core processors, each core has its own cache memory, of which it is virtually the sole accessor. A *cache coherence* protocol ensures that:

- At any given time, a memory location can either be accessed by a single cache controller, in which case both writing and reading are allowed, or by any number of cache controllers, in which case only reading is allowed.
- Any copy of a memory location held in a cache has the most up-to-date value.

Maintaining this cache coherence requires exchanges of information between cache memories. These exchanges can be the source of a large amount of additional traffic, a potential hindrance that we qualify of *implicit* interference, because of how difficult to predict they are. Additionally, it can result in the removal of elements from the cache, which may lead to time consuming communications with the system's memory (*cache misses*).

While multi-core processors feature hardware to efficiently and automatically handle cache coherence, the black-box nature of commercial processors leads to a lack of control, visibility, and predictability of the cache coherence protocol and, by extension, of the delays it may create.

Current Research Practices. Several approaches have been developed in the literature to deal with the interference found in multi-core processors. The main solutions to ensure predictability are 1) preventing any kind of uncontrolled interference (e.g. run-time services [15, 28]); 2) enforcing a unique access to any shared resource at any time, so as to be equivalent to a single core situation (e.g. execution models [18, 5, 12]). Because its interference is difficult to predict, most of the considered hardware do not have or use automatic cache coherency. Instead, the burden of cache management is placed on the developers, forcing an application-specific solution (e.g. *scratchpad memory* [25, 22]). Such solutions prevent the gains in performance that would otherwise be provided by automatic hardware cache management mechanisms.

Contributions. We believe that the implicit interference generated by the cache coherence can be exposed and taken into account to achieve predictable programming of a multi-core processor. In this work, we focus on exposing these unexpected delays, the analysis of a formal model of the processor.

We start this paper by going into more details on how cache coherence can be achieved (Section 2), the type of system we are interested in (Section 3), and the categories of interference it can host (Section 4). We then present the tools that we use to model and analyze it (Section 5). Afterwards, we explain our choices in how we modeled the cache coherence in a multi-core processor (Section 6). Finally, we showcase some of the results that can be extracted from our model (Section 7), before listing some related works (Section 8) and concluding (Section 9).

2 Cache Coherence Protocols

We start by introducing archetypal systems on which coherence protocols run. We then present how those protocols behave.

A number of components (see Figure 1) are involved in the coherence.

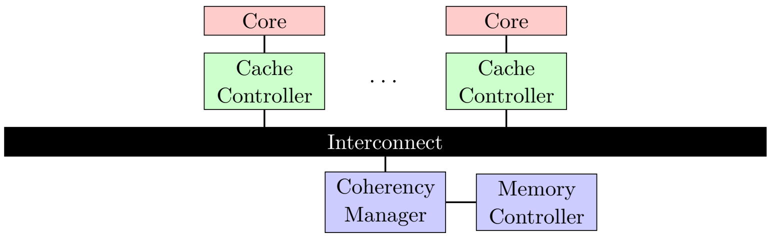


Figure 1 Components involved in cache coherence.

Memory Element: The main memory is composed of chunks (or memory elements) which have a fixed size and contain multiple addressable elements. Reading/writing from/to an address in the main memory actually corresponds to reading or writing a whole memory element. The distinction between an addressable space and a memory element is not relevant to cache coherence, and thus, for simplification purposes, this paper considers that each memory element has a single address.

Core: The component actually using and modifying memory element values. Instead of accessing the original memory elements through the interconnect, each core is linked to its own private cache. The content of this cache is managed by an associated cache controller. The core can ask its cache controller for the value held by the memory element at a given address through a `load` request. It can also send a `store` request to modify this value. Additionally, the core can issue an `evict` request, which tells its cache controller to invalidate a memory element copy. While it is rare for cores to be the initiators of `evict` requests, it remains a possibility (e.g. for micro-optimization). Cores can be made to `stall` by their cache controller, delaying the emission of a request until the cache controller is ready to accept it.

Cache Controller: Component that handles requests from its core, potentially initiating a transaction by making a query on the interconnect. Such queries take the form of a `GetS` when asking for a read-only copy of a memory element and that of a `GetM` when asking for a read-and-write copy. Queries that indicate a new value for the memory element are done through `PutM` messages. Depending on the protocol, variants of these messages may be used. Cache controllers are also able to reply to the query of another cache controller with a data reply (`data`). Additionally, cache controllers may initiate `evict` requests on themselves to make space for new memory element copies. These self-requests are controlled by a *cache replacement policy*, which is most commonly a speed-over-accuracy variation on the *Least Recently Used* policy.

Coherency Manager: Component that stores information on the state of the cache controllers, to help maintain the cache coherence. Using this stored information, it can tell if a query should be answered by the memory controller or not. This component is very much dependent on which protocol is being implemented, and can range from being a simple link between the cache controllers to actually being multiple separate components (e.g. all directory nodes of a directory-based cache coherence protocol). It is usually found inside the interconnect.

Memory Controller: Component that handles the modification or copy of the original memory elements.

Interconnect: Component that regulates and handles the propagation of messages between cache controllers, memory controllers, and the coherency manager.

▶ **Definition 1** (Request, Message, Query, and Data Reply). *To keep things separate, we use the term request when talking about communications between a core and its cache controller, and the term message when talking about communications that use the interconnect. As such, queries (e.g. `GetM`, `GetS`, `PutM`) are messages, and so are data replies (e.g. `data`). Thus, messages = queries ∪ data replies.*

▶ **Definition 2** (Transaction). *A transaction is composed of a query and of all the data messages the completion of that query requires.*

Each message transiting through the interconnect, and each cache controller query, is about a specific memory element. Upon receiving either one of those, cache controllers look up the state they associate with their copy of the memory element for this address, and act according to the cache coherence protocol.

2.1 Protocols

Most cache coherence protocols are based on the MSI protocol, named after the states given to copies of the memory elements by the cache controllers. M stands for Modified, the state a cache controller gives its copy of the memory element to indicate that it has both read-and-write access to the original. S stands for Shared, and is the equivalent for read-only access. I stands for Invalid, when a cache controller does not currently have a copy.

MSI-based protocols are all categorized as *Write-Back*, because caches may contain a more up-to-date value of the memory element than the RAM.

The aforementioned protocols are referred to according to their states and general idea, however, the definition of their behavior depends on the system they are implemented on.

There are two main families of cache coherence implementation: snooping-based and directory-based. When using a snooping-based protocol, cache controller queries are broadcasted to all cache controllers and to the coherency manager. The protocol also ensures that only one of the components answers the query. This answer is not broadcasted, but is instead only meant for the query's originator. For such protocols to properly function, all the components have to receive the queries in the same order. In the sequel, we only take into consideration snooping-based protocols.

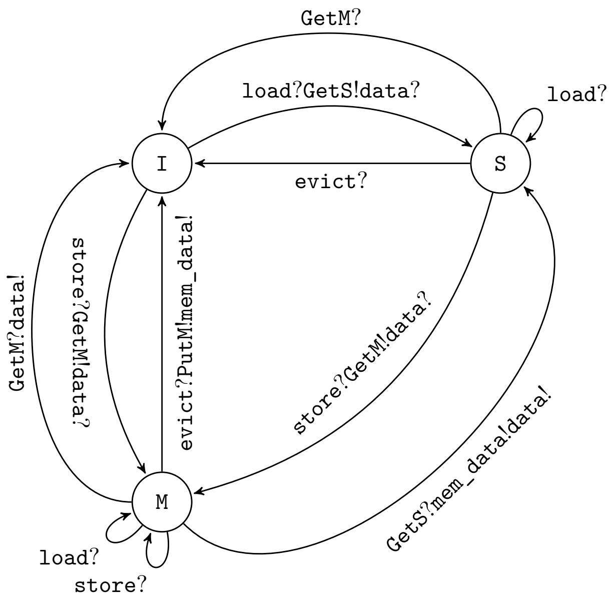


Figure 2 Generic MSI Cache Controller.

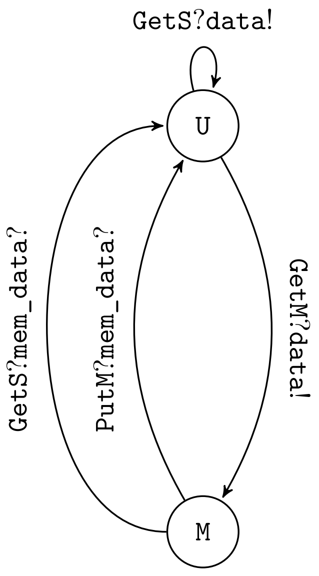


Figure 3 Generic MSI Coherency Manager.

Automata describing a generic snooping-based MSI protocol can be seen in Figure 2 and Figure 3. Figure 2 shows how the state given to a memory element's copy evolves when receiving a request (`store`?, `load`?, or `evict`?), or a query (`GetM`? or `GetS`?). Data exchanges between cache controllers are also represented (`data`! and `data`?). Cache controllers do not differentiate between data sent from another cache controller and data sent from the memory controller (both use `data`?). Sending data *to* the memory controller, however, is marked as `mem_data`!. Figure 3 represents the coherency manager, which keeps track of whether the memory has the most up-to-date value for a memory element (state `U`) or not (state `M`).

This particular protocol considers that cache controllers delay incoming requests until they are able to use the interconnect, and that transactions cannot take place simultaneously.

These automata actually describe a generic snooping-based MSI protocol. They feature *macro-transitions* (a succession of atomic transitions). The next section presents a more detailed protocol.

3 MSI Snooping-Based Protocol

3.1 A Few Caveats

These are the hypotheses made on the targeted hardware. Placing such hypotheses (or lack thereof) is necessary to properly define the targeted cache coherence protocol.

▶ **Hypothesis 1** (Non-Atomic Requests). *Cores are able to issue `load`, `store`, and `evict` requests to their cache controller regardless of whether the cache controller is currently able to initiate a transaction on the interconnect. In this paper, we consider that this is implemented through the use of a FIFO queue between each cache controller and the interconnect.*

▶ **Hypothesis 2** (Unique Interconnect). *The interconnect is unique. As a result, all cache controllers are able to see all transactions, and those transactions are all seen in the same order. Examples of excluded hardware include many-core processors, which feature a Network-On-Chip.*

▶ **Hypothesis 3** (Split-Transaction Interconnect). *The interconnect supports simultaneous transfer of data and queries and allows multiple transactions to take place simultaneously.*

3.2 From Abstract to Concrete Behaviour

In Subsection 2.1, we have seen automata using macro-transitions to describe a generic snooping-based MSI protocol. Let us now look in details at what is composing the transition from I to S with the `load?GetS!data?` label. Let us consider two cache controllers, CC_0 and CC_1, each of which is driven by its own core (CU_0 and CU_1, respectively) and a memory element. Let us assume that, while CC_1 already has read-only access to that memory element, CC_0 does not, and that core CU_0 issues a `load` request to acquire it.

In the sequence diagram of Figure 4, we see the behavior of all components involved. Once the core issues the `load` request, the cache controller generates a `GetS` query to the interconnect. The latter broadcasts the `GetS` to all cache controllers, including the query's originator, and the coherency manager. As the owner of the data is the memory, the coherency manager transmits that query to the memory controller, which, in turn, sends the data to the core CU_0.

In order to expose the interference, we need to model the atomic transitions and intermediate states, such as the ones shown in the figure.

3.3 Detailed Snooping-Based MSI Protocol

Instead of representing the full automaton as a graph, we use a matrix representation (see Figure 6). The first column details every possible states. As in [20], the naming of each state is determined by the following reasoning: Invalid (`I`), Shared (`S`), and Modified (`M`) are the three *stable* states of the `MSI` protocol. The other states are *transient*. Reception of a request that requires use of the interconnect will usually lead to a $\texttt{XY}^{\texttt{BD}}$ transient state, which means that the cache controller is handling a transition between the stable states `X` and `Y`,

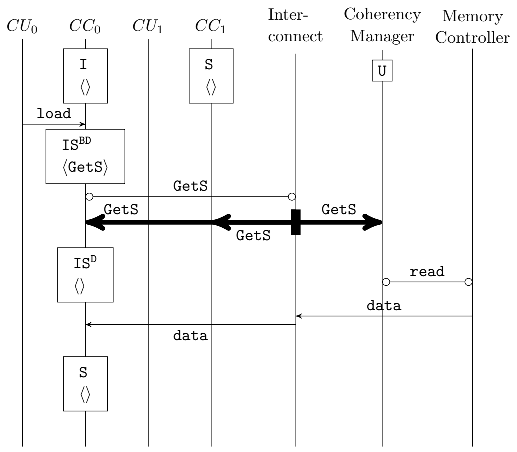


Figure 4 `load` request.

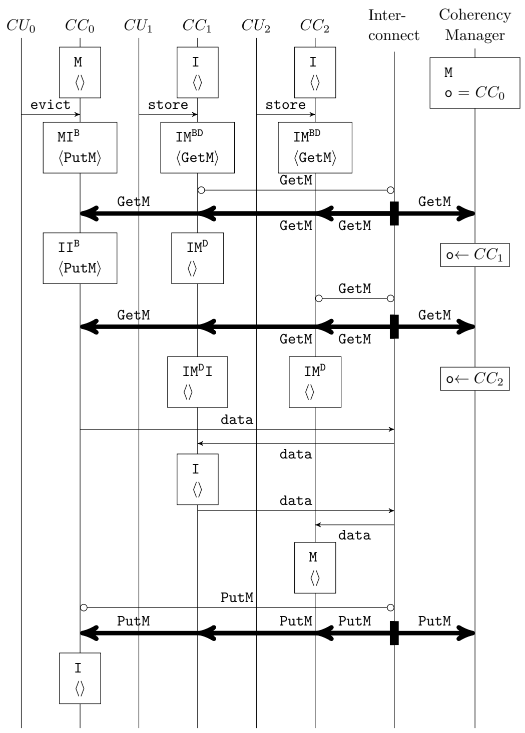


Figure 5 Double `store`.

with (^{B}) indicating that this transition requires the acquisition of the interconnect and (^{D}) the reception of a related data reply (whether it comes from an other cache controller or the memory). This can be followed by XY^D if the cache controller sees its own query before receiving a reply, or XY^B if a reply is received before the query is processed. This happens when, despite processing all queries in the same order, not all cache controllers take the same time to do so. Another possibility is for an external query to be received when in the XY^D state. Indeed, at that point, the system pretty much considers that the cache controller is in the `Y` state and thus has the responsibilities that the `Y` would require. This makes it possible for a cache controller to see a query it needs to act upon before being actually ready to do so (e.g. observing a `GetM` query while waiting for `data`). These states have a XY^DA form (which means that when all is handled, the cache controller ends up in the `A` state), or XY^DAB (which ends up leading to the `B` state). As it may be that the required action is to reply to said query, it is sometimes necessary to remember the originator of the query. This is marked as `r`←`s`.

When the core makes a request (`load`, `store` or `evict`), the second macro-column indicates how the cache controller behaves. The `a`/`b` notation denotes the emission of an `a` message on the interconnect, followed by a transition of the memory element copy's state to `b`. If you look at the `load` from the `I` state, the cell indicates that the `GetS` request will be generated and the reached state is IS^{BD}. We recognize the beginning of the sequence diagram described in Figure 4. Grayed out cells indicate situations that cannot occur in the protocol, due to our hypotheses.

The third macro-column (named *Interconnect access*) indicates what happens when the previously queued query is broadcasted on the interconnect. When in the IS^{BD} state, we know that, at some point, our previously queued `GetS` query is going to be broadcasted

State	Core Request			Interconnect Access	Data Reply	Received Queries		
	load	store	evict			GetS	GetM	PutM
I	GetS/IS^{BD}	GetM/IM^{BD}	▨	▨	▨	-	-	-
IS^{BD}	stall	stall	stall	-/IS^{D}	-/IS^{B}	-	-	-
IS^{B}	stall	stall	stall	-/S	▨	-	-	▨
IS^{D}	stall	stall	stall	▨	-/S	-	-/IS^{D}I	▨
IS^{D}I	stall	stall	stall	▨	-/I	-	-	▨
IM^{BD}	stall	stall	stall	-/IM^{D}	-/IM^{B}	-	-	-
IM^{B}	stall	stall	stall	-/M	▨	-	-	-
IM^{D}	stall	stall	stall	▨	-/M	r←s -/IM^{D}S	r←s -/IM^{D}I	▨
IM^{D}I	stall	stall	stall	▨	r!data -/I	-	-	▨
IM^{D}S	stall	stall	stall	▨	r!data m!data -/I	-	-/IM^{D}SI	▨
IM^{D}SI	stall	stall	stall	▨	r!data m!data -/I	-	-	▨
S	hit	GetM/SM^{BD}	-/I	▨	▨	-	-/I	▨
SM^{BD}	hit	stall	stall	-/SM^{D}	-/SM^{B}	-	-/IM^{BD}	▨
SM^{B}	hit	stall	stall	-/M	▨	-	-/IM^{B}	▨
SM^{D}	hit	stall	stall	▨	-/M	r←s -/SM^{D}S	r←s -/SM^{D}I	▨
SM^{D}I	hit	stall	stall	▨	r!data -/I	-	-	▨
SM^{D}S	hit	stall	stall	▨	r!data m!data -/S	-	-/SM^{D}SI	▨
SM^{D}SI	hit	stall	stall	▨	r!data m!data -/I	-	-	▨
M	hit	hit	PutM/MI^{B}	▨	▨	m!data s!data -/S	s!data -/I	▨
MI^{B}	hit	hit	stall	m!data -/I	▨	m!data s!data -/II^{B}	s!data -/II^{B}	▨
II^{B}	stall	stall	stall	-/I	▨	-	-	-
	Handling Requests					Handling Queries		

Figure 6 Cache Controller Memory Element State Changes (adapted from [20]).

on the interconnect. This will result in reaching the IS^{D} state. As a side note, if the core makes a second `load` request on the same memory element while the copy is IS^{BD}, that new request is stalled.

The fourth macro-column describes the behavior upon reception of a data reply.

The fifth macro-column (named *Received Queries*) defines the behavior of the cache controller when snooping a transiting query that is not its own (which would otherwise pertain to the third macro-column). For instance, from state `S`, when snooping a `GetS`, the cache controller does not do anything, as can be seen with core CU_1 in the sequence diagram of Figure 4.

Replying with a message `d`, meant for `t` ($\mathtt{t} = \mathtt{m}$ when sending to the memory controller and the coherency manager, $\mathtt{t} = \mathtt{s}$ when sending to the cache controller that initiated the transaction, and $\mathtt{t} = \mathtt{r}$ when sending to the initiator of an earlier query) is written as `t!d`.

▶ **Example 3.** *Let us have a look at a more complex behavior: when 2 cores attempt modification of the same memory element. This is illustrated in the sequence diagram of Figure 5. CC_0 starts with read-and-write access to the memory element (its copy being in the `M` state), neither CC_1 nor CC_2 have a copy (state `I`), and the coherency manager knows that its value is out of date (state `M`).*

The sequence starts when CU_0 issues an `evict` request and both CU_1 and CU_2 issue a `store` request. CC_0 receives the `evict` request, queues a `PutM` query and now considers the memory element to be MI^B (that is, "was Modified, will be Invalid once access to the interconnect is granted"). On the other hand, the other two caches receive their `store` requests, queue a `GetM` query, and now consider the memory element to be IM^{BD} ("was Invalid, will be Modified after access to the interconnect and reception of a `data` reply").

All the cache controllers want to access the interconnect. The internal behavior of the interconnect will drive this choice. Most of the time, the interconnect is based on Fair-RR (Round Robin) [11]. In this scenario, the interconnect first broadcasts the `GetM` query from CC_1's queue, which is now empty.

CC_1, seeing its own query, confirms that it has accessed the interconnect, and switches to the IM^D state to await a `data` reply. The coherency manager ignores the query. Seeing CC_1's `GetM` query passing through the interconnect, CC_0 has to reply with a `data` message (this corresponds to `s!data` in the protocol definition), containing its value for the memory element, and to transition to the II^B state.

CC_2's `GetM` query is broadcasted. As it is about to receive the data with read-and-write access, CC_1 is the component that should reply to CC_2's query. Not having the data yet, CC_1 is currently unable to do so. Instead, it transitions to the IM^DI, remembering that it should send the data to CC_2 as soon as possible.

Finally receiving the data, CC_1 applies completes CU_1's request, sends the updated data to CC_2 and transitions to the `I` state (as CC_2 wants read-and-write access).

CC_2 receives the data and completes CU_2's request.

CC_0's `PutM` is broadcasted, but has been superseded by a previous `GetM` and thus causes no reaction in the other cache controllers or the coherency manager. CC_0 transitions to `I`, completing its core's request.

3.4 Coherency Manager

State	Received Queries				Data Reply
	GetS	GetM	PutM (Owner)	PutM (Other)	data
U	s!data	s!data o←s -/M	▨	-	▨
U^D	stall	stall	stall	-	-/U
U^B	o← ∅ -/U	-	o← ∅ -/U	-	▨
M	o← ∅ -/U^D	o←s	o← ∅ -/U^D	-	-/U^B

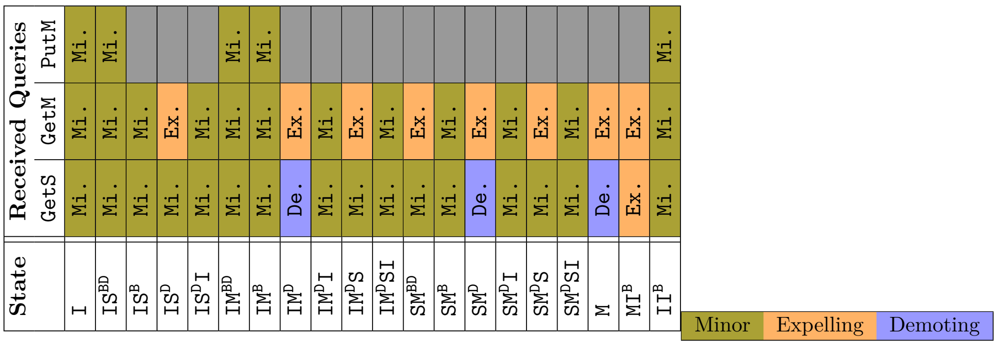
Figure 7 Coherency Manager Memory Element State Changes (adapted from [20]).

Figure 7 shows how the coherency manager keeps track of whether the RAM has the most up-to-date value for a memory element (state `U`) or if a cache controller does (state `M`). This is used to determine if the RAM should be the one to reply when either a `GetS` or a `GetM` query passes through the interconnect. The `U` state indicates that the RAM currently has the most up-to-date value. The $\texttt{U}^\texttt{D}$ state indicates that the RAM should be the one to respond to queries, but it still hasn't received the latest value. Unlike the cache controller, it will not switch to a dedicated state but instead force queries from the interconnect to stall until the problematic query can be fulfilled. $\texttt{U}^\texttt{B}$ indicates that the RAM has received the latest value, but has not yet seen the query that led this data to be sent.

The exact cache controller currently in charge of the memory element is kept track of. Change of ownership are marked as $\texttt{o}{\leftarrow}\texttt{s}$ (the query originator becomes the new owner) and $\texttt{o}{\leftarrow}\,\emptyset$ (there is no longer an owner, meaning that the RAM is currently responsible for it).

▶ **Example 4.** *Back to the sequence diagram of Figure 5 and to Example 3, let us observe the behavior of the coherency manager. The coherency manager reacts to each* `GetM` *query, updating its internal state to reflect the change of ownership. Thus, the coherency manager starts by considering that CC_0 is the only one to have a valid (i.e. up-to-date) value of the memory element, then, upon seeing the first* `GetM`*, considers CC_1 to be responsible for it ($\texttt{o}{\leftarrow}\texttt{s}$ in the table). As a result, at the end of the execution, the coherency manager knows that the* `PutM` *query is originating from a cache controller that is not currently responsible for that memory element and can thus safely ignore it.*

4 Interference



State	Received Queries: GetS	GetM	PutM
$\texttt{I}$	Mi.	Mi.	Mi.
$\texttt{IS}^\texttt{BD}$	Mi.	Mi.	Mi.
$\texttt{IS}^\texttt{B}$	Mi.	Mi.	
$\texttt{IS}^\texttt{D}$	Mi.	Ex.	
$\texttt{IS}^\texttt{D}\texttt{I}$	Mi.	Mi.	
$\texttt{IM}^\texttt{BD}$	Mi.	Mi.	Mi.
$\texttt{IM}^\texttt{B}$	Mi.	Mi.	Mi.
$\texttt{IM}^\texttt{D}$	De.	Ex.	
$\texttt{IM}^\texttt{D}\texttt{I}$	Mi.	Mi.	
$\texttt{IM}^\texttt{D}\texttt{S}$	Mi.	Ex.	
$\texttt{IM}^\texttt{D}\texttt{SI}$	Mi.	Mi.	
$\texttt{SM}^\texttt{BD}$	Mi.	Ex.	
$\texttt{SM}^\texttt{B}$	Mi.	Mi.	
$\texttt{SM}^\texttt{D}$	De.	Ex.	
$\texttt{SM}^\texttt{D}\texttt{I}$	Mi.	Mi.	
$\texttt{SM}^\texttt{D}\texttt{S}$	Mi.	Ex.	
$\texttt{SM}^\texttt{D}\texttt{SI}$	Mi.	Mi.	
$\texttt{M}$	De.	Ex.	
$\texttt{MI}^\texttt{B}$	Ex.	Ex.	
$\texttt{II}^\texttt{B}$	Mi.	Mi.	Mi.

■ **Figure 8** Occurrences of Interference.

Let us now categorize how a cache controller may be negatively affected by the actions of another. Figure 8 summarizes the occurrences of each interference category. In the Figures 9, 11, and 10, the dark gray area indicates when the cache controller is unavailable due to having to handle the incoming query (deciding how to act and, potentially, updating its internal state), and the light gray area shows when its core's next request for that memory element may be negatively impacted by the change of state.

▶ **Definition 5** (Minor Interference). *Cache controllers have actions to perform upon receiving any type of request. Because of this, every time a cache controller has to deal with an incoming query, there is a very small amount of time during which it cannot be used by its core. We call this unavailability period minor interference. And, while the effect of each minor interference is so small as to be considered negligible, their accumulation most definitely is not. Indeed, minor interferences are one of the main motivations behind the use of a directory-based coherency protocol (in which minor interferences are only experienced by cache controllers likely to have a use for that query) over a snooping-based one (in which* all *cache controllers are affected by every query).*

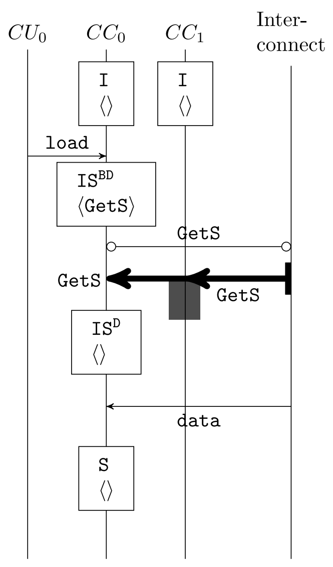


Figure 9 Minor.

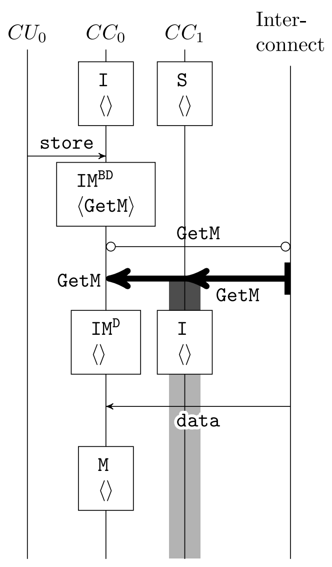


Figure 10 Expelling.

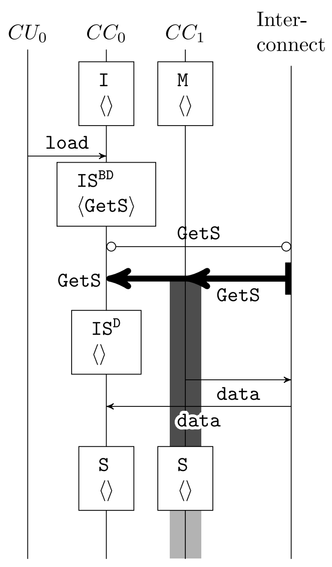


Figure 11 Demoting.

Figure 9 shows an example of minor interference: the CC_1 cache controller has to process the `GetS` broadcast, despite that message not requiring any reply or internal state update from CC_1.

▶ **Definition 6** (Expelling Interference). *To maintain the principles of cache coherency, it may be required for a cache controller to dispense of its copy of a memory element, relinquishing its access rights. This is caused by another cache controller demanding read-and-write access to that memory element (a `GetM` query). We have, however, marked the reception of a `GetS` query for an element in the `MI`^{B} as being an expelling interference in Figure 8. It could be argued that reaching the `MI`^{B} indicates that the cache controller is already in the process of evicting its copy of the memory element. But, as the `MI`^{B} state allows immediate (i.e. `hit`) access for both writing and reading that memory element, we still consider this event to have a negative impact.*

Figure 10 shows an example of expelling interference: the CC_1 cache controller, receiving a demand for read-and-write access, is forced to relinquish its read-only copy.

▶ **Definition 7** (Demoting Interference). *Another type of interference is the demoting interference, in which a cache controller has to abandon its writing access rights to a memory element, while retaining its reading access.*

Figure 11 shows an example of demoting interference: the CC_1 cache controller, receiving a demand for read access on that memory element, has to update the value from the main memory and go from read-and-write access to read-only access.

5 Formal Modeling of Real-Time Systems with Timed Automata

To expose the interference presented in the previous section, we chose to use formal methods. More precisely, we are relying on timed automata [1] to model and analyze our system.

A *timed automaton* is an extended automaton with variables and clocks. During the system's execution, the state of timed automaton is defined as a location, the value of its integer variables and of its clocks. The evolution of these integer variables is controlled by the automaton's transitions, whereas all of the system's clocks progress at the same rate, following the passing of time. To indicate that a location should be left immediately, UPPAAL [4] offers the following location modifiers:

Urgent: The location must be left before any time passes.

Committed: The location must be left before any time passes, and the next transition must originate from a **committed** location.

Invariant ϕ: The location is defined only if a linear constraint ϕ holds true. ϕ may reason over the automaton's integer variables, clocks, or both.

The automata transitions are composed of the following:

Guard: Prerequisite (linear constraint) for this transition to be able to *fire*. The condition uses the automaton's integer variables, clocks, or both.

Synchronization: Allows to have more than one automaton transitioning during a step, by synchronizing multiple transitions over a channel. The channel can be used in either receiver (with a `?` suffix) or sender (with a `!` suffix) mode. On a channel that was declared without modifier, the transition requires exactly two automata to synchronize during this step: the sender, and the receiver. It is also possible for a channel to have been declared as a **broadcast** channel, in which case the sender synchronizes with all available receivers. Furthermore, the channel may have been declared as **urgent**, which prevents waiting in a location if the synchronization can occur. Finally, priorities between channels may be put in place.

Update: Sequence of instructions to alter the automaton's integer variables, or reset its clocks.

Select: The transition selects the given integer variables' next value from a specified range.

Example. This subsection presents an example of UPPAAL model: a processor attempts to read a variable, which may be either in RAM or in its cache. The automaton in Figure 12 corresponds to the core, the one in Figure 13 to the RAM controller, and the remaining one (Figure 14) is used to mark a transition as urgent by having an automaton always ready to synchronize on a dedicated urgent channel (`FORCE_URGENT`). In this model, the `FORCE_URGENT` and `READ_LINE` channels are both declared as urgent.

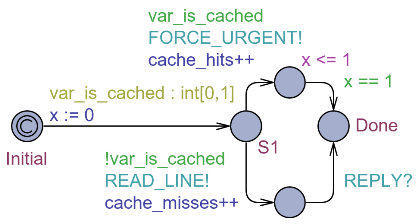


Figure 12 Core and cache.

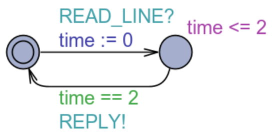


Figure 13 RAM.

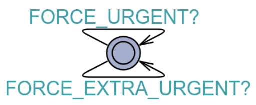


Figure 14 Urgence.

The Core Automaton (Figure 12). Its initial location is marked as committed, meaning that it is left immediately. The exiting transition sets the `x` clock to 0, and the *var_is_cached* variable to a value in the $[0, 1]$ range. The `x` clock will be used to know how long it took for the processor to get its variable. Two transitions are fireable from the *S1* location, depending on whether the targeted variable is cached or not. If it is indeed cached, the transition labeled `FORCE_URGENT` is the only one fireable and it synchronizes with the automaton of Figure 14, forcing it to be taken as soon as possible. Additionally, the transition increases an integer variable that counts the number of times a variable was found in the cache. Taking said transition leads to a location in which the only exiting transition requires the `x` clock to equal 1 unit of time before arriving in the *Done* location.

If the variable was not in the cache, the other transition from *S1* is active and leads to a synchronization on the **`READ_LINE`** which is also to be taken as soon as possible. This time, however, it is possible for that synchronization to not be immediately available, as the RAM controller automaton may be handling another query and thus not be ready to synchronize as it would not be in its initial location. This also justifies not marking the location as urgent or committed: the automaton may have to wait an unknown amount of time. Once the synchronization does happen, an integer variable counting the number of times the variable was not found in the cache is incremented, then the automaton waits for the RAM automaton to synchronize on the **`REPLY`** channel before considering it has acquired the variable.

The RAM Controller Automaton (Figure 13). Its initial location awaits synchronization on the **`READ_LINE`** channel. Since **`READ_LINE`** is urgent, the transition happens as soon as possible. It resets the automaton's `time` clock back to 0. The synchronization leads to a location which has to be left strictly before more than 2 units of time pass, as defined by the invariant. To ensure that the automaton stays in this location for exactly 2 units of time, the only exiting transition has a guard stating just that. This transition also requires a synchronization on the **`REPLY`** channel before allowing a return to the automaton's initial location.

6 Model of the Cache Coherence

This sections describes the general ideas behind how we modeled the cache coherence in UPPAAL. We have released the model under an LGPL v3 license at `https://www.onera.fr/sites/default/files/598/ecrts19.zip`.

6.1 Modeling Strategy

The model contains one automaton per component present in Figure 1, an automaton in charge of synchronizing on the **`FORCE_URGENT`** channel (in an identical manner to the one in Figure 14), as well as message queues for both queries and data (Sub-section 6.6). Each core runs exactly one program. To change the number of cores, one simply has to add or remove cores (and associated cache controllers) and to change the value of a dedicated system-wide constant. Moreover, each component has a unique identifier, which is used both to target a specific automaton on some synchronization, and to indicate the emitter of requests and queries.

The states and transitions seen on the automata do not visibly reflect any program or protocol. This means that the stable states (`M`, `S`, `I`) and the transient states ($\mathtt{IS}^{\mathtt{BD}}$, $\mathtt{IS}^{\mathtt{D}}$, ...) will not appear explicitly. Instead, the automata's designs are focused on their synchronizations, with the logic (and state) of the protocols being held in their variables instead. As such, the same automaton can easily be used for any program or protocol (provided the hypotheses from Sub-Section 3.1 remain), only requiring small changes in the definition of the functions found in its transitions.

Priorities on synchronizations are used to reduce the number of redundant system states. For example, any transition that exits a waiting location (i.e. location in which nothing happens until a clock has reached a certain count) has a higher priority than any other type of transition.

6.2 Core

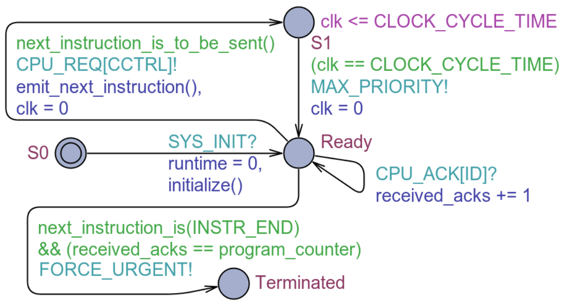


Figure 15 Model of the Core.

```
program_line_t program_0 [7] =
   {
      {INSTR_LOAD ,    1},
      {INSTR_LOAD ,    2},
      {INSTR_STORE ,   3},
      {INSTR_LOAD ,    3},
      {INSTR_STORE ,   1},
      {INSTR_EVICT ,   1},
      {INSTR_END ,     0}
   };
```

Figure 16 Model of a Program.

Programs are modeled using arrays of address-targeting instructions, not so dissimilar to their binary executable. These arrays only contain instructions related to memory accesses (`INSTR_LOAD`, `INSTR_STORE`, `INSTR_EVICT`), and one (`INSTR_END`) to indicate that the execution of the program is completed. An example can be seen in Figure 16.

The automaton corresponding to the core is shown in Figure 15. Progress of the program's execution is tracked by the `program_counter`, which is incremented each time an instruction has been started. Another integer variable, `received_acks`, counts how many times the cache controller has confirmed that a request has been fulfilled. The sending of each instruction to the cache controller is separated by at least the time of a clock cycle.

To ensure that synchronization occurs with the right automaton, the request uses the cache controller's identifier to select a sub-channel of `CPU_REQ`. Conversely, acknowledgments are received on the sub-channel of `CPU_ACK` corresponding to the core's identifier. Upon reaching the `INSTR_END` instruction, the automaton has to wait until all of its outstanding requests have been fulfilled before being able to reach the `TERMINATED` state.

6.3 Coherency Manager and Memory Controller

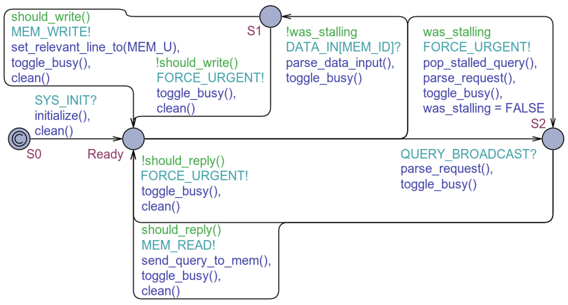


Figure 17 Model of the Coherency Manager.

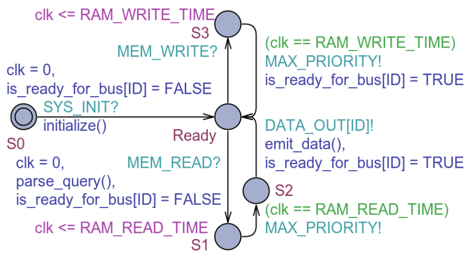


Figure 18 Model of the Memory Controller.

The timed automaton modeling the coherency manager can be seen on Figure 17. The coherency manager has to know for which memory elements the RAM copy is to be considered as superseded by a cache controller. For this purpose, it maintains an array associating a state to each memory element address. The size of this array must be able to accommodate all cache controllers having their caches full of superseding copies of memory elements. In effect, $|mem\_array| = |cache\_array| \times |caches|$.

After initializing its array with default values, the timed automaton waits for either a cache controller query or a `data` message.Receiving any of these leads to an update of the internal state associated with the related memory element, as described by the array in Figure 7.

Upon receiving a cache controller query, the update to the internal state may indicate the need to provide `data` from the RAM, leading the automaton to synchronize with the memory controller to wait for `RAM_READ_TIME` units of time before providing a reply to the query's originator. Alternatively, when receiving `data`, the automaton synchronizes with the memory controller to wait for `RAM_WRITE_TIME` units of time. The memory controller's automaton is shown in Figure 18. It has a local clock, `clk`, which is used to wait either `RAM_WRITE_TIME` or `RAM_READ_TIME`, depending on what the coherency manager demands.

6.4 Interconnect

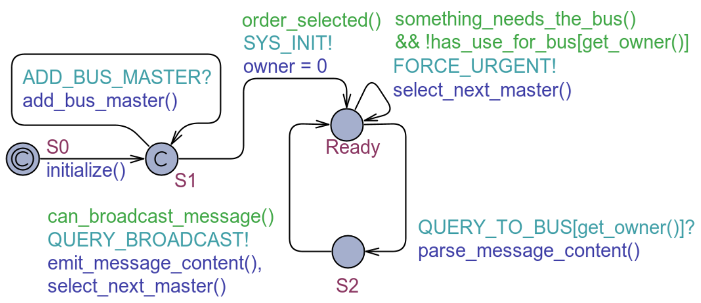


Figure 19 Model of the Interconnect.

Figure 19 shows the timed automaton for the interconnect. It starts (`S1`) by waiting for cache controllers to synchronize through the `ADD_BUS_MASTER` so that they can be added to the bus policy. The order in which the cache controllers make that synchronization is not deterministic. This results in all possible orders being explored when analyzing the system. Once all cache controllers have been added, the automaton proceeds and synchronizes with all the other components by broadcasting on the `SYS_INIT` channel.

Using a component identifier to select the appropriate sub-channel, the interconnect awaits either an incoming cache controller query, or a notice that the cache controller does not have any to send (`Ready`). If the latter happens, the access policy is followed to determine which cache controller should be made able to send its query (e.g. with a Fair-Round-Robin the next cache controller is chosen). With the former, the query is first received by the interconnect (`Ready`→`S2`), then, in a second transition (`S2`→`Ready`), it is broadcasted to all components that listen for cache controller queries. This broadcast is stalled if any of the components that need to receive it indicate that they are not ready to do so (e.g. because their incoming query queue is full).

6.5 Cache Controller

The automaton used to model a cache controller is rather complex. As previously stated, it does not feature any of the states found in the protocol description (e.g. the ones of the matrix in Figure 6). Instead, this automaton keeps an array that indicates the protocol state associated with a given memory element. The automaton starts by synchronizing with the interconnect so that it is taken into account by the interconnect's access policy (`S0`→`S1`). It then waits for the broadcast on the `SYS_INIT` channel (`S1`→`Ready`).

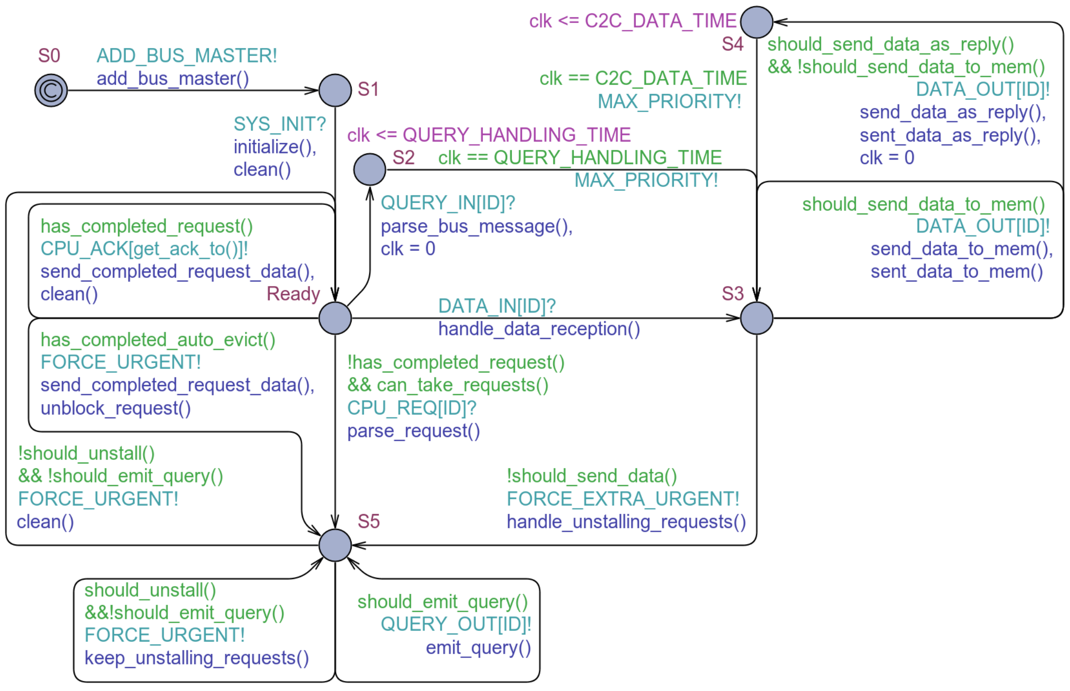


Figure 20 Model of the Cache Controller.

CPU Communications. Each cache controller has a queue of outstanding requests from its core, as well as a queue of completed requests to inform the core of. Both queues are first in, first out. Upon receiving a request from its core (middle `Ready`→`S5` transition), the cache controller attempts to find a line in its array either corresponding to the associated address, or, if none exists, one that is not currently used (*Invalid*). If no such line is found, the request is stalled, meaning that it is simply put in the outstanding requests queue for later. Otherwise, the behavior of the cache controller depends on the cache coherence protocol and the state held by the line, such as indicated in Figure 6. If the eviction policy is applicable and no line can currently accommodate the request, an automated eviction occurs. The cache controller is re-evaluated once the eviction has been completed (leftmost `Ready`→`S5` transition). In our model, we use an accurate LRU eviction policy, meaning that the cache controller keeps track of the order in which its cache lines have been used and will allow an automated eviction to occur if the least recently used line points to a state for which the protocol does not indicate stall in case of `evict` request.

There are two possible reasons for a request to be acted upon: it is an incoming request from a core, or it is a previously stalled request on a memory element which just changed state.

hit: the request is moved to the completed requests queue. The handling of stalled requests continues. This also counts as a use of the line according to the eviction policy, if the request is not an `evict`.

stall: the request is put in the outstanding requests queue, if it is not already there. The handling of stalled requests is stopped.

***msg*/*state*:** the state of the line is set to *state*, the request is put in the outstanding requests queue, if it is not already there. If this is encountered during the un-stalling of requests, the request is re-evaluated. In the latter case, this counts as a use of the line according to the eviction policy, if the request is not an `evict`.

Interconnect Communications. Handling of pending incoming queries is done through the `Ready`→`S2`→`S3` transitions. This updates the internal state of the cache following what was indicated by Figure 6 and has a waiting period that accounts for the simulated query handling time period. Handling of pending incoming data is similar (`Ready`→`S3`). The `S3` location is where data emission is handled. Data can be sent to either memory or another cache controller (the latter introducing yet another delay). This data is actually sent to a FIFO queue and not to the other components directly. When there is no data to send, the `S3`→`S5` transition evaluates the impact the changes had on the currently stalled core requests.

6.6 Message Queues

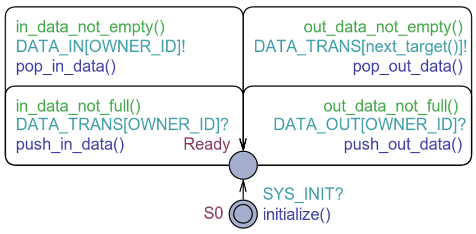


Figure 21 Model of the Data FIFOs.

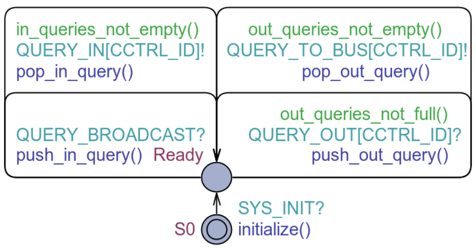


Figure 22 Model of the Query FIFOs.

Access to the bus is done through message queues. We use separate automata for data and query queues to avoid over-encumbering the automata that use them (we would otherwise need to add their transitions to nearly all the locations of the cache controller automaton). These automata actually handle both an incoming and outgoing queue. Each cache controller has a dedicated instance of both automata. The memory controller has an instance of the data queues automaton.

The data and query queues automata are fairly straightforward, having one transition to take and one transition to push items in either direction. However, the actual condition for incoming queries to be allowed in is hidden behind a shared variable. Indeed, the queries come from broadcasts made by the bus and UPPAAL does not allow conditions on transitions receiving from a broadcast channel. Thus, the condition of having all query message queues ready to receive is actually handled on the side of the interconnect.

7 Checking Properties

UPPAAL lets users check if their model verify properties. These properties can be used to know if *at least one* (`E`) or *all* (`A`) execution paths *always* ($\Box$) or *at least once* ($\Diamond$) verify a given formula over the automata's clocks, integer variables, or location. In addition, UPPAAL has an operator that looks for the highest value reachable by an automaton's clock or integer variable.

For example, taking the system from Section 5, with two CPUs ($C0$ and $C1$), we can know if both processors always end up getting their variable (all paths lead to both automata reaching the *Done* location, $A\Diamond(C0.\text{Done}\ \&\&\ C1.\text{Done})$), or the longest time it would take for one of them to do so (what is the maximum value the clock can reach before the automaton reaches its *Done* location, `sup{`not $C0$.Done`}`: $C0$.`x`).

```
program_line_t program_200 [11] =
   {
      {INSTR_STORE,   1},
      {INSTR_STORE,   2},
      {INSTR_LOAD,    1},
      {INSTR_STORE,   1},

      {INSTR_LOAD,    2},
      {INSTR_STORE,   2},
      {INSTR_LOAD,    1},
      {INSTR_STORE,   1},

      {INSTR_LOAD,    2},
      {INSTR_STORE,   2},
      {INSTR_END,     0}
   };
```

Figure 23 Program Model 200.

```
program_line_t program_201 [11] =
   {
      {INSTR_STORE,   3},
      {INSTR_STORE,   4},
      {INSTR_LOAD,    3},
      {INSTR_STORE,   3},

      {INSTR_LOAD,    4},
      {INSTR_STORE,   4},
      {INSTR_LOAD,    3},
      {INSTR_STORE,   3},

      {INSTR_LOAD,    4},
      {INSTR_STORE,   4},
      {INSTR_END,     0}
   };
```

Figure 24 Program Model 201.

7.1 Exposing Interference

Using such properties, we are able to expose the interference in a number of fashions. The example we will take for showcasing them is that of a dual core on which two instances of the program modeled by Figure 23 are running.

- **Counting Hits & Misses:** An easy metric to measure is the number of cache hits and misses for each address. This can be achieved by simply looking at the state of the memory element upon reception of a core's request, and increasing the right integer variable accordingly (much like in Section 5).
  In the dual core example, this shows that each core has 2 cache hits and 3 cache misses for the first address; one core has 2 cache hits and 3 cache misses for the second address, whereas whereas the other has 1 cache hit and 4 cache misses.
- **Counting All Occurrences:** We can expose interference by counting all of its occurrences, without regards for whether it had an impact on the system's execution or not. In effect, this equates to having one integer variable per address and type of interference, and increasing the right one according to what is described in Figure 8.
  When applied to the dual core example, we can see that for the second address, both caches have 4 occurrences of minor interference, 1 occurrence of demoting interference, and 2 occurrences of expelling ones. For the first address, one cache has 4 minors, 1 demoting, and 1 expelling, whereas the other has 3 minors, no demoting, and 3 expelling.
- **Counting Meaningful Occurrences:** Another pertinent information is an account of the interference that actually has an impact on the system. Since we are already able to detect any occurrence of the interference, we simply have to isolate the occurrences which impacted the cache's completion of core's requests. To do so, each cache keeps track, for each address, of whether an interference occurred since that address was last involved in a core request. Thus, if the CPU requests a read on an address for which the expelling flag is active, we consider that a meaningful expelling interference occurred.
  Using this with the dual core example, we can see that, for the second address, both caches are affected by the effects of 1 demoting and 1 expelling interference. For the first address, one cache has the same and the other experiences the effects of 2 expelling interferences.
- **Execution Time Analysis:** A more general metric is the execution time. Indeed, we can measure the impact that cache coherence has on an application's execution time. This can be achieved by simply replacing all accesses to shared variables made by the target

application with accesses to new variables, setting the time impact of minor interferences to nil, and having the framework compute the new maximum execution time so that it can be compared to the one with shared variables left intact.

On the dual core example, we first measure the execution time with the system as is, then replace the program running on one of the two cores by Figure 24 and set the cost of minor interferences to zero. Our first analysis indicates a maximum execution time of 1602 time units, the second one indicates 1050 time units. This implies that cache coherence causes a 16 percent increase in execution time.

Alternatively, by keeping the time impact of minor interferences to its default value, a WCET of these two programs lets us deduce how much time is lost due to minor interferences. In the dual core example, the result is still 1050 time units, showing a lack of negative effects from minor interference.

7.2 Model Validation

In addition, we can assert that the behavior of our model does indeed correspond to what we expect. The successful verification of all these properties gives us a reasonable confidence in the validity of the protocol used in our model. The validation of the chosen timing parameters, however, would still require a few judicious benchmarks.

- **Programs Always Terminate:** By checking that all possible execution paths lead all cores to the `Terminated` location, we ensured that there are no deadlocks in our model.
- **No Incompatible States:** As stated in Section 1, there should never be two cache controllers simultaneously having writing access to the same memory element. Thus, we checked that if a cache controller is in a state where it may write to a memory element, then the others are not in a state where they may read that memory element.
- **Values Are Always Up-To-Date:** Another point stated in Section 1 is that the values in cache should be up-to-date. We verified that it is the case in our model by creating a version in which the exact value of each memory element is taken into account. Using a shared variable to keep track of the expected system-wide value, we tested that every time an action (either read or write) was taken on a memory element, it the local copy of that memory element had a value equal to the system-wide one. This is a standard property to validate coherency protocol [10, 19].

8 Related Works

- **WCET Analysis for Single-Core:** The authors of [9] introduce *METAMOC*, a UPPAAL-based framework for modular WCET analysis of programs running on single-core processors. It transforms program binary executables into timed automata, one for each function of the program. These programs are simplified. For example, a conditional jump may be removed if it would lead to less instructions being executed. This is justified by the assumption that the more instructions there are, the longer the execution time is (the reverse of which is called a *time anomaly*). METAMOC supports instruction pipelines, which are modeled using five timed automata (*fetch*, *decode*, *execute*, *memory*, and *writeback*). These five automata have to be manually made for the targeted architecture. Caching is also supported, and requires a similar attention the architecture's specifics. As it is intended for single-core architectures, METAMOC obviously does not have any concept of cache coherence. We are, however, taking a very similar approach to tackle our problematic.

The work in [6] also shares similarities with [9], as UPPAAL is used to estimate WCET for programs running on single core processors with pipeline and cache, in what is presented as a modular framework. It attempts to improve on the weaknesses of METAMOC by replacing the value analysis based control flow graphs with program slicing. In effect, statements that do not affect dynamic jump addresses are replaced with `nop` (i.e. "do nothing") operations. In [7], they address the state explosion issue.

- **WCET Analysis for Multi-Core with Private Caches:** Readers can refer to [17] for an overview of Multi-Core WCET Analysis. [16], proposes a UPPAAL-based framework to estimate the WCET of applications running on a multi-core processor. They consider the delays caused by contention on the interconnect and a private instruction cache for each core (data caches are not considered). They perform analysis on the memory blocks pertinent to the instructions of the program. A memory block may contain one or more instruction. For each instruction, they are only interested in whether it: is always found in the cache; is always found except on the first access; is never found in the cache; is undecided. They have defined a timed automaton to model each of these possibilities (modeling the need for interconnect access, time to read the memory blocks, and updates to the cache). They consider programs as control flow graphs in which each node is a memory block. As such, they model each program by a single timed automaton based on the control flow graph, but in which each instruction has been replaced by one of the aforementioned timed automata corresponding to its impact. Their paper presents models for two types of interconnects: TDMA and FCFS, which control the order the bus can be accessed by the timed automata modeling the instructions. Cache coherence is not addressed.
- **WCET Analysis for Multi-Core with Shared Caches:** The authors of [8] focus on the estimation of WCET on multi-core processors. Their point of interests are the delays caused by hierarchical caches, the use of a shared cache, and the interconnect. They do not use UPPAAL, but instead model the applications as task-dependency graphs and perform computations to estimate the WCET. Their approach starts by analyzing how the L1 caches are accessed, to remove elements that are sure to always be present from further consideration. The other accesses are dependent on both the content of the L2 cache, and access to the bus. The content of the L2 cache depends on which tasks are running, which in turns, depends on bus access time access. To resolve the circular dependency, they propose an iterative approach: starting by considering the worse case scenario in which all tasks interfere, they estimate the running time of the tasks, which lets them remove any interference between two tasks whose running time are disjoint, and start over until a fix point is reached. Data caches are not taken into account and are assumed to have no effect on the calculations. Cache coherence is not addressed.

  The authors of [29] study the impact of a shared cache (including data caches) on execution time. To do so, they represent each program as an *address flow graph*, in which edges correspond to instruction, and vertices correspond to the state of the cache and its access history. They actually build a *combined cache conflict graph*, which is pretty much the combination of each core's *address flow graph* into a single graph. Cache coherence is not addressed.

  The work done in [13] has similarities with ours, as it uses UPPAAL to calculate WCET of programs running on multi-core processors. Their focus is not on cache coherence, but it does feature some, as `write` requests lead to the invalidation of the memory element in the other caches.

- **Cache Coherence Protocol Comparison:** The authors of [2] compare the efficiency of common snooping-based cache coherence protocols. To do so, they described a multi-core processor and the cache coherence protocols in *Simula*. Much like ours, the programs running on this simulation are described as a succession of memory related instructions. However, they do not use explicit addresses for these instructions. Instead, they have defined system-wide weights to regulate the probability of an instruction to be applied on a private memory element (i.e. a memory element the cache is the sole user of) or a shared block (i.e. a memory element used by multiple blocks). Thus, cache coherence *is* addressed, but only in a very broad context. Indeed, whereas our work focuses on the impact of cache coherence on specific applications on a specific architecture, the cache coherence protocol comparison made by the authors of [2] provides a general idea of which protocol is more fitted for which type of application.
- **Predictable Cache Coherence:** An alternative to trying to predict how cache coherence is going to behave is to use a kind of cache coherence designed to be predictable. [24] lists the cache coherence related latencies that need to be known before predictability of the protocol can be achieved. Its authors argue that write-through, update-based protocols (i.e. writes are propagated to other caches and to the memory) can be made to be predictable.

  [14] presents PMSI, a variation on the MSI protocol that uses a TDM bus to achieve predictability. Emission of coherence queries and is restricted to a core's TDM slots. As a result, a cache does not suffer from interference during its own TDM slots. [21] expands on this by introducing HourGlass, which allows separate handling of critical and non-critical cores. HourGlass uses timers to allow cores to hold access to a memory element for a predefined time duration. The evaluation of queries that would remove an access currently protected a timer are delayed until its time is up. Both PMSI and HourGlass require hardware modification, which prevents them from being used in a context that relies on COTS.

9 Conclusion and Future Work

When using cache coherence, the execution of a program running on a core is affected by the execution of the programs running on the other cores. Because of this, analysis of the execution time becomes much more difficult. In this paper, we categorized the types of interference that cache coherence induces: *minor interference*, caused by the handling of queries irrelevant to the cache controller; *demoting interference*, when an external event forces the loss of writing rights; and *expelling interference*, when an external event forces eviction of a cache line.

We also presented timed automata as a way to model cache coherence so that this interference can be studied and exposed. For this purpose, we also showed and explained our current model for the analysis of cache coherence, as well as the hypotheses made for that model to be applicable.

We are also working on a tool to automatically switch which MSI variant (MESI, MOSI, MOESI, MESIF) is used by the model. We also intend to add another type of instruction to programs soon, adding more non-determinism to the model by having a `INSTR_CALC` instruction that causes the CPU to wait for any amount of time in a given range. Lastly, we have planned to perform a benchmark comparison on the Keystone TCI6630K2L [23] from Texas Instruments to further validate our approach.

Our current model was tested with up to 6 cores. We are working on its scalability issues, and intend to make use of SAT/SMT [3] to tackle this limitation.

References

1 Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theor. Comput. Sci.*, 126(2):183–235, April 1994. doi:10.1016/0304-3975(94)90010-8.

2 James Archibald and Jean-Loup Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Trans. Comput. Syst.*, 4(4):273–298, September 1986. doi:10.1145/6513.6514.

3 Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. *Handbook of Satisfiability*, chapter Satisfiability Modulo Theories, pages 825–885. IOS Press, 2009.

4 Gerd Behrmann, Alexandre David, and Kim G. Larsen. *A Tutorial on Uppaal*, pages 200–236. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. doi:10.1007/978-3-540-30080-9_7.

5 Frédéric Boniol, Hugues Cassé, Eric Noulard, and Claire Pagetti. Deterministic Execution Model on COTS Hardware. In *Proceedings of the 25th International Conference on Architecture of Computing Systems (ARCS'12)*, pages 98–110, 2012.

6 Franck Cassez and Jean-Luc Béchennec. Timing Analysis of Binary Programs with UPPAAL. In *13th International Conference on Application of Concurrency to System Design, ACSD 2013*, pages 41–50. IEEE Computer Society, July 2013. doi:10.1109/ACSD.2013.7.

7 Franck Cassez and Pablo González de Aledo Marugán. Timed Automata for Modelling Caches and Pipelines. In Rob J. van Glabbeek, Jan Friso Groote, and Peter Höfner, editors, *Proceedings Workshop on Models for Formal Analysis of Real Systems, MARS 2015, Suva, Fiji, November 23, 2015.*, volume 196 of *EPTCS*, pages 37–45, 2015. doi:10.4204/EPTCS.196.4.

8 Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. Modeling Shared Cache and Bus in Multi-cores for Timing Analysis. In *Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems*, SCOPES '10, pages 6:1–6:10, New York, NY, USA, 2010. ACM. doi:10.1145/1811212.1811220.

9 Andreas E. Dalsgaard, Mads Chr. Olesen, Martin Toft, René Rydhof Hansen, and Kim Guldstrand Larsen. METAMOC: Modular Execution Time Analysis using Model Checking. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 113–123, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7. doi:10.4230/OASIcs.WCET.2010.113.

10 Giorgio Delzanno. Automatic Verification of Parameterized Cache Coherence Protocols. In *Proceedings of the 12th International Conference on Computer Aided Verification*, CAV '00, pages 53–68, London, UK, UK, 2000. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=647769.734088.

11 Philip Enslow, Jr. Multiprocessor Organization - a Survey. *ACM Comput. Surv.*, 9(1):103–129, March 1977.

12 Sylvain Girbal, Xavier Jean, Jimmy le Rhun, Daniel Gracia Pérez, and Marc Gatti. Deterministic Platform Software for Hard Real-Time systems using multi-core COTS. In *34th Digital Avionics Systems Conference (DASC'15)*, 2015.

13 Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET Analysis of Multicore Architectures Using UPPAAL. In *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, pages 101–112, 2010. doi:10.4230/OASIcs.WCET.2010.101.

14 Mohamed Hassan, Anirudh M. Kaushik, and Hiren D. Patel. Predictable Cache Coherence for Multi-core Real-Time Systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2017, Pittsburg, PA, USA, April 18-21, 2017*, pages 235–246, 2017. doi:10.1109/RTAS.2017.13.

15 Xavier Jean, David Faura, Marc Gatti, Laurent Pautet, and Thomas Robert. Ensuring robust partitioning in multicore platforms for IMA systems. In *31st Digital Avionics Systems Conference (DASC'16)*, 2012.

16 M. Lv, W. Yi, N. Guan, and G. Yu. Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software. In *2010 31st IEEE Real-Time Systems Symposium*, pages 339–349, November 2010. `doi:10.1109/RTSS.2010.30`.

17 Claire Maiza, Hamza Rihani, Juan Maria Rivas, Joël, Godelieve Goossens, Sebastian Altmeyer, and Robert I. Davis. A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. Technical report, Grenoble INP/Ensimag/Verimag, 2018.

18 Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A Predictable Execution Model for COTS-Based Embedded Systems. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium RTAS 2011*, pages 269–279, 2011.

19 Fong Pong and Michel Dubois. A New Approach for the Verification of Cache Coherence Protocols. *IEEE Trans. Parallel Distrib. Syst.*, 6(8):773–787, August 1995. `doi:10.1109/71.406955`.

20 Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence.* Morgan & Claypool Publishers, 1st edition, 2011.

21 Nivedita Sritharan, Anirudh M. Kaushik, Mohamed Hassan, and Hiren D. Patel. HourGlass: Predictable Time-based Cache Coherence Protocol for Dual-Critical Multi-Core Systems. *CoRR*, abs/1706.07568, 2017. `arXiv:1706.07568`.

22 V. Suhendra, T. Mitra, and A. Roychoudhury and. WCET centric data allocation to scratchpad memory. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 10 pp.–232, December 2005. `doi:10.1109/RTSS.2005.45`.

23 Texas Instruments. TCI6630K2L Multicore DSP+ARM KeyStone II System-on-Chip. Technical Report SPRS893E, Texas Instruments Incorporated, 2013.

24 Sascha Uhrig, Lillian Tadros, and Arthur Pyka. MESI-Based Cache Coherence for Hard Real-Time Multicore Systems. In Luís Miguel Pinho Pinho, Wolfgang Karl, Albert Cohen, and Uwe Brinkschulte, editors, *Architecture of Computing Systems – ARCS 2015*, pages 212–223, Cham, 2015. Springer International Publishing.

25 L. Wehmeyer and P. Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *Design, Automation and Test in Europe*, pages 600–605 Vol. 1, March 2005. `doi:10.1109/DATE.2005.183`.

26 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-case Execution-time Problem - Overview of Methods and Survey of Tools. *ACM Transactions Embedded Computing Systems*, 7(3):36:1–36:53, May 2008.

27 Reinhard Wilhelm and Jan Reineke. Embedded systems: Many cores - Many problems. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, pages 176–180, 2012.

28 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'13)*, pages 55–64, 2013.

29 Wei Zhang and Jun Yan. Static Timing Analysis of Shared Caches for Multicore Processors. *JCSE*, 6(4):267–278, 2012. `doi:10.5626/JCSE.2012.6.4.267`.

Arbitration-Induced Preemption Delays

Farouk Hebbache
CEA, List, 91191 Gif-sur-Yvette, France
farouk.hebbache@cea.fr

Florian Brandner
LTCI, Télécom ParisTech, Université Paris-Saclay, France
florian.brandner@telecom-paristech.fr

Mathieu Jan
CEA, List, 91191 Gif-sur-Yvette, France
mathieu.jan@cea.fr

Laurent Pautet
LTCI, Télécom ParisTech, Université Paris-Saclay, France
laurent.pautet@telecom-paristech.fr

Abstract

The interactions among concurrent tasks pose a challenge in the design of real-time multi-core systems, where blocking delays that tasks may experience while accessing shared memory have to be taken into consideration. Various memory arbitration schemes have been devised that address these issues, by providing trade-offs between predictability, average-case performance, and analyzability. Time-Division Multiplexing (TDM) is a well-known arbitration scheme due to its simplicity and analyzability. However, it suffers from low resource utilization due to its non-work-conserving nature. We proposed in our recent work dynamic schemes based on TDM, showing work-conserving behavior in practice, while retaining the guarantees of TDM. These approaches have only been evaluated in a restricted setting. Their applicability in a preemptive setting appears problematic, since they may induce long memory blocking times depending on execution history. These blocking delays may induce significant jitter and consequently increase the tasks' response times.

This work explores means to manage and, finally, bound these blocking delays. Three different schemes are explored and compared with regard to their analyzability, impact on response-time analysis, implementation complexity, and runtime behavior. Experiments show that the various approaches behave virtually identically at runtime. This allows to retain the approach combining low implementation complexity with analyzability.

2012 ACM Subject Classification Computer systems organization → Real-time system architecture

Keywords and phrases Dynamic Time-Division Multiplexing, Predictable Computing, Multi-Criticality, Preemption

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.19

1 Introduction

Multi-core architectures pose many challenges in real-time systems, which arise from the manifold interactions between concurrent tasks during their execution – most notably accesses to shared main memory. These interactions make it difficult to tightly bound the Worst-Case Execution Time (WCET) of real-time tasks. Systematically considering the worst-case behavior of an arbitration policy with regard to memory accesses in the presence of concurrent requests is too pessimistic, as it leads to low resource utilization at runtime. Considering Mixed-Criticality (MC) systems is one approach to increase resource utilization. In an MC system, tasks with different levels of criticality, and even non-critical tasks, execute on the same multi-core architecture. The non-critical tasks in such a system may then exploit resource budgets that have not been fully consumed by critical tasks – which rarely fully exploit the worst-case budgets reserved for them. However, most MC systems take decisions only at the level of task scheduling. Our aim here is, hence, to apply similar ideas to the arbitration of memory requests in order to improve memory utilization.

Amongst the memory arbitration policies, Time-Division Multiplexing (`TDM`) divides time into slots and allocates them to cores to exclusively access memory. Using `TDM`, the accesses within a slot no longer depend on whether concurrent requests exist or not. `TDM` provides predictable behavior and improves composability by bounding access latencies and guaranteeing bandwidth independently from other cores. However, the access latency of a memory request when using `TDM` now depends on the scheduling of these time slots, even if they are unused. Such unused slots appear when the owner of a `TDM` slot does not (yet) have a memory request ready to be served. Under a strict `TDM` scheme, these unused slots cannot be reclaimed by another task. This *non-work-conserving* behavior of `TDM` often leads to low resource utilization. This problem is further amplified as the number of cores increases, leading to longer `TDM` schedules. Another source of `TDM` pessimism stems from the length of `TDM` slots, expressed in clock cycles, which have to be longer than the worst-case latency for handling memory requests. The arbiter consequently has to wait for the beginning of the next `TDM` slot, even when memory requests complete earlier than this worst-case latency. Resulting in low average-case performance and poor memory utilization.

To overcome these limitations, we recently explored novel dynamic `TDM`-based arbitration schemes [13]. In that work, we envisioned that the task's criticality level should not only be used by the task scheduler, but also by the memory arbiter. The arbiter associates a deadline with each memory request of a critical task, which corresponds to the end of its corresponding slot under a strict `TDM` scheme. The deadline then allows to compute the *slack time* of a critical task, i.e., the amount of time that the task's last request completed before the request's deadline. This slack time then can be exploited by the arbiter, under certain conditions, to favor requests of non-critical tasks over requests from critical tasks, to freely reorder memory requests, and to schedule memory requests at the granularity of clock cycles – instead of being confined to `TDM` slots. The resulting arbiter significantly reduces the memory idle time, compared to a regular `TDM` arbiter. The improvements go up to a factor of 350 and even remain above a factor of 50 under high memory load [13].

However, the proposed dynamic `TDM`-based arbitration techniques face issues under a preemptive execution model. In this paper, we define two memory delays induced by preemptions, the *memory blocking delay* and the *misalignment delay*, which may lead to significant jitter and increase task response times. Even worse, due to non-critical tasks, the memory blocking delay may be unbounded in some circumstances. We explore three different approaches to analyze the impact of these arbitration-induced preemption delays considering preemptive [18] (`SHDp`) and non-preemptive [1] (`SHDw`) memory requests. Finally, we propose a new technique (`SHDi`) to resolve these issues by adapting (priority or rather) *criticality inheritance* known from scheduling theory. This allows us to manage and easily bound these preemption delays. Our evaluation shows that our new approach successfully limits the worst-case preemption delays experienced at runtime under our dynamic `TDM`-based arbitration schemes. At the same time we see virtually no impact on average-case performance and success rate. Note, in addition, that the proposed technique is not limited to the dynamic `TDM`-based arbitration schemes and is also applicable to other arbitration techniques, e.g., arbitration based on fixed priorities [1].

The remainder of this paper is organized as follows. Section 2 describes the considered system model. In Section 3, we briefly review the dynamic `TDM`-based arbitration schemes. Section 4 identifies the different delays caused by preemptions in dynamic `TDM`-based arbitration strategies, while Section 5 proposes three preemption models to handle these delays. In Section 6, we evaluate our contributions and demonstrate the improvements in terms of blocking delay reductions and success rates on randomly generated task sets. Section 7 presents related work, before concluding in Section 8.

2 System Architecture

2.1 Task Model

In our previous work [12, 13] on TDM-based dynamic memory arbitration, we assumed a restricted task model, where each core executes a single independent and periodic task without preemption. Here, we relax these assumptions to allow multiple tasks to be scheduled on each core under a fixed-priority preemptive scheduler. We assume a finite task set Γ consisting of independent and periodic tasks. Each task $\tau_i \in \Gamma$ is assigned a fixed priority[1] i and is characterized by the tuple $\tau_i = (C_i, T_i, D_i, M_i)$, representing the task's WCET, its period, its implicit deadline, and finally its worst-case number of memory requests respectively. Each task generates an infinite sequence of jobs at runtime, which, in turn, generate sequences of memory requests $\tau_{i,j}$ where $j \leq M_i$. Requests are separated by a dynamic number of processor clock cycles (see Section 3). This *distance* represents the amount of computation performed between the completion of request $\tau_{i,j}$ and the issuing of the consecutive request $\tau_{i,j+1}$ or the job's termination. This allows us to model any *dynamic* job execution (even input-dependent) considering a deterministic hardware platform (see below).

Ultimately we aim at mixed-criticality systems, where tasks can be associated with multiple WCETs depending on the system's execution mode (criticality). This work focuses on the impact of the memory arbitration scheme on such MC systems, more precisely the interplay between dynamic TDM-based arbitration and task preemptions. We consider two classes of tasks: *critical* tasks τ_i^c and *non-critical* tasks τ_i^{nc}. However, we currently only consider a single C_i value per task in our task model, i.e, we assume a multi-criticality system. Our experiments are based on simulations to evaluate memory arbitration under realistic circumstances. We here assume that the WCETs (C_i) and deadlines (D_i) of critical tasks are firm and have to be respected under all circumstances. An execution thus fails if a critical task misses its deadline. Non-critical tasks, on the other hand, are executed in a best-effort manner by the underlying computer platform and memory arbitration scheme. During execution, they may exceed their WCET budget, potentially causing deadline misses – for themselves or other (critical) tasks. Handling timing failure events, in particular switching critical tasks to a different behavior as in MC scheduling [5], is out of the scope of this work. It will be addressed in future work. For the formal analysis, based on response-time analysis (RTA) [2], we simply require (for now) *firm* WCETs and deadlines for non-critical tasks in order to bound the response times of critical tasks.

2.2 Hardware Architecture

We assume a hardware platform consisting of m cores connected via a central arbiter to a shared main memory. The memory requests dynamically generated by the jobs at runtime thus compete for this shared resource. We assume that each core is equipped with internal caches. Memory requests thus represent transfers of cache blocks resulting from cache misses.

[1] A larger task index indicates higher priority.

In order to ensure that the aforementioned *distances* between requests are independent from the execution of other tasks, we assume composable compute cores [9] and the absence of any external events that may interfere with the execution of a core. The interference between the independent tasks consequently stems from accesses to the shared memory only and, in particular, depends on the employed memory arbiter.

For simplicity, we assume that all cores, the memory bus/arbiter, and the memory itself operate at the same clock speed. We thus generally refer to time in *clock cycles*.

2.3 Scheduling Policy

On multi-core platforms, task scheduling can be performed globally among all/a subset of the available m cores [16] or in a partitioned manner [3, 4]. Partitioned scheduling statically assigns each task onto a fixed core, while global scheduling allows tasks to migrate among cores dynamically. In principle both of these scheduling policies could be combined with the approaches proposed here. For brevity, we limit our discussion on *partitioned scheduling* using *fixed priorities*.

This enables the scheduler on each core to determine at any moment in time and *in advance*, which task will need to be activated next on its core. Instead of triggering preemptions periodically, we assume that the scheduler programs a hardware component that signals the need to preempt the currently running task to both the core and the memory arbiter. This ensures that preemption handling does not interfere with the execution on a given core – up until to the moment when a preemption is triggered. We assume that each core is equipped with such a component, separately tracking the next upcoming preemptions stemming from a non-critical or a critical task. This also ensures that the preemption mechanism does not interfere with the computation (*distance*) between memory accesses.

3 Background

This section provides a brief introduction to our recent work on TDM-based dynamic memory arbitration as well as well-established iterative techniques for response-time analysis.

3.1 TDM-Based Dynamic Memory Arbitration

TDM-based arbitration is popular due to its simplicity and predictability, despite the fact that strictly applying TDM often results in under-utilization of resources. Our recent work on dynamic TDM-based memory arbitration schemes [12, 13] addresses resource under-utilization by (a) introducing two classes of memory requests (critical and non-critical) and (b) by exploiting slack to allow for more dynamic scheduling decisions as long as the deadlines of critical requests can be met safely.

The key idea of the *dynamic TDM with slack counters* (TDMds) arbitration scheme [12] is to interpret TDM scheduling as driven by deadlines. Under strict TDM each request completes precisely at the end of the request owner's next TDM slot, which can be seen as a deadline. The deadlines of TDMds similarly correspond to the end of TDM slots. However, instead of systematically delaying requests until their respective deadlines, requests are processed dynamically in any order – as long as deadlines are met. This allows to compute the *slack time* of a critical task, i.e., by how much the task's last request completed earlier w.r.t. the request's deadline. This slack is stored in dedicated counters and allows to prioritize non-critical requests, i.e., spend the slack of a critical task in favor of a non-critical task. Note, however, that slack accumulated within one job of a task naturally is not preserved for subsequent jobs. Slack counters are consequently reset at job start. Also note that deadlines are ensured for critical requests only, while non-critical requests are processed in a best-effort manner. The dynamic processing allows non-critical tasks, for instance, to reclaim otherwise unused `TDM` slots.

A critical request may arrive earlier than expected under a strict `TDM` scheme if the request's owner accumulated some slack before. The request's deadline would then also appear earlier than under strict `TDM` if one would simply consider the request's issue date to compute the deadline, i.e., the end of the next `TDM` slot after the issue date. Under `TDMds` the deadline of a new critical request is instead computed from a *delayed issue date*, which is derived by adding the previously accumulated slack to the request's original issue date. Given enough slack this may push the request's deadline into the future, i.e., past the next `TDM` slot, and provide additional freedom to the arbiter. This also ensures that the deadlines under `TDMds` **exactly** correspond to the deadlines/completion dates that would be observed under strict `TDM`. This holds for **any** dynamic execution of **any** program [13].

The `TDMds` arbiter keeps critical requests in a priority queue (ordered by increasing deadlines) and non-critical requests in a simple FIFO. At the beginning of each `TDM` slot the arbiter schedules the top-most request from one of these two queues in order to be processed by the shared memory. The arbiter prioritizes non-critical requests over critical requests, as long as the deadline of the top-most critical request is *not* at the end of the current `TDM` slot. This ensures that non-critical requests can quickly access memory as long as critical requests have enough slack, while also guaranteeing that deadlines of critical requests are met.

An extension of `TDMds`, called *dynamic* `TDM` *with early release* (`TDMer`) even goes a step further by performing arbitration at the level of (bus) clock cycles instead of `TDM` slots [13]. The key insight is that memory can begin the processing of a request at any moment (even in the middle of a slot), if the arbiter can ensure that subsequent critical requests do not miss their deadlines. Two cases can be distinguished. Firstly, consider that memory is idle during `TDM` slot i, while a request by the owner of the upcoming `TDM` slot $i + 1$ is pending. This request can immediately be processed, as the `TDM` slot i is unused and the request is guaranteed to finish before its deadline, which may not lie before the end of `TDM` slot $i + 1$. The processing cannot interfere with requests of other slots (e.g., $i + 2$). A similar argument can be built for arbitrary pending requests, if the owner of the upcoming `TDM` slot has enough slack (e.g., more than a slot length). In this case, the processing of another request can start in the middle of the unused `TDM` slot i and complete in slot $i + 1$. This may, in the worst-case, delay a request of the owner of slot $i + 1$. However, the deadline of the delayed request, under all circumstances, falls into the next `TDM` period, due the use of the *delayed issue date* to derive its deadline. Exploiting these two cases, `TDMer` was shown to be work-conserving even under high memory load, when every job of critical tasks is granted an initial slack counter value of a single slot length [13] (not shown in the following example).

Figure 1 shows an execution of a task set under `TDMer`, considering two critical tasks (τ_0^c, τ_1^c) and a non-critical task (τ_2^{nc}). All tasks execute on separate cores, which issue critical and non-critical requests on behalf of the respective tasks. Only critical tasks have dedicated `TDM` slots (`C0`, `C1`) that take 8 clock cycles each and 16 clock cycles for both, corresponding to the `TDM` slot length Sl and the `TDM` period P respectively. The non-critical task thus may only access memory during unused slots or when the slack of critical tasks permits. The visualization of a request includes the request's *issue date* (↰) and *deadline* (⋯]). The processing of a request by the main memory is indicated using a solid green hatched bar (▨), whose right edge indicates the request's *completion date*. The memory is not always busy, memory idling is indicated by a red hatched bar (▨). The deadline may lie after the

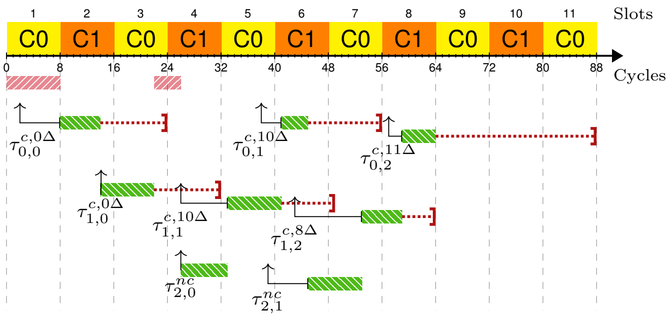


Figure 1 Dynamic arbitration using `TDMer` of critical tasks τ^c_0 and τ^c_1 and non-critical task τ^{nc}_2.

request's completion date, which then generates slack for the task issuing the request. The value of the slack counter when issuing a request is displayed as a superscript. For instance, task τ^c_0 accumulated 11 cycles of slack due to the early completion of requests $\tau^c_{0,0}$ and $\tau^c_{0,1}$. When issuing request $\tau^c_{0,2}$ this slack (superscript 11Δ) pushes the request's deadline beyond the next `TDM` slot owned by τ^c_0 (beyond slot 9 to 11). The arbiter is no longer tied to `TDM` slots and can choose one of the issued requests, independently from the actual owner of the slot and the alignment with the `TDM` schedule – given enough slack. This is illustrated by the non-critical request $\tau^{nc}_{2,1}$, which is issued in `TDM` slot 5. The request first has to wait for the ongoing request $\tau^c_{1,1}$ to complete. Then, $\tau^c_{0,1}$ is prioritized, since τ^c_0 is the owner of the upcoming `TDM` slot 7. At the moment when the processing of $\tau^{nc}_{2,1}$ starts, τ^c_0 is the owner of the upcoming slot 7, but has no pending request. However, any request issued by τ^c_0 is known to have a deadline that lies beyond slot 7, due to its slack counter value of 11. It is thus safe to begin the processing of $\tau^{nc}_{2,1}$. We can also see that the non-critical request is prioritized over the pending critical request $\tau^c_{1,2}$, as τ^c_1 has accumulated 8 cycles of slack. Note, that the slack counters are all reset for subsequent job instances of the critical tasks τ^c_0 and τ^c_1.

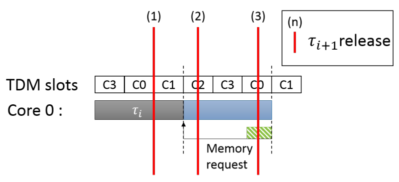
3.2 Worst-Case Response Time Analysis

We assume that schedulability is verified using a variant of *Response-Time Analysis* (RTA) [2] based on the usual recurrence equations:

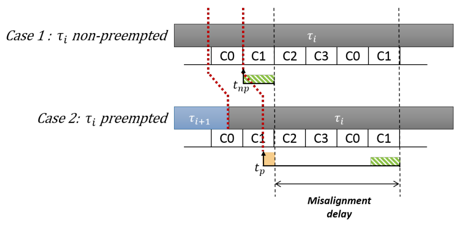
$$R_i^{n+1} = C_i + B_i + \sum_{\forall j \in \mathrm{hp}(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \tag{1}$$

The recurrence equations are initialized to $R_i^0 = 0$ and then iteratively reevaluated until a fixed point is reached. R_i then indicates the response time of task i, having a WCET bound C_i (see Section 2). In addition, the impact of preemptions by tasks with higher priority than i ($j \in \mathrm{hp}(i)$) is considered via their WCET bounds C_j and periods T_j. B_i indicates an upper bound on the time task i may be blocked (e.g., by semaphores). However, we will ignore this part of the equation in the remainder of this paper and instead consider additional delays potentially caused by the memory arbitration policy.

The dynamic memory arbitration approaches presented above may induce additional preemption delays and *jitter* that need to be considered during schedulability analysis in order to be certain that the tasks actually meet their deadlines safely. Both issues are caused by the way memory requests are processed as explained in the following section.



(a) Three different scenarios of τ_{i+1} preempting τ_i at release dates (1), (2), and (3).



(b) Alignment of task τ_i w.r.t. to its TDM slot C1 without (top) and with preemption (bottom).

Figure 2 Preemption effects w.r.t. TDM memory arbitration.

4 Arbitration-Induced Preemption Delays

We now investigate the issues raised by TDM-based arbitration in general and our dynamic schemes [12, 13] in particular, considering the preemptive system model from Section 2. For brevity, we will focus on preemption costs caused by the arbitration policy and ignore other costs due to the scheduler invocation, context switching, pipeline flushes, or Cache-Related Preemption Delays (CRPD). From here on, we use the term *preemption cost* to refer to the costs related to the *arbitration policy only*.

4.1 Preemption Costs for strict TDM Arbitration

Subfigure 2a depicts three preemption scenarios of a task τ_{i+1} that preempts a lower-priority task τ_i at different release dates (red vertical lines). The first case (1) refers to a preemption occurring while the CPU performs computations (gray area). The task then can be preempted right away (ignoring potential pipeline stalls). The second case (2) refers to a situation where the preemption occurs while the CPU stalls, waiting to access the shared memory. While it would be possible to abort the pending memory request and immediately preempt τ_i [18], this requires modifications to the processor pipeline. In the last case (3), the preemption occurs while the memory actually processes a memory request. It would theoretically be possible to also abort the request at this stage – requiring modifications to the processor pipeline and throughout the entire memory hierarchy. A simpler alternative for cases (2) and (3), both in terms of hardware and timing analysis, would be to avoid aborting the request and simply wait for its completion [1].

Clearly, all three cases may induce preemption-related delays that need to be bounded and taken into consideration by the schedulability test (in addition to classical CRPD). The worst case delay experienced for case (1) is *trivial* and only depends on the characteristics of the processor pipeline. Case (3) similarly is analyzable and can be bounded by the worst-case memory latency, e.g., a TDM slot length [1]. The analysis of case (2) is more complex, since the behavior potentially depends on the other tasks in the system and the memory arbitration policy. We first analyze the timing behavior for strict TDM and later extend this analysis to the dynamic TDM-based approaches.

▶ **Definition 1.** *Under a system with fixed-priority preemptive scheduling, the* ***memory blocking delay*** *(MB) denotes the number of clock cycles that a higher-priority task τ_h is blocked from executing on its core after its release, due to a pending memory request by a lower-priority task τ_l ($l < h$).*

However, the blocking time does not cover all preemption-related delays of strict `TDM`. Recent work [14, 19] proposed sophisticated WCET analyses, which exploit the relative alignment of the program execution with regard to the `TDM` schedule, without preemption. A preemption may impact the program's relative alignment – unless task scheduling itself is aligned with the `TDM` period.

▶ **Definition 2.** *The **misalignment delay** (MA) denotes the number of additional clock cycles that the first memory access of a task takes, w.r.t. the worst case considered by the WCET analysis, when resuming after a preemption.*

Subfigure 2b depicts such a misalignment delay under strict `TDM` for a request of a critical task τ_i, owning `TDM` slot `C1`. Case 1 (top) illustrates an execution without preemption, where a request is issued right at the beginning of the task's `TDM` slot at time instant t_{np}. The program's alignment w.r.t. the `TDM` schedule is ideal and the request is processed immediately. The second case (bottom) shows the *same* execution of τ_i after a preemption by τ_{i+1}. In the absence of other side-effects, such as CRPDs, the same computations are performed by τ_i up to its first memory request (as indicted by the red dotted lines), which now is issued at time instant t_p. However, the task's alignment w.r.t. the `TDM` schedule was slightly shifted due to the preemption. The request thus has to wait longer than expected by the WCET analysis, which assumed an execution without preemption.

4.2 Preemption Costs for Dynamic TDM-based Arbitration

The dynamic TDM-based arbitration schemes [12, 13] inherit the memory blocking and misalignment delays from strict `TDM`. Figure 3 shows 3 tasks executing on 2 cores that share slots `C0` and `C1` within a `TDM` period under the `TDMds` arbitration scheme. The mapping between tasks and cores is indicated by matching colors (yellow 🟨 and orange 🟧). Critical tasks τ_0^c and τ_1^c are executed on the same core, which results in a preemption of task τ_0^c by τ_1^c (▤). Non-critical task τ_2^{nc} executes alone on the other core. A core has a dedicated `TDM` slot when it executes a critical task (τ_1^c, τ_0^c), while the slots of cores executing non-critical tasks (here only τ_2^{nc}) are *shared* by the running non-critical tasks (see the next section).

Lets assume that tasks cannot be preempted while performing a memory access [1]. Task τ_0^c then blocks the higher-priority task τ_1^c after its release in `TDM` slot 4. The preemption's blocking time, highlighted by the blue cross-hatched bar (▩), appears to be larger than the worst-case memory access latency under strict `TDM`. The latency of request $\tau_{0,2}^c$ amounts to two entire `TDM` periods and τ_1^c starts executing only after its completion. The reason for this long delay is the high slack counter value ($\Delta 16$) for request $\tau_{0,2}^c$, combined with the interference of the non-critical task τ_2^{nc} running on the other core. The non-critical task

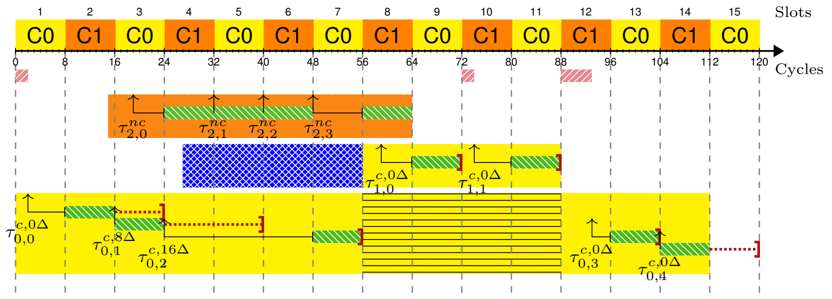


Figure 3 Impact of slack accumulation on the memory blocking time for `TDMds`.

consumes all the slack of the critical request $\tau^c_{0,2}$, delaying its completion, and thus also delaying the preemption. The situation is potentially even worse when a non-critical task is preempted during a memory request. Recall that non-critical requests are executed in a best-effort manner. The blocking time caused by such a request may even be unbounded.

The example illustrates that preemption-induced delays need to be taken into consideration during system analysis. Note that the same reasoning (see Section 4.1) used for strict `TDM` to establish the misalignment delay bound applies for all dynamic `TDM`-based approaches considered here. Next, we investigate three different approaches to integrate the memory blocking delay due to preemptions for our dynamic `TDM`-based arbitration schemes by extending response-time analysis. In addition, we compare the approaches w.r.t. their implementation complexity and actual runtime behavior.

5 Arbitration-Aware Preemption Techniques

Let us assume that memory requests are never aborted – cf. cases (2) and (3) from Subfigure 2a. In this case, preemptions have to be delayed until a potentially ongoing request completes. Under strict `TDM` arbitration, this corresponds to the usual worst-case memory access latency w.r.t. the `TDM` period P and `TDM` slot length Sl, which can be bounded by [1]:

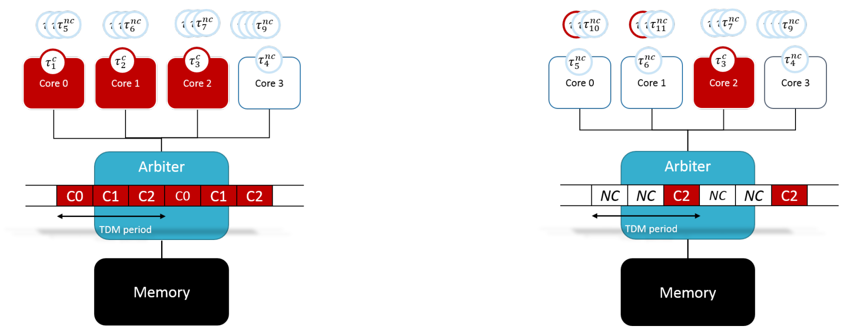
$$\mathrm{MB}^{\texttt{TDM}} = P + Sl - 1 \tag{2}$$

As demonstrated in Section 4, this bound is not valid for `TDMds` and `TDMer`, due to additional delays caused by the slack counters. While it certainly appears feasible to determine bounds on the slack counter values of tasks, it can be expected that these bounds would be rather pessimistic. In this section, we first explore this option to simply *wait* [1] due to its uncomplicated hardware implementation and later refer to it as `SHDw` (for *ScHeDuling wait*). Next, we explore an alternative option (`SHDp`, for *ScHeDuling preempt*), which assumes that ongoing requests can be *preempted* as long as they are not yet processed by the main memory [18] – cf. case (2) from Subfigure 2a. Finally, we propose a new solution (`SHDi`, for *ScHeDuling inheritance*) that tries to limit the impact of the slack counters by imposing a (new) deadline on an ongoing request when the core's timer signals a preemption by a critical task, i.e., the request of the preempted task *inherits* the criticality of the preempting task. However, before discussing these preemption models, we first need to refine the architecture model considering the fact that the cores can be shared by the tasks. Finally, we show how these models can be integrated into a response time analysis.

5.1 `TDM` Schedule and Preemption

We assume a multi-core platform with m cores and partitioned fixed-priority scheduling on each core. Critical and non-critical tasks may reside on the same core – without any restrictions on priority assignment. Notably, non-critical tasks may have a higher priority than critical tasks.

This raises the question on how `TDM` slots are assigned among cores/tasks. We propose a pragmatic solution for this work, but other alternatives are obviously possible. Since each core that executes at least one critical task may require a `TDM` slot at some moment, we assign one `TDM` slot to each such core. However, the slot is only reserved exclusively for that core when it actually executes a critical task, i.e., the core is considered critical. Cores that do not execute a critical task at a specific moment are themselves non-critical. The `TDM` slots that are not reserved by critical cores are marked as NC and shared among all running tasks on all cores in the system. This strategy is illustrated by Figure 4, showing a system with



(a) All `TDM` slots are reserved by critical cores. **(b)** One reserved `TDM` slot, two are shared (*NC*).

Figure 4 Hardware architecture.

4 cores. The `TDM` schedule consists of three slots (`C0`, `C1`, and `C2`), since only three of the cores may execute critical tasks (red). In Subfigure 4a, these three cores execute critical tasks and are thus themselves critical. In Subfigure 4b, on the other hand, only a single critical task executes. Consequently two of the `TDM` slots are marked *NC* and shared by all running tasks. Note, that non-critical tasks on core 3 may suffer from starvation on the depicted platform. For the formal analysis we require that at least one `TDM` slot is marked *NC* whenever a non-critical task executes in order to rule starvation out. This can be seen as a form of hierarchical arbitration similar to the work by Paolieri et al. [18] on Round-Robin. We thus define a larger `TDM` period for non-critical tasks using an application-specific constant k (e.g., k represents the number of non-critical cores divided by the number of *NC* slots when applying FIFO arbitration among non-critical requests): $P^{nc} = k \cdot P$.

Also, recall that we assume that preemptions are pre-programmed through a timer-like hardware component (see section 2.3). Using these components, we can control under which conditions preemptions are actually triggered, e.g., to block the current preemption until an ongoing memory request has completed or to preempt a pending request, et cetera.

5.2 Scheduling with Request Waiting (`SHDw`)

This strategy simply waits that an on-going memory request finishes [1]. Compared to strict `TDM`, the memory blocking time can however be considerably larger due to the slack counters. Consequently, a timing analysis technique is required that allows to bound the maximum slack counter value of a critical task τ_i^c. A trivial bound Δ_i^{max} can be computed by multiplying a task's worst-case number of memory requests (M_i) with the maximum slack possibly accumulated per request: $M_i \cdot (P - Sl)$ (`TDMds`) or $M_i \cdot (P + Sl - 1 - l)$ (`TDMer`), where l indicates the minimum memory latency. The resulting bound appears highly pessimistic, but more precise bounds are out of the scope of this work.

To bound the memory blocking time of a critical task τ_i^c two cases need to be considered. Firstly, the blocking delay of preempting some lower-priority non-critical tasks (lpnc(i)) has to be considered via $\text{MB}_i^{nc,\texttt{SHDw}}$ – which is similar to Equation 2 for strict `TDM`. Secondly, the blocking delay of preempting another lower-priority critical task (lpc(i)) has to be considered via $\text{MB}_i^{c,\texttt{SHDw}}$. Here, the maximum slack counter values (Δ_l^{max}) over all lower-priority critical tasks τ_l^c, has to be considered in addition to the cost of a single memory access as follows:

$$\mathrm{MB}_i^{c,\mathtt{SHDw}} = \begin{cases} 0 & \text{if } \mathrm{lpc}(i) = \emptyset, \\ P + Sl - 1 + \max\limits_{l \in \mathrm{lpc}(i)} \Delta_l^{max} & \text{else} \end{cases}$$

$$\mathrm{MB}_i^{nc,\mathtt{SHDw}} = \begin{cases} 0 & \text{if } \mathrm{lpnc}(i) = \emptyset, \\ P^{nc} + Sl - 1 & \text{else} \end{cases}$$

$$\mathrm{MB}_i^{\mathtt{SHDw}} = \max(\mathrm{MB}_i^{c,\mathtt{SHDw}},\ \mathrm{MB}_i^{nc,\mathtt{SHDw}}) \tag{3}$$

The main advantage of this approach is its simplicity in terms of hardware complexity. The timer-like component triggering the preemption on behalf of the task scheduler simply has to detect whether a memory request is pending and, if so, delay the preemption until the request completes. The downside is that it requires precise information on slack counters, so a complex analysis that appears to go against a simple analysis, main advantage of `TDM`.

5.3 Scheduling with Request Preemption (`SHDp`)

An alternative approach is to preempt ongoing memory requests. We consider that requests can only be preempted while pending at the arbiter, but not while being processed by the memory [18]. Consequently, preemptions are still delayed when a request is currently processed by the memory (cf. case 3 of Subfigure 2a). The memory blocking delay for critical and non-critical tasks then can trivially be bounded by the worst-case memory latency, which in turn is bounded by Sl:

$$\mathrm{MB}_i^{\mathtt{SHDp}} = Sl - 1 \tag{4}$$

However, the preempted task later has to reissue the memory request that was preempted, which causes additional delays that need to be accounted for in its response time. A trivial bound for this reissuing delay is the `TDM` period P, i.e., the maximum latency of a pending request after the preemption. Note that there is no need to consider the slack counter value, since it is already covered by the WCET. Furthermore, the misalignment delay always applies to the preempted/reissued request. The misalignment delay thus already covers this overhead (see Subsection 5.5).

Figure 5 shows an execution of the same task set introduced in Subsection 4.2 using `TDMds` and the `SHDp` preemption scheme. This time request $\tau_{0,2}^c$, waiting to access the memory, is immediately preempted by task τ_1^c and is reissued once τ_0^c resumes execution. The request $\tau_{0,2}^c$ thus appears twice in the figure, once before the preemption (aborted) and once thereafter (reissued). Note that the slack counter, due to the request's preemption, diminished from 16Δ to 13Δ.

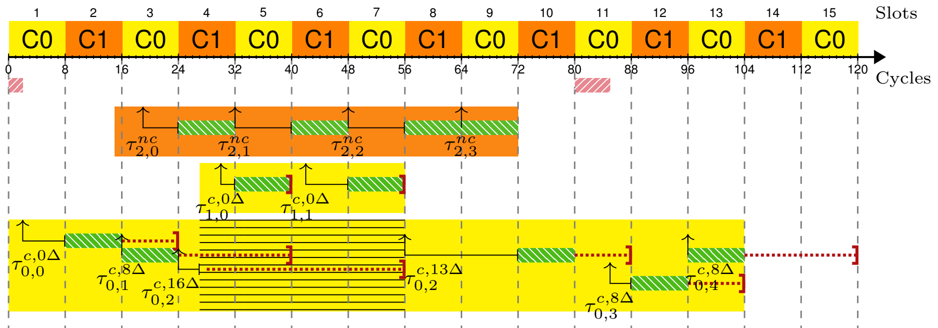


Figure 5 Memory blocking delays considering request preemption (`SHDp`).

This solution appears to be ideal in terms of preemption costs. However, preempting ongoing memory requests can be complex to implement. The processor pipeline has to be extended such that the memory access instruction and all instructions that started execution after it can be aborted. All these instructions have to be reexecuted and thus are not allowed to cause side-effects on the processor state. Such side-effects are, unfortunately, very common. Examples include branch prediction and instruction cache accesses, which occur early in processor pipelines and whose side-effects cannot (easily) be reverted. These effects consequently need to be taken into consideration through dedicated timing analyses. Similarly extensions are required on the memory hierarchy, including caches (aborting cache updates), the memory bus (cache coherence), and the memory arbiter itself. It thus appears preferable to explore an alternative approach that strikes a compromise in terms of implementation and analysis complexity.

5.4 Scheduling with Criticality Inheritance (`SHDi`)

The aim of this approach is to provide a means to control the impact of the slack counters and the interference from non-critical tasks on the memory blocking delay of preempting tasks. The dynamic `TDM`-based arbiters considered here are all based on the notion of *deadlines*. The idea of criticality inheritance is to *impose* a new deadline on a pending request at the moment when a critical task is released, regardless of the criticality of the preempted task. The preemption is still delayed – as before under the `SHDw` scheme. However, the blocking time is bounded by the newly imposed deadline.

This new deadline is computed in the same way as ordinary request deadlines. The only difference is that the issue date is replaced by the release date of the preempting task and that the current value of the slack counter, belonging to the preempted task, is always considered to be 0 (`TDMds`) or *Sl* (`TDMer`). This yields a deadline that certainly falls within the current or the next `TDM` period, and thus effectively bounds the memory blocking delay. An important aspect is that the slack counter for `TDMer` needs to be *Sl* for this computation, and not 0. This is required for the simple reason that, without slack, the deadline could fall on the immediate next `TDM` slot. Under `TDMer` this could cause a clash with an ongoing request from another core that was granted access by the arbiter based on the – then valid – slack counter value of the preempted task. Setting the slack counter to *Sl* thus ensures that any ongoing request can finish before the request from the preempted task is handled.

At runtime two scenarios can be distinguished, depending on the criticality of the preempted task. Firstly, if another critical task is preempted it is clear that its pending critical request already carries a deadline. Replacing this deadline is easy, it suffices to signal

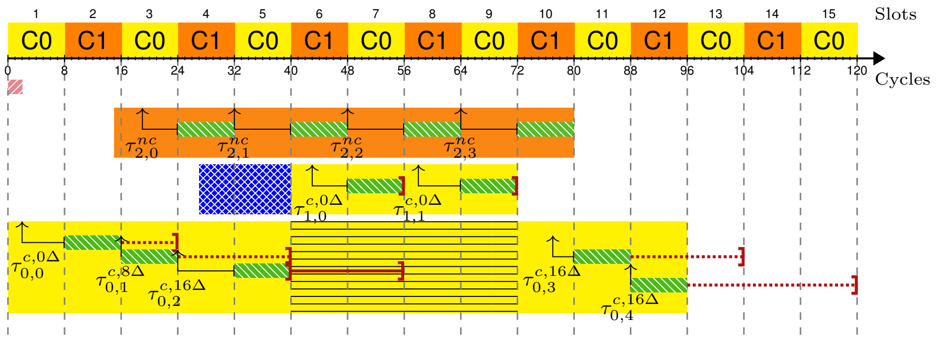


Figure 6 Memory blocking delays considering criticality inheritance (`SHDi`).

to the memory arbiter that the recomputation of the deadline is needed using the current cycle, i.e., the release of the preempting task. Non-critical requests do not carry a deadline and may be held in a structure separated from critical requests (e.g., a FIFO for `TDMer`). The request thus needs to be taken out of this structure and reissued as a critical request with the appropriate deadline. Finally, the core has to reclaim its `TDM` slot, which up to now has been marked non-critical, i.e. *`NC`*. These operations only concern the arbiter and do not impact the processor pipeline or other parts of the memory hierarchy.

Assuming that both cases require a constant amount of clock cycles t_{id} to associate the ongoing request with the newly imposed deadline, we obtain the following bound for the memory blocking delay under `SHDi` for `TDMds` and `TDMer` respectively:

$$\mathrm{MB}_i^{\texttt{SHDids}} = P + Sl - 1 + t_{id} \tag{5}$$

$$\mathrm{MB}_i^{\texttt{SHDier}} = P + 2 \cdot Sl - 1 + t_{id} \tag{6}$$

Note that the second Sl in Equation 6 stems from the non-zero slack counter, which ensures that all requests can access memory without clashes. We currently do not apply criticality inheritance when the preempting task is non-critical.[2] This falls in the domain of defining a (sensible) task model, which is not the subject of this work. We plan to address this in future work, along with the handling of timing failure events, in order to fully implement mixed-criticality systems. Non-critical tasks currently implicitly follow the `SHDw` strategy.

Figure 6 again shows an execution of the same task set as in the previous examples, this time under `TDMds` combined with `SHDi`. Critical task τ_1^c is released while critical request $\tau_{0,2}^c$ is pending. The pending request has its original deadline at the end of `TDM` slot 7 ($\dashv$). As before, it is subject to interference from task τ_2^{nc} on the other core, which otherwise would be prioritized due to the far deadline of $\tau_{0,2}^c$. However, the preemption imposes a new deadline at the end of the immediate next `TDM` slot (slot 5). The arbiter thus has to prevent the interference from the other core and has to assign the next slot to the preempted task – effectively bounding the memory blocking delay.

This approach combines a reasonable memory blocking delay bound with moderate implementation complexity and simple analysis. The hardware modifications only concern the memory arbiter that reacts to the core's timer, providing the release date and the criticality of the preempting task.

5.5 Misalignment Delays

Strict `TDM` as well as the dynamic `TDM`-based schemes all suffer from misalignment delays, highlighted in Section 4, for the same reasons. The delay appears when the task's misalignment *at the first* memory access after a preemption is larger w.r.t. the task's own `TDM` slot as determined by the WCET analysis [14, 19]. In the worst case the associated memory request misses the task's `TDM` slot by a single cycle, i.e., the issue date (`TDM`) or delayed issue date (dynamic `TDM`) miss the slot by a cycle. The request consequently completes in the worst case in the next `TDM` period, resulting in the following bound for all the `TDM`-based approaches:

$$\mathrm{MA}^{\texttt{TDM}} = \mathrm{MA}^{\texttt{TDMds}} = \mathrm{MA}^{\texttt{TDMer}} = P \tag{7}$$

The delay can be larger for non-critical tasks ($P^{nc} = k \cdot P$). Also note that the bound is smaller than the worst-case memory access latency of strict `TDM` ($P + Sl - 1$), since the memory wait time of to the first `TDM` slot is accounted for in the WCET (highlighted in beige 🟧 in Subfigure 2b).

[2] Deadlines could easily be imposed for preemptions by non-critical tasks.

5.6 Response-Time Analysis

The memory blocking (MB) and misalignment delay (MA) bounds, as described above for the preemption mechanisms `SHDw`, `SHDp`, and `SHDi` as well as the arbitration policies `TDMds` and `TDMer`, can now be integrated into the recurrence equations of the response-time analysis.

With regard to a task τ_i, the misalignment delay may appear every time any task τ_j $(i < j)$ resumes after a preemption. This is independent of whether τ_j directly preempts τ_i or some other task. The misalignment delay bound MA of the respective arbitration scheme thus needs to be added for every potential preemption that might occur.

Every preempting critical or non-critical task τ_i $(i > 0)$ may experience the memory blocking delay *before* starting to execute. The bound thus can be seen as part of the task's WCET, i.e., the ongoing memory request of the preempted task is essentially considered to be executed by the preempting task. We thus add MB to those parts of the equation that represent a WCET (C_i, C_j):

$$R_i^{n+1} = (C_i + \mathrm{MB}_i) + \sum_{\forall j \in \mathrm{hp}(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil ((C_j + \mathrm{MB}_j) + \mathrm{MA}) \tag{8}$$

For the `SHDw` and `SHDp` schemes, independently of the `TDM` policy, but also for `SHDi` when combined with `TDMds`, the term MB_j can thus be safely removed from this equation. The latency of the ongoing memory request, owned by the preempted task, is indeed counted twice: once for the preempting task (MB_j) and the preempted task (C_i). This is illustrated by Figure 6, were the blue area (▩), representing the MB_j term for task τ_1^c, ends at the deadline of request $\tau_{0,2}^c$ and thus the C_i of τ_0^c. For `TDMer` under `SHDi`, the imposed deadline may be greater than the original deadline of the ongoing request, since the imposed deadline is recomputed with a slack counter of Sl. The term MB_j is thus pessimistic and a possible refinement is to subtract the minimal memory latency l from this term.

6 Experiments

We use simulation to evaluate the runtime behavior of the various preemption mechanisms. Before discussing the results, we provide details on the experimental setup.

6.1 Experimental Setup

We developed a simulation engine that is able to simulate a dynamic execution trace of an entire multi-core platform. Traces are specified according to the system/hardware model from Sections 2 and 5. The engine includes a fixed-priority preemptive task scheduler, and various memory arbitration schemes (`TDMfs`, `TDMds`, `TDMer`), which can be combined with the aforementioned preemption mechanisms (`SHDw`, `SHDp`, `SHDi`). The `TDMfs` arbiter represents a variant of strict `TDM`, where non-critical tasks may only reclaim unused slots, and serves as a baseline for our measurements.

The engine can be configured in terms of the number of (non-)critical cores, `TDM` slots in a period (at least 1 per critical core), and (non-)critical tasks. The task scheduler respects user-defined core affinities that can be assigned freely to tasks. However, for our experiments each task is assigned to a single fixed core only (partitioned scheduling). Tasks are represented by a sequence of jobs, which, in turn, represent dynamic execution traces consisting of memory accesses that are separated by a fixed amount of computation time (cf. the various examples). This allows to simulate the same execution trace of a task set using different platform configurations and compare the results. Note that the framework does not model the actual computations, only the *time* needed for computations.

6.1.1 Task Set Generation

The *UUniFast* algorithm [6] allows to randomly generate a task set based on two parameters n and U, where n specifies the total number of tasks and U the total utilization desired. As we assume partitioned scheduling, we apply the algorithm for each core of the multi-core system separately and combine the resulting tasks, with the corresponding affinities, into a final task set Γ.

For each core, UUniFast first generates n different utilization values $\{u_0, u_1, \ldots, u_{n-1}\}$, one for each task τ_i. The sum of these utilization numbers yields the core utilization $U \leq 1$. From the utilization parameters, the task periods T_i are generated. Note that we constrain our system to harmonic periods, which ensure that hyper-periods and simulation times remain reasonable. The period of the first task T_0 is 20ms, while all other periods are random multiples of T_0, obtained from a uniform random distribution in the range $[1, 5]$. The individual task periods are hence in the range from 20ms to 100ms. From the task periods and the utilization numbers as well as the task set's hyper-period h, we then derive the worst-case execution time of each task $C_i = T_i \cdot u_i$, the implicit deadline $D_i = T_i$, and the number of jobs for each task $J_i = h/T_i$. Finally, task priority and criticality (on critical cores) are randomly assigned when the final task set Γ is constructed.

6.1.2 Traffic Generator

The simulation framework then requires a memory trace for each job of the generated task set, i.e., the (time) distances between consecutive memory accesses (see Section 2). We use the same traffic generator as in our previous work [13]. We thus only provide a brief overview here and invite interested readers to consult the original paper for additional details.

The goal of the traffic generator is to provide synthetic memory access patterns representing *cache misses* of some random dynamic program execution. The generator is thus calibrated against *actual dynamic execution traces* collected using the MiBench benchmark suite [8] on the Patmos processor architecture [20]. Note, however, that the generated memory accesses have to be consistent with the task's WCET (C_i). The generator thus tracks the evolution of a WCET bound as it proceeds. For each newly generated memory access this WCET bound is incremented by the worst-case memory access latency, which is bounded by $P - 1 + Sl$ cycles. Note that the same bound is applied for critical and non-critical tasks, even if no `TDM` slot is allocated to a core executing only non-critical tasks. The generator simply stops once the bound reaches the task's C_i. The resulting execution times thus rather closely approach the tasks' WCETs. The memory load is higher than can be expected in the average case. Note that we capture this effect in our experiments by varying the total system load.

6.1.3 `TDM` Schedule

The duration of a `TDM` slot length Sl, corresponds to an upper bound of the memory access latency previously determined on a Terasic DE-10 Nano evaluation board that is equipped with an Intel Cyclone V SoC-FPGA and 1 GB of DDR3 memory. A single Patmos processor was implemented in the FPGA and performed memory accesses in isolation via the SoC's multi-port memory controller running at 100 Mhz (the remaining components of the SoC were deactivated). At any moment a single memory access was in-flight during these measurements. Depending on the internal state of the memory controller and DDR memory (refresh, open page, etc.), we measured a memory latency between 21 and 40 cycles. In the simulation runs we thus consider a `TDM` slot length of 40 cycles. For `TDMer`, which is able to exploit memory requests completing faster than the `TDM` slot length, we simulate a varying latencies using a uniform random distribution in the range $[21, 40]$ clock cycles.

6.1.4 Simulation Setup

Using the task set and memory traffic generators, we perform simulations with varying numbers of cores (2, 4, 8, 16), critical cores (power of 2 in the range from 1 to the number of cores), and the normalized system utilization (steps of 10 from 10% to 100%, normalized to the number of cores). The number of tasks is randomly chosen between the number of cores and 32 tasks, while the number of critical tasks is randomly chosen between 1 and the number of tasks assigned to a critical core.

The `TDM` schedule assigns a single slot to each critical core, where each slot takes $Sl = 40$ cycles. The period P hence ranges from 40 to 640 cycles. The slack counters are reset at every job start to 0 or 40 cycles for `TDMds` and `TDMer` respectively. For each of these system configurations, 10 simulation runs were performed using 10 different task sets, which results in 12 600 runs.

6.2 Results for Preemption Schemes

We start by analyzing the memory blocking delays on the simulated task sets with the three preemption mechanisms considered. Subfigure 7a shows the cumulative average memory blocking delay over an entire simulation run for the `SHDw` preemption mechanism considering the `TDMfs`, `TDMds`, and `TDMer` arbitration policies.[3] As expected, the cumulative overhead can become very large going up to $1.6 \cdot 10^6$ cycles for both `TDMfs` and `TDMds` (showing virtually identical results), while only reaching $3 \cdot 10^3$ cycles for `TDMer`– despite the fact that preemptions are rare events.[4] We observed a task set executing on 16 cores experiencing **502**ms (an average of **31**ms per core) of blocking delays within **590**ms of total execution time (assuming a clock speed of 100 Mhz). The other approaches, `SHDp` and `SHDi`, have significantly lower overhead compared to `SHDw` (not shown due to space considerations).

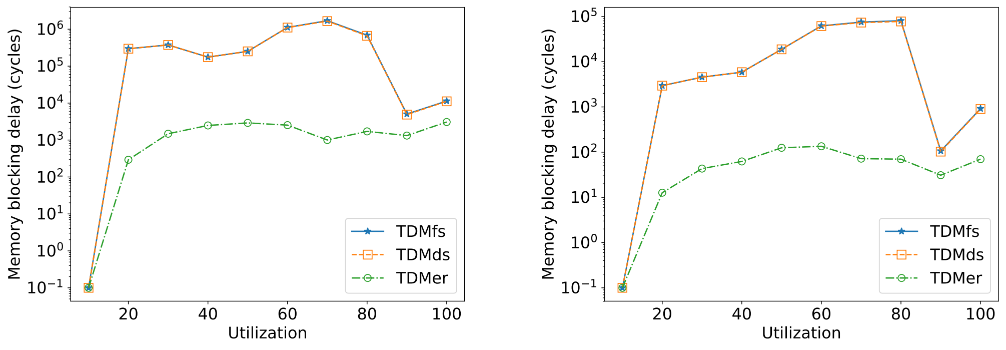
Since non-critical tasks execute in a best-effort manner, their memory blocking delays are expected to potentially become quite large, in particular, when preempting another non-critical task. Subfigure 7b thus highlights the average maximum memory blocking delay experienced by non-critical tasks across utilization levels for `SHDw` in combination with either `TDMfs`, `TDMds`, or `TDMer`. As can be seen, individual preemptions consistently take thousands of cycles, which corresponds to about **18**ms (maximum observed memory blocking delay). As these events are still rare, some volatility in the simulations is visible through the large drop at 90% utilization. Note that typically non-critical tasks represent the majority of the computation and memory load of the generated task sets. Consequently, non-critical tasks experience most of the preemptions as well as the associated memory blocking delays.

Finally, Subfigure 8a shows the maximum memory blocking delays experienced by critical tasks considering all three preemption mechanisms in combination with `TDMfs`, `TDMds`, and `TDMer`. The delays are normalized w.r.t. the `TDM` period in order to avoid penalizing simulation configurations with shorter periods. These delays are representative of the upper bounds defined in Subsection 5. Note that the results for `TDMfs` are virtually identical to those for `TDMds` and are thus not shown.

As expected, the `SHDp` scheme presents very low memory blocking delays, as can be seen in more detail in Subfigure 8b. Under `TDMds` this preemption mechanism leads to no noticeable memory blocking. This is explained due to the non-work-conserving nature of this arbitration technique, which leads to long memory wait times and consequently increases

[3] Each point represents an average value over all schedulable task sets at the corresponding normalized utilization.

[4] On average, there are 7 preemptions per run, with a maximum number of 288 preemptions in rare cases.



(a) Average cumulative memory blocking delay over all tasks.

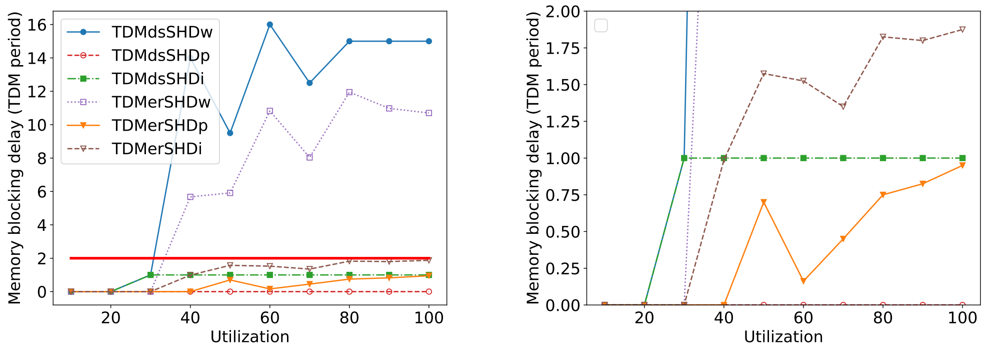
(b) Average maximum memory blocking delay for non-critical tasks.

Figure 7 Memory blocking delays across normalized system utilization for SHDw.

the probability of preempting a pending request. The situation is different for TDMer. Due to its high efficiency, the probability of preempting a pending request is much lower. It is instead more likely to preempt a request that is currently processed by the memory, resulting in moderate memory blocking delays for SHDp. Note that these delays may never exceed a single TDM period, notably for configurations with a single critical core. The highest memory blocking delays are observed for SHDw under TDMds. The memory delay amounts to up to 16 TDM periods for a configuration with 2 cores, where both cores are critical (i.e., $P = 80$ cycles). The TDMer arbiter fares slightly better, with a maximum of about 12 TDM periods. This demonstrates the high overhead experienced even by critical tasks when using SHDw.

The SHDi scheme, as expected, falls in-between the two other schemes. In combination with TDMds the memory blocking delay is at most one TDM period. However, starting from 50% utilization, we can notice that TDMer exhibits slightly worse results compared to TDMds. Nevertheless, the preemption cost for TDMer always stays below 2 TDM periods (highlighted by Subfigure 8b) – notably for configurations with a single critical core (cf. Equation 6).

These result confirm the intuitive expectation that the overhead induced by the three preemption schemes is strictly increasing from SHDp over SHDi to SHDw. However, this is not always the case due to the extra costs induced by the slack counters under TDMer (a counter-example was encountered for a larger slot length of $Sl = 100$, while evaluating the



(a) Maximum memory blocking delay w.r.t. the arbitration and preemption schemes.

(b) Zoom on the plot from the left, focusing on blocking delay up to 2 TDM periods.

Figure 8 Maximum memory blocking delay for critical tasks across normalized system utilization.

impact of varying the memory access latency[5]). Recall that the newly imposed deadline during a preemption is computed considering a minimum slack of a single `TDM` slot length (Sl) in order to avoid clashes on the immediate next `TDM` slot (see Subsection 5.4). Accounting for this additional runtime overhead $\varepsilon \leq Sl$, we can derive a relationship between the various preemption schemes: $\mathrm{MB}_i^{\mathtt{SHDp}} \leq \mathrm{MB}_i^{\mathtt{SHDi}} \leq \mathrm{MB}_i^{\mathtt{SHDw}} + \varepsilon$.

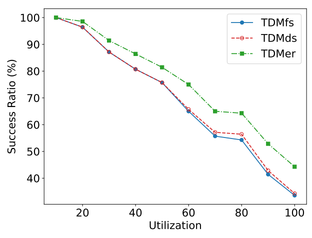
To conclude, the arbitration-induced preemption costs for critical tasks can be very high at runtime when the `SHDw` scheme is used, while it is similar for the `SHDi` and the `SHDp` schemes. This means that other criteria, such as predictability and implementation complexity, can be used to choose among the schemes. `SHDi` appears to strike a reasonable balance between these two criteria. Besides, the memory blocking delays are lower using `TDMer` than when using `TDMfs` or `TDMds`.

6.3 Results for (Preemptive) Arbitration Schemes

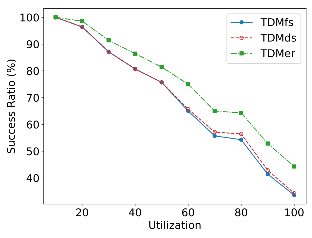
We now evaluate the success rate of our preemptive arbitration policies. The success ratio refers to the number of task sets[6] that were *schedulable* for each level of utilization, i.e., simulation ended its execution without any deadline miss for critical tasks. Subfigures 9a, 9b, and 9c depict this success rate for our 3 arbitration policies under `SHDw`, `SHDp`, and `SHDi` respectively. Results are shown considering the same task set for all combinations. Comparing the impact of the preemption schemes across all utilization levels, little difference among them is visible. This can be explained by the number of preemptions, which is small compared to the total execution time of each simulation. As seen in Section 6.2, individual preemptions may however cause considerable overhead.

Overall, there was no significant difference in the success ratio of `TDMfs` and `TDMds`, except for a normalized utilization above 70%. In that case, we can notice better results for `TDMds`, regardless of the preemption technique. This is different from the results obtained in our previous work [12, 13], which shows an improved memory utilization for `TDMds` compared to `TDMfs`. In this previous work, the observed improvement reached up to a factor of 3.3 [13] in terms of memory utilization at low total system utilization, which then leveled off considerably at higher load. It appears that, `TDMds` improves the memory utilization mostly for situations where system load is not jeopardizing *schedulability*, explaining the small impact on success rates. However, regardless of the preemption techniques, `TDMer` shows better results at almost all utilization levels. This can be explained by the decoupling from `TDM` slots and

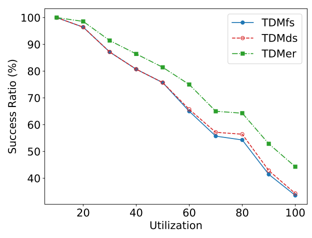
[5] Results not shown due to space constraints.
[6] Each point represents an average value over 124 runs.



(a) `SHDw`.

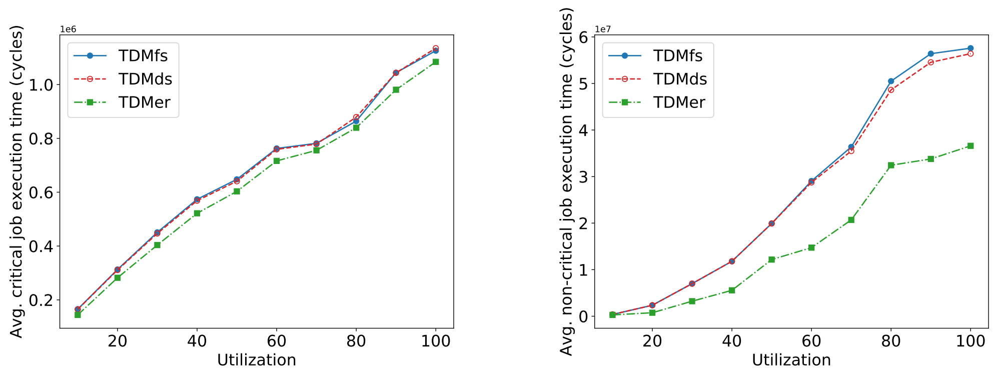


(b) `SHDp`.



(c) `SHDi`.

Figure 9 Average *schedulability* success ratio through normalized system utilization.



(a) Critical jobs execution time. **(b)** Non-critical jobs execution time.

Figure 10 Average jobs execution times with varying normalized system utilization using `SHDi`.

the fine-grained dynamic memory arbitration. This effectively renders the `TDMer` scheme work-conserving. These results confirm those previously obtained considering a restricted task model (one task per core), where the observed memory utilization improvements for `TDMer` were considerable, even at high utilization levels [13]. `TDMer` thus has a relevant impact on the success rates.

As the most reasonable choice for a preemption scheme is `SHDi`, Subfigures 10a and 10b depict the average execution time of critical and non-critical tasks respectively using this scheme. The relative gain by `TDMer` is particularly visible for the average execution time of non-critical tasks at high utilization levels. For instance, at 100% system utilization `TDMer` achieves an improvement of 36%. Clearly concluding that `TDMer` improves the execution times of critical and non-critical tasks, and thus can reduce the probability of missing the deadline of critical tasks. Additionally, as preemptions are rare events, the runtime impact of the different preemption schemes is small compared to the impact of the arbitration policy.

7 Related work

`TDM` strategies have been used at different design levels of real-time systems. Tabish et al. [21] split tasks into execution phases with and without main memory activities, which are scheduled in order to avoid contention by applying `TDM`. `TDM` is applied at the task scheduling level, with slot sizes accommodating the longest phase. Phases without main memory activity may only access data stored in a scratchpad. Preemption-related delays due to memory accesses thus cannot appear.

The *slot* shifting approach by Fohler [7], applies a similar idea as our dynamic arbitration schemes, but to task scheduling. The granularity of slots allocated to tasks is consequently much larger than individual memory requests considered here. Memory blocking delays may be present, depending on the underlying platform, *even* when only a single core is considered, but not taken into consideration.

Altmeyer et al. [1] worked on a compositional framework for multi-core response time analysis. Among other things, they explored the impact of different arbitration schemes (Fixed-Priority, FIFO, Round-Robin and `TDM`) and defined upper-bounds for the arbitration induced preemptions delays. They assume that all requests sent to the bus are non-preemptive, which is similar to our `SHDw` approach. Their results only apply to strict `TDM` and not to our dynamic `TDM`-based approaches. However, their bounds for Fixed-Priority arbitration seem to be incorrect. Suppose a higher-priority task τ_i preempts another tasks τ_j $(j < i)$. The bus interference experienced by an ongoing request of τ_j may then block τ_i, i.e., the memory blocking delay. The proposed equation [1, Eq. 12], however, misses interference from requests issued by a higher priority task τ_k running on another core, where $j < k < i$. This cannot be resolved easily using the framework of Altmeyer et al. – except if a scheme similar to `SHDi` is applied. In that case it suffices to consider the ongoing request of the preempted task in the context of the preempting task – via the term $S_i^x(t)$.

Yonghui et al. *skip* unused slots in a `TDM` period, supporting variable-sized `TDM` slots [17]. Hassan et al. [11, 10] similarly propose a work-conserving variant of `TDM` along with a technique to generate harmonic `TDM` schedules accommodating critical and non-critical tasks. The approaches do not preserve the relative alignment of the program execution with the `TDM` schedule, offset analyses [14, 19] thus cannot be applied. Preemption mechanism are not explicitly discussed, it appears that a strategy similar to `SHDp` is envisioned by the authors. Memory blocking delays exist and have not been analyzed previously. The bounds for strict `TDM` appear to be applicable – except for preemptions of non-critical tasks in a harmonic `TDM` schedule. Finally, it is unclear whether the `TDM` schedule can be changed in response to a task preemption.

Paolieri et al. [18] propose a multi-core platform for mixed-criticality systems that is equipped with a hierarchical Round-Robin arbiter, which always to prioritizes critical tasks. In contrast to the approaches presented here, their arbiter is not able to exploit task criticalities to improve memory utilization or to reduce the average execution time of non-critical tasks. The authors envision a preemption scheme similar to `SHDp`, but do not analyze the preemption costs. The ideas underlying Altmeyer's analysis for Round-Robin and Fixed-Priority [1] could be used to model the system.

Kostrzewa et al. [15] propose a mechanism, which provides latency guarantees for hard real-time transmissions in a network-on-chip. For each critical task the amount of slack time is computed off-line and programmed into a counter at the level of NoC interface at every job start. Best-effort transmissions, similar to the dynamic arbiters considered here, may then delay critical transmissions as long as slack is available. Task preemptions are then considered for hard real-time and best-effort transmissions. Critical tasks may run out of slack in the middle of a best-effort transmission which are thus preempted. A delay – similar to the memory blocking delay – is taken into consideration to avoid exhausting the slack budget before all best-effort transmissions are successfully suspended.

8 Conclusion

The work presented in this paper extends previous work on dynamic `TDM`-based memory arbitration schemes by adding support for a preemptive execution model and by identifying the limitations of such a system. We identify two sources of arbitration-induced preemption delays, the *memory blocking delay* and the *misalignment delay*, and propose means to manage, and finally bound these delays. While bounding the misalignment delay is straightforward, limiting the memory blocking delay is more involved. We thus propose formal bounds for two obvious preemption schemes based on waiting and preemptable memory requests (`SHDw` and `SHDp`). Additionally, we propose an alternative scheme (`SHDi`), which imposes new deadlines for critical requests and leverages *criticality inheritance* when a critical task is blocked by a non-critical request. The experimental results showed that the preemption mechanisms show little difference at runtime, which allows us to select the best approach according to other criteria, such as low implementation complexity and analyzability. The new `SHDi` approach appears to offer the best trade-off in terms these criteria.

References

1 Sebastian Altmeyer, Robert I. Davis, Leandro Indrusiak, Claire Maiza, Vincent Nelis, and Jan Reineke. A Generic and Compositional Framework for Multicore Response Time Analysis. In *Proceedings of the International Conference on Real Time and Networks Systems*, RTNS '15, pages 129–138. ACM, 2015. `doi:10.1145/2834848.2834862`.

2 N. Audsley, A. Burns, M. Richardson, K. Tindell, and A.J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292, 1993.

3 S. Baruah and N. Fisher. The partitioned multiprocessor scheduling of sporadic task systems. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 9 pp.–329, December 2005. `doi:10.1109/RTSS.2005.40`.

4 Sanjoy Baruah and N. Fisher. The partitioned multiprocessor scheduling of deadline-constrained sporadic task systems. *IEEE Transactions on Computers*, 55(7):918–923, July 2006. `doi:10.1109/TC.2006.113`.

5 Alan Burns and Robert I. Davis. A Survey of Research into Mixed Criticality Systems. *ACM Comput. Surv.*, 50(6):82:1–82:37, November 2017. `doi:10.1145/3131347`.

6 P. Emberson, R. Stafford, and R.I. Davis. Techniques For The Synthesis Of Multiprocessor Tasksets. In *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pages 6–11, 2010.

7 G. Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Proc. Real-Time Systems Symposium (RTSS)*, pages 152–161, Pisa, Italy, December 1995.

8 M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of the Int. Workshop on Workload Characterization*, pages 3–14, 2001.

9 S. Hahn, J. Reineke, and R. Wilhelm. Towards Compositionality in Execution Time Analysis: Definition and Challenges. *SIGBED Rev.*, 12(1):28–36, 2015.

10 M. Hassan and H. Patel. Criticality- and Requirement-Aware Bus Arbitration for Multi-Core Mixed Criticality Systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11, April 2016. `doi:10.1109/RTAS.2016.7461327`.

11 M. Hassan, H. Patel, and R. Pellizzoni. A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 307–316, April 2015. `doi:10.1109/RTAS.2015.7108454`.

12 F. Hebbache, M. Jan, F. Brandner, and L. Pautet. Dynamic Arbitration of Memory Requests with TDM-like Guarantees. In *International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'17)*, December 2017.

13 F. Hebbache, M. Jan, F. Brandner, and L. Pautet. Shedding the Shackles of Time-Division Multiplexing. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 456–468, December 2018. `doi:10.1109/RTSS.2018.00059`.

14 T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. Static Analysis of Multi-core TDMA Resource Arbitration Delays. *Real-Time Syst.*, 50(2):185–229, March 2014.

15 A. Kostrzewa, S. Saidi, and R. Ernst. Slack-based Resource Arbitration for Real-time Networks-on-chip. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '16, pages 1012–1017. EDA, 2016.

16 H. Leontyev and J. H. Anderson. Generalized Tardiness Bounds for Global Multiprocessor Scheduling. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 413–422, December 2007. doi:10.1109/RTSS.2007.33.

17 Y. Li, B. Akesson, and K. Goossens. Architecture and Analysis of a Dynamically-scheduled Real-time Memory Controller. *Real-Time Syst.*, 52(5):675–729, September 2016.

18 Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware Support for WCET Analysis of Hard Real-time Multicore Systems. In *Proceedings of the International Symposium on Computer Architecture*, ISCA '09, pages 57–68. ACM, 2009. doi:10.1145/1555754.1555764.

19 Hamza Rihani, Matthieu Moy, Claire Maiza, and Sebastian Altmeyer. WCET analysis in shared resources real-time systems with TDMA buses. In *Proceedings of the International Conference on Real Time and Networks Systems*, RTNS '15, pages 183–192. ACM, 2015. doi:10.1145/2834848.2834871.

20 M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. Thorn. Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, OASICS, pages 11–21, 2011.

21 R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo. A Real-Time Scratchpad-Centric OS for Multi-Core Embedded Systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11, April 2016.

Fast and Effective Multiframe-Task Parameter Assignment Via Concave Approximations of Demand

Bo Peng
Department of Computer Science, Wayne State University, Detroit, MI, USA
et7889@wayne.edu

Nathan Fisher
Department of Computer Science, Wayne State University, Detroit, MI, USA
fishern@wayne.edu

Thidapat Chantem
Department of Electrical and Computer Engineering, Virginia Tech, Arlington, VA, USA
tchantem@vt.edu

Abstract

Task parameters in traditional models, e.g., the generalized multiframe (GMF) model, are fixed after task specification time. When tasks whose parameters can be assigned within a range, such as the frame parameters in self-suspending tasks and end-to-end tasks, the optimal offline assignment towards schedulability of such parameters becomes important. The GMF-PA (GMF with parameter adaptation) model proposed in recent work allows frame parameters to be flexibly chosen (offline) in arbitrary-deadline systems. Based on the GMF-PA model, a mixed-integer linear programming (MILP)-based schedulability test was previously given under EDF scheduling for a given assignment of frame parameters in uniprocessor systems. Due to the NP-hardness of the MILP, we present a pseudo-polynomial linear programming (LP)-based heuristic algorithm guided by a concave approximation algorithm to achieve a feasible parameter assignment at a fraction of the time overhead of the MILP-based approach. The concave programming approximation algorithm closely approximates the MILP algorithm, and we prove its speed-up factor is $(1+\delta)^2$ where $\delta > 0$ can be arbitrarily small, with respect to the exact schedulability test of GMF-PA tasks under EDF. Extensive experiments involving self-suspending tasks (an application of the GMF-PA model) reveal that the schedulability ratio is significantly improved compared to other previously proposed polynomial-time approaches in medium and moderately highly loaded systems.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases generalized multiframe task model (GMF), generalized multiframe task model with parameter adaptation (GMF-PA), self-suspending tasks, uniprocessor scheduling, mixed-integer linear programming, concave approximation, linear programming

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.20

Acknowledgements We are grateful to the anonymous reviewers whose comments helped to significantly improve our paper. This research has been supported in part by the US National Science Foundation (Grant Nos. CNS-1618185, IIS-1724227, and CSR-1618979).

1 Introduction

A generalized multiframe (GMF) task, whose model [3] generalizes the multiframe task model [16] (MF) and the sporadic task model, consists of a number of ordered frames where each frame has its own execution time, relative deadline, and frame separation time (the minimum interval between two frames' release times). The GMF model generalizes the sporadic model by using a set of ordered frames to represent an instance of a sporadic task.

Instead of setting an identical implicit frame deadline and minimal separation time for each frame as in the MF model, the GMF model assigns each frame an individual deadline and a minimum frame separation time.

The multiframe models (GMF/MF) have many applications. For example, Andersson [1] presented the schedulability analysis of flows in multi-hop networks comprising software-implemented Ethernet switches according to the GMF model. Ding et al. [11] scheduled a set of tasks with an I/O blocking property under the MF model. The self-suspension tasks [18] can be represented using the GMF model but the problem size can be very large, e.g., in automotive systems. In the keynote [7] of ECRTS 2012, Buttle has shown many scheduling challenges as the number of ECUs in vehicles increases rapidly each year; there are more than 100 ECUs nowadays and each task can easily have 50-300 functions. In such complex systems, there are several self-suspension tasks (each consisting of multiple functions) and their end-to-end latencies need to be maintained in distributed settings.

The GMF model increases flexibility compared to the sporadic and MF task models, and all parameters in the GMF model are typically not mutable after task specification time. However, frame parameters can be adjustable (under the constraints of task parameters) to improve schedulability in applications such as the self-suspension tasks [20] and end-to-end flows [19]. Frame parameters are mainly used to maintain execution order in such applications (e.g., frame priorities in FP scheduling and frame deadlines in EDF scheduling [22]). In order to optimally assign parameters to improve schedulability, Peng and Fisher [18] extended the GMF model and presented the GMF with parameter adaptation model (GMF-PA). In the GMF-PA model, frame deadlines and separations can be selected under a set of constraints. In this flexible model, frame parameters are optimally assigned (towards schedulability) offline for each frame under the MILP algorithms [18].

Although the GMF-PA model is more flexible, it has been shown that both the feasibility and the parameter selection problems are very hard to solve. On the feasibility side, Ekberg and Yi [12] proved that the feasibility of sporadic task systems remains coNP-complete under bounded utilization. On the parameter selection side, the priority assignment of subtasks in end-to-end task systems (originally the classical job-shop scheduling algorithm) has been shown to be NP-hard [13]. The scheduling of self-suspending tasks (even for self-suspending tasks with at most two frames) is NP-hard in the strong sense [21].

In order to address the feasibility test and parameter selection problem, Peng and Fisher [18] gave an exact schedulability test of GMF-PA tasks when frame parameters are integers. The test is based on mixed-integer linear programming (MILP) under EDF scheduling in uniprocessor systems. A sufficient, MILP-based schedulability test was also developed. Although this sufficient approximation algorithm [18] is quite efficient, it is still MILP-based and thus may require exponential-time to solve in general. The goal and contribution of this paper are to give an efficient linear programming-based algorithm that can determine the feasibility and select the frame parameters of GMF-PA tasks.

The MILP-based algorithm contains a set of integer variables which form a set of staircase functions/constraints (detailed in Section 5). To transform the MILP-based algorithm into a LP-based algorithm, our idea is to use a set of linear functions to approximate all staircase functions. As such, the selection of the slope values of the linear functions is directly related to the schedulability of a system; if the slope values are not properly set, the linear functions can grossly over-approximate, resulting in low schedulability ratio (the number of successfully scheduled systems over the total tested).

In order to get a close approximation, we first use a set of concave functions that very closely tracks the demand staircase functions to incur only a very small speed-up factor compared to the MILP algorithm. Since there exist no known efficient methods to solve concave programming problems, we use the concave functions to guide the slope assignment of linear functions in our iterative LP-based algorithm. That is, the LP algorithm runs multiple times during which the algorithm adjusts the slopes of the linear functions based on the concave functions. According to experiments, after a small number of iterations, the LP-based algorithm can approach (or reach) the local optimal[1]. We apply the LP-based algorithms to schedule self-suspending tasks under EDF scheduling in uniprocessor systems as a test case.

Our Contributions:

- We give a concave approximation algorithm based on the MILP algorithm and prove the speed-up factor of the algorithm is $(1 + \delta)^2$ with respect to the exact schedulability test of GMF-PA tasks under EDF scheduling on uniprocessors. The positive constant δ is a user-defined constant which can be made arbitrarily close to zero.
- Since there is no known tractable way to solve a concave programming problem, we develop a LP-based heuristic algorithm based on the concave approximation algorithm for GMF-PA tasks. The LP-based algorithm is an efficient schedulability test and can select frame parameters at the same time.
- We apply the LP-based algorithm to schedule multiple-suspending tasks. To exploit the unique property of one-suspending tasks, as opposed to multi-suspending tasks, we present an improved heuristic algorithm for GMF-PA tasks.
- We conduct extensive experiments and show that the LP-based algorithms with fixed numbers of iterations outperform previous work in terms of schedulability and average running time. The fixed numbers of iterations make the LP-based algorithms pseudo-polynomial (the input size depends on the maximum interval length [3]), which is more efficient than the MILP-based approach.

Section 2 surveys the related work. We review our GMF-PA model in Section 3, and we formally state the goal of this paper in Section 4. Section 5 reviews our parameter-adaptation method using mixed-integer linear programming (MILP) to obtain a schedulability test under EDF scheduling. The concave approximation algorithm based on the MILP algorithm is presented in Section 6. Since concave programming algorithm does not scale well, two iterative LP-based algorithms are presented in Section 7. After applying the LP-based algorithms to self-suspending tasks, Section 8 provides extensive experimental results compared to state-of-the-art results. At last, Section 9 concludes this work and proposes future work.

2 Related Work

In this section, we introduce the related work on the GMF-PA model in Section 2.1, and survey one of its applications, the self-suspending tasks, in Section 2.2.

2.1 The Generalized Multiframe Model

The generalized multiframe model (GMF) was presented by Baruah et al.[3] to extend the sporadic task model and multiframe task model (MF) [16]. The recurring real-time task (RRT) model [2] generalizes the GMF model to handle conditional code. The digraph model [23] further generalizes the RRT model to allow arbitrary directed graphs (with loops),

[1] The local optimal of the iterative LP-based algorithm is reached when all variables converge.

and it was shown that the feasibility problem on preemptive uni-processor systems remains tractable (pseudo-polynomial complexity with bounded system utilization). A complete review is surveyed by Stigge and Yi [24].

The GMF model has great advantages and has been applied to multiple areas, as described earlier. However, current related models typically assume that parameters are fixed during task specification time. In the GMF-PA model [18] which extends the GMF model, frame parameters are flexible and can be chosen by the MILP-based approach in uniprocessor systems. The dGMF-PA model [19] extends the GMF-PA model to represent end-to-end flows in distributed systems. Similar flexible models, such as the parameter-adaptation model [8] and elastic model [6], are also used in many applications.

2.2 The Self-Suspending Task Model

A typical self-suspension task model [15] contains two computational frames separated by a self-suspending frame. After the first computational frame finishes, the job suspends executing the other computational frame until an external operation completes. The order of the frames is required and a task suspends itself to communicate with external devices, I/O operations, computation offloading, etc. We call such tasks *one-suspension* self-suspending tasks.

For one-suspension self-suspending tasks, Ridouard et al. [21] proved that scheduling such periodic self-suspending tasks on a uniprocessor is NP-hard in the strong sense. Due to the hardness of such scheduling problems, Chen and Liu [9] gave a fixed-relative-deadline (FRD) scheduling algorithm to improve the schedulability of sporadic self-suspending tasks on uniprocessor systems. The FRD algorithm assigns frame relative deadlines and schedules the ordering of frames of tasks under EDF scheduling.

The multiple-segment suspending task model [14], which allows multiple suspending frames, explicitly considers the execution sequence of frames in a task. Peng and Fisher [18] utilize MILP to select frame parameters of multiple-segment self-suspending tasks. The MILP algorithm [18] is an optimal FRD algorithm which extends the work by Chen and Liu [9]. A recent review on scheduling self-suspending tasks (mostly on one-suspension tasks) can be found in the work by Chen et al. [10].

3 Model

We review the generalized multiframe (GMF) model [3] and the generalized multiframe model with parameter adaptation (GMF-PA) in this section.

A GMF task τ_i consists of a set of ordered frames and each frame ϕ_i^j has its own execution time E_i^j, relative deadline D_i^j, and frame separation time P_i^j. All frames of a task τ_i can be represented by the 3-vector tuple $(\overrightarrow{E_i}, \overrightarrow{D_i}, \overrightarrow{P_i})$ where $\overrightarrow{E_i}$=$[E_i^0, E_i^1,..., E_i^{N_i-1}]$, $\overrightarrow{D_i}$=$[D_i^0, D_i^1,..., D_i^{N_i-1}]$, and $\overrightarrow{P_i}$=$[P_i^0, P_i^1,..., P_i^{N_i-1}]$. The ℓ'th frame of task τ_i arrives at time a_i^ℓ, has deadline at $a_i^\ell + d_i^\ell$, and worst-case execution time e_i^ℓ. Since frames arrive in sequence, the ℓ'th frame corresponds to frame $\phi_i^{\ell \bmod N_i}$, and we have:

1. $a_i^{\ell+1} \geq a_i^\ell + P_i^{\ell \bmod N_i}$
2. $d_i^\ell = D_i^{\ell \bmod N_i}$
3. $e_i^\ell = E_i^{\ell \bmod N_i}$

Based on the GMF model, the GMF-PA model [18] is derived to allow frame parameters to be assigned instead of fixing them during task specification time. Let $\mathcal{T} = \{\tau_0, \tau_2,..., \tau_{n-1}\}$ be the task system of n GMF-PA tasks executing on one processor. The task τ_i=$[\phi_i^0, \phi_i^1, \phi_i^2,..., \phi_i^{N_i-1}]$ consists of N_i frames where ϕ_i^j=$(E_i^j, \underline{D}_i^j, \overline{D}_i^j, \underline{P}_i^j, \overline{P}_i^j)$. The j'th frame execution time of the i'th task is E_i^j, and the i'th task-wise execution time is $E_i = \sum_{j=0}^{N_i-1} E_i^j$. The lower bound of relative deadline D_i^j (respectively, the minimum inter-arrival time between consecutive frames, P_i^j) is $\underline{D}_i^j$ (respectively, $\underline{P}_i^j$) and the upper bound of D_i^j (respectively, P_i^j) is $\overline{D}_i^j$ (respectively, $\overline{P}_i^j$). The frame parameters D_i^j and P_i^j can be flexibly assigned in the ranges $[\underline{D}_i^j, \overline{D}_i^j]$ and $[\underline{P}_i^j, \overline{P}_i^j]$, respectively. The frame distance $D_i^{j,k} = D_i^k + \sum_{p=0}^{(k-j-1) \mod N_i} P_i^{(j+p) \mod N_i}$ represents the relative time between the release of the j'th frame and the deadline D_i^k of the k'th frame. For example, $D_i^{2,4} = P_i^2 + P_i^3 + D_i^4$. The task deadline $\mathcal{D}_i$ is the upper bound of $D_i^{N_i-1} + \sum_{j=0}^{N_i-2} P_i^j$, and the task minimum inter-arrival time $\mathcal{P}_i$ is the upper bound of $\sum_{j=0}^{N_i-1} P_i^j$. The utilization of task τ_i is $U_i = E_i/\mathcal{P}_i$, and the utilization of a task system is $U_{cap} = \sum_{i=0}^{n-1} U_i$.

Frame parameters (D_i^j and P_i^j) must satisfy the localized Monotonic Absolute Deadlines (l-MAD) property [3] to maintain frame execution order. That is, the absolute deadline of the j'th frame must be no later than the one of the $j+1$'th frame ($D_i^j \leq P_i^j + D_i^{(j+1) \mod N_i}$, $\forall i, j$). Figure 1 shows an example of the GMF model with the l-MAD property. The l-MAD property is widely used in systems which use first-in first-out (FIFO) scheduling for a shared resource. E.g., a network can be seen as a shared resource and packets sent from a computational node to a network node follow FIFO scheduling.

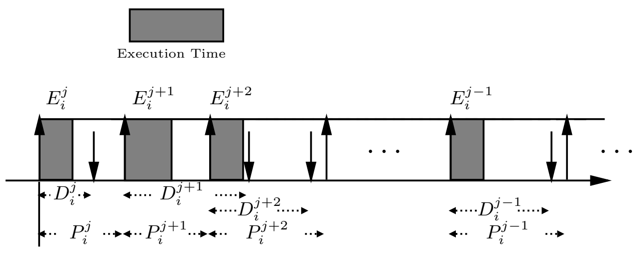


Figure 1 This figure contains all task τ_i's ordered frames from the j'th frame to the $(j-1)$ mod N_i'th frame (we omit "mod N_i" in this figure for simplicity). The starting frame can be any frame ϕ_i^j in an interval length t. Note that each frame deadline can be larger than frame separation time, e.g., $D_i^{j+1} \geq P_i^{j+1}$ in this figure. The details will be shown in Section 5.

4 Problem Statement

Let $\mathsf{dbf}_i(t, \vec{F}_i)$ be the *task* demand bound function of a GMF-PA task τ_i within the interval length t. Let $\vec{F}_i = [D_i^0, P_i^0, D_i^1, P_i^1, ..., D_i^{N_i-1}, P_i^{N_i-1}]$ represent an assignment of values for all the task parameters (frame deadline and separations) of task τ_i. The task demand bound function $\mathsf{dbf}_i(t, \vec{F}_i)$ accounts for task τ_i's accumulated execution time of frames which have both release times and deadlines inside the interval of length t. We use the notation $\mathsf{dbf}_i(t, D_i^{j,k})$ to represent the demand for the k'th frame when the first frame to arrive in the interval length t is the j'th frame. The relationship between the frame demand and task demand will be presented in Section 5. In a uniprocessor system $\mathcal{T}$, the sufficient and necessary condition for schedulability of a task set $\mathcal{T}$ is shown in Equation 1.

$$\sum_{\tau_i \in \mathcal{T}} \mathsf{dbf}_i(t, \vec{F}_i) \leq t, \quad \forall t. \tag{1}$$

▶ **Problem Definition.** *Given the above model, our goal is to find an optimal and valid assignment $\vec{F}_i$ of frame parameters of all tasks so that the worst-case demand $\sum_{\tau_i \in \mathcal{T}} \mathsf{dbf}_i(t, \vec{F}_i)$ over all time intervals of length t is minimized.*

5 The MILP Algorithm

We review the MILP algorithm [18] to solve the problem defined in Section 4 under EDF scheduling in uniprocessor systems since the proposed concave programming and LP-based algorithms are closely related to the MILP algorithm.

Figure 2 shows the MILP algorithm. Notations in bold font are constants and the other notations are variables. Lines 3 and 5 are the requirements that a feasible system must obey. Line 4 shows the l-MAD property. Line 6 shows the calculation of the demand for every possible sequence of frames of task τ_i over any interval of length t. To calculate all possible frame demands, we use the notation[2] $y_i^{j,k}(t)$ to denote the demand of the k'th frame of task τ_i starting from the j'th frame over a t-length interval. To calculate the worst-case demand under EDF scheduling, the starting j'th frame arrives exactly at the start of the interval and subsequent frames arrive as soon as possible (e.g., see [3] for GMF schedulability).

The inequality $\frac{t-t_b}{\mathcal{P}_i} \leq x_i^{j,k}(t) - \frac{realmin}{\mathcal{P}_i}$ is the constraint function that decides the value of $x_i^{j,k}(t)$ where $x_i^{j,k}(t)$ decides the value of $y_i^{j,k}(t)$ in turn. The length t_b is the summation of the previous periods $\lfloor \frac{t}{\mathcal{P}_i} \rfloor \cdot \mathcal{P}_i$ and the frame distance from the starting j'th frame to k'th frame $D_i^{j,k}$, and the constant $realmin$ is the smallest representable positive number for the MILP solver. For example, the length $t_b = D_i^{1,3} + \lfloor \frac{t}{\mathcal{P}_i} \rfloor \cdot \mathcal{P}_i$ if we consider the interval starting with an arrival of the first frame and ending at the deadline of the third frame. When $t \geq t_b$, the integer variable $x_i^{j,k}(t) \in [0, 1]$ must be one for the inequality in Line 6 to be feasible and the demand E_i^k contributes to $y_i^{j,k}(t)$. When $t < t_b$, $x_i^{j,k}(t)$ can be either zero or one. However, the MILP tends to choose zero for $x_i^{j,k}(t)$ to obtain a smaller demand (shown in Lemma 1). We calculate demand $y_i^{j,k}(t)$ for all possible combinations of i, j, k, and t in Line 6. For simplicity, we use "$\forall$" to represent the ranges of variables. The task index i ranges from zero to $n-1$. The superscripts j and k range from zero to $N_i - 1$. The maximum integer interval length [3] $H = \lceil \frac{U_{cap}}{1-U_{cap}} \cdot \max_{\tau_i \in \mathcal{T}}(\mathcal{P}_i) \rceil$.

▶ **Lemma 1** (from [18]). *The value of $y_i^{j,k}(t)$ in the MILP is the exact worst-case demand of frame ϕ_i^k over a t-length interval when the first frame to arrive in the interval is ϕ_i^j (with respect to the frame parameters assigned to each frame of τ_i by the MILP).*

Line 7 calculates task τ_i's demand $y_i^j(t)$ whose starting frame in the t-length interval is the j'th frame. In Line 8, the demand $y_i(t)$ is the maximum demand for τ_i over all $y_i^j(t)$. At last, the demand of all tasks $\sum_{i=0}^{n-1} y_i(t)$ is set to be no larger than $\mathcal{L} \cdot t$ as shown in Equation 1.

[2] The term $\mathsf{dbf}_i(t, D_i^{j,k})$ represents the frame demand, and the term $y_i^{j,k}(t)$ is a free variable in the mathematical programming formulation that is used to calculate the demand $\mathsf{dbf}_i(t, D_i^{j,k})$.

Parameter Selection and Exact Feasiblity Test.

```
1   minimize: $\mathcal{L}$
2   subject to:
3       $\boldsymbol{E_i^k} \leq \boldsymbol{\underline{D}_i^k} \leq D_i^k \leq \boldsymbol{\overline{D}_i^k}, \quad \forall i, k.$
        $\boldsymbol{E_i^k} \leq \boldsymbol{\underline{P}_i^k} \leq P_i^k \leq \boldsymbol{\overline{P}_i^k}, \quad \forall i, k.$
4       $D_i^k \leq P_i^k + D_i^{(k+1) \mod N_i}, \quad \forall i, k.$
5       $\sum_{k=0}^{N_i-1} P_i^k \leq \boldsymbol{\mathcal{P}_i}, D_i^{N_i-1} + \sum_{j=0}^{N_i-2} P_i^j \leq \boldsymbol{\mathcal{D}_i}, \quad \forall i.$
6       $\boxed{\begin{array}{l} y_i^{j,k}(t) = x_i^{j,k}(t) \cdot \boldsymbol{E_i^k} + \lfloor \frac{\boldsymbol{t}}{\boldsymbol{\mathcal{P}_i}} \rfloor \cdot \boldsymbol{E_i^k}, \quad \forall i, j, k, t. \\ \frac{\boldsymbol{t} - t_b}{\boldsymbol{\mathcal{P}_i}} \leq x_i^{j,k}(t) - \frac{\boldsymbol{realmin}}{\boldsymbol{\mathcal{P}_i}}, \quad \forall i, j, k, t. \\ t_b = D_i^{j,k} + \lfloor \frac{\boldsymbol{t}}{\boldsymbol{\mathcal{P}_i}} \rfloor \cdot \boldsymbol{\mathcal{P}_i} \end{array}}$
7       $y_i^j(t) = \sum_{k=0}^{N_i-1} y_i^{j,k}(t), \quad \forall i, j, t.$
8       $y_i(t) \geq y_i^j(t), \quad \forall i, j.$
9       $\sum_{i=0}^{n-1} y_i(t) \leq \mathcal{L} \cdot \boldsymbol{t} \quad \forall t.$
10  and: $D_i^k, P_i^k, y_i^{j,k}(t), y_i(t), \mathcal{L} \in \mathbb{R}^*, \boxed{x_i^{j,k}(t) \in \{0,1\}}.$
```

Figure 2 This figure shows the MILP algorithm [18]. In the concave programming and LP-based algorithms (shown in Sections 6 and 7), we only change the frame demand in Line 6 and remove all integer variables $x_i^{j,k}(t)$.

If the system is schedulable, $\mathcal{L} \leq 1$. We minimize $\mathcal{L}$ in the MILP which also minimizes the summation of all task demand over all interval lengths[3] t. The MILP algorithm's necessity and sufficiency for feasibility are proved in Theorem 2.

▶ **Theorem 2** (from [18]). *For arbitrary, real-valued parameters, our MILP is a necessary feasibility test when $\mathcal{L} \leq 1$. When frame parameters are restricted to be integers (i.e., D_i^k, $P_i^k \in \mathbb{N}\ \forall\ i,\ k$), then the MILP is an exact feasibility test when $\mathcal{L} \leq 1$.*

6 The Concave Approximation Algorithm

We reviewed our previously proposed MILP in the last section. In this section, we give a concave approximation algorithm for the MILP algorithm and prove the speed-up factor of the concave approximation algorithm (compared to the optimal FRD/the MILP algorithm) can approach one. Although there is no known efficient way to solve a concave programming problem, our concave approximation algorithm plays a key role in the LP-based algorithms presented in the next section.

6.1 The Concave Functions

We first use the concave function (Equation 2) (illustrated by the blue dashed curve of Figure 3) to approximate the exact frame demand determined by the MILP in Line 6 of Figure 2.

[3] We take integer-valued t since we cannot check all real-valued t. We also use integer constants t in the concave programming and LP-based algorithms later.

$$\boxed{\mathsf{dbf}_i^{\text{concave}}(t, D_i^{j,k}) = \max\left\{0, E_i^k \cdot (1+\delta) - E_i^k \cdot \delta \cdot e^{\mu \cdot (D_i^{j,k} + \lfloor \frac{t}{\mathcal{P}_i} \rfloor \cdot \mathcal{P}_i - t)}\right\} + \lfloor \frac{t}{\mathcal{P}_i} \rfloor \cdot E_i^k} \tag{2}$$

The concave programming algorithm is constructed by replacing all staircase functions in Line 6 of Figure 2 with $y_i^{j,k}(t) = \mathsf{dbf}_i^{\text{concave}}(t, D_i^{j,k})$ and removing all integer variables. The other lines in Figure 2 remain the same.

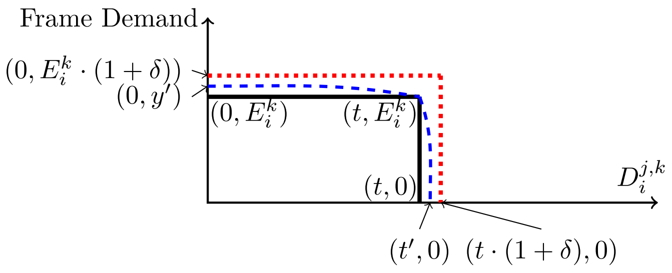


Figure 3 This example shows the frame demand within interval length $t < \mathcal{P}_i$. The blue dashed curve is a concave function and the staircase function in black solid line represents the exact frame demand in the MILP. The red dotted staircase line with error rates δ on both axes represents an upper bound on the concave function.

Equation 2 shows our proposed concave approximation function $\mathsf{dbf}_i^{\text{concave}}(t, D_i^{j,k})$ (e.g., the blue dashed curve in Figure 3) for the k'th frame demand of task τ_i during the t-length interval in which the starting frame is the j'th frame. We define the system-wide maximum error rate[4] δ. The rate δ must be larger than zero to ensure the demand of any approximation function be larger than the staircase function for any given deadline. We set δ as a designer-defined constant in the system, and set the constant $\mu = \frac{1}{\delta} \cdot \ln\left(1 + \frac{1}{\delta}\right)$ as shown in Lemma 3. In Lemma 3, we prove that the maximum error rate of the concave function is smaller than the system maximum error rate δ, and the concave function approaches the staircase function when δ decreases.

▶ **Lemma 3.** *The demand of the concave function in Equation 2 over-approximates the one in the MILP algorithm, and the error rate of the concave function is smaller than the system error constant δ when we set μ in Equation 2 as follows,*

$$\mu = \frac{1}{\delta} \cdot \ln\left(1 + \frac{1}{\delta}\right). \tag{3}$$

Proof. Let δ_y and δ_d be the worst-case error rates on the demand (on y-axis) and deadline (on x-axis) directions of concave functions, respectively. Let $t_b = D_i^{j,k} + \lfloor \frac{t}{\mathcal{P}_i} \rfloor \cdot \mathcal{P}_i - t$. The worst rates happen when, in Figure 3 for example, $E_i^k \cdot (1+\delta_y) = y'$ and $t \cdot (1+\delta_d) = t'$. We will prove that $\delta \geq \delta_y$ and $\delta \geq \delta_d$.

When $0 \leq t_b \leq t$, the largest demand of the concave function happens at $t_b = 0$. By substituting $E_i^k \cdot (1+\delta_y)$ (respectively, 0) for $y_i^{j,k}(t)$ (respectively, t_b), the concave function becomes $E_i^k \cdot (1+\delta_y) = E_i^k \cdot (1+\delta) - E_i^k \cdot \delta \cdot e^{\mu \cdot (0-t)}$. After simplification, we get

[4] The error rate (with respect to the exact frame demand function) of an approximation function is its percentage increase in the y-axis direction for $t \leq D_i^{j,k}$ or its percentage increase in the x-axis dimension if $t > D_i^{j,k}$. The maximum error rate is the largest error rate over all $t > 0$. E.g., the error rate on the x-axis of the point $(t \cdot (1+\delta), 0)$ in Figure 3 is δ. The maximum error rate of any approximation function must be smaller than the system-wide maximum error rate δ.

$\delta_y = \delta - \delta \cdot e^{\mu \cdot (-t)}$. Thus, $\delta \geq \delta_y$ and δ is an upper bound of δ_y. Since the concave function is a decreasing function and it passes the points $(0, E_i^k \cdot (1 + \delta_y))$ and (t, E_i^k), the concave function over-approximates the corresponding demand in MILP when $0 \leq t_b \leq t$.

When $t_b > t$, the maximum error on the deadline direction happens at $t_b = t \cdot (1 + \delta_d)$. By substituting 0 (respectively, $t \cdot (1 + \delta_d)$) for $y_i^{j,k}(t)$ (respectively, t_b), we have $0 = E_i^k \cdot (1 + \delta) - E_i^k \cdot \delta \cdot e^{\mu \cdot (t \cdot (1+\delta_d) - t)}$. After simplification, we have $\delta_d = \frac{1}{t \cdot \mu} \cdot \ln(1 + \frac{1}{\delta})$. We set $\mu = \frac{1}{\delta} \cdot \ln(1 + \frac{1}{\delta})$, and $\delta_d = \frac{\delta}{t}$ after replacing μ in δ_d. Since $t \geq 1$, $\delta \geq \delta_d$. ◀

6.2 Speed-Up Factor Analysis

A speedup factor is a value that quantifies the quality of an approximation algorithm with respect to the optimal scheduling algorithm. A speedup factor $\mathcal{S} > 1$ [4] means that an approximation algorithm can schedule a task system at a speed-$\mathcal{S}$ processor if an optimal algorithm can schedule the system at a speed-one processor.

Let $\mathcal{L}_{MILP}$ be the value of the objective function returned by the MILP algorithm and $\mathcal{L}_{concave}$ be the value returned by the concave programming algorithm. We will prove that $\mathcal{L}_{MILP} < \mathcal{L}_{concave} < \mathcal{L}_{MILP} \cdot (1 + \delta)^2$. $\mathcal{L}_{MILP} < \mathcal{L}_{concave}$ indicates that a task system will be deemed schedulable by the MILP algorithm if the system is schedulable by the concave programming algorithm (which means $\mathcal{L}_{MILP} < \mathcal{L}_{concave} \leq 1$). By the definition of the speed-up factor, $\mathcal{L}_{concave} < \mathcal{L}_{MILP} \cdot (1 + \delta)^2$ indicates that the speed-up factor of our concave programming algorithm is $(1 + \delta)^2$ with respect to the MILP algorithm. In other words, $\mathcal{L}_{concave}/(1 + \delta)^2 < \mathcal{L}_{MILP}$ indicates a task system can be scheduled by the concave programming algorithm under a $(1 + \delta)^2$-speed processor if the system can be scheduled by the MILP algorithm under the corresponding one-speed processor.

We prove $\mathcal{L}_{MILP} < \mathcal{L}_{concave}$ in Lemma 4, and $\mathcal{L}_{concave} < \mathcal{L}_{MILP} \cdot (1+\delta)^2$ from Lemma 5 to Lemma 8. By Lemmas 4 and 8, we prove that the speed-up factor of our concave programming algorithm is $(1 + \delta)^2$ with respect to the MILP algorithm in Theorem 9.

▶ **Lemma 4.** *Let $\mathcal{L}_{MILP}$ and $\mathcal{L}_{concave}$ be the values returned by the MILP and concave programming algorithms (assume they exist), respectively. We have:*

$$\mathcal{L}_{MILP} < \mathcal{L}_{concave}. \tag{4}$$

Proof. Let $\mathcal{L}'_{MILP}$ be the value calculated as follows. Assume there exists such a solver that can solve the concave programming algorithm and return $\mathcal{L}_{concave}$, frame deadlines and separations. We assign the returned frame parameters from the concave programming algorithm to the formulation of the MILP algorithm and get the value of $\mathcal{L}'_{MILP}$.

Under the same values of frame parameters, any frame demand of concave programming algorithm is larger than its corresponding demand of the MILP algorithm, as shown in Lemma 3. The task demands of concave programming algorithm with the preassigned frame parameters are thus also larger than the ones from the MILP approach. When we summarize task demands over any interval length, $\mathcal{L}'_{MILP}$ is thus always less than $\mathcal{L}_{concave}$. Since $\mathcal{L}'_{MILP}$ is calculated by preassigned frame parameters, $\mathcal{L}'_{MILP}$ must not be smaller than $\mathcal{L}_{MILP}$. If the frame parameters returned by the MILP and concave programming algorithms are all identical, $\mathcal{L}'_{MILP} = \mathcal{L}_{MILP}$. In all, $\mathcal{L}_{MILP} \leq \mathcal{L}'_{MILP} < \mathcal{L}_{concave}$ and this lemma is proved. ◀

In order to prove $\mathcal{L}_{concave} < \mathcal{L}_{MILP} \cdot (1 + \delta)^2$, we first define $\mathcal{L}'_{concave}$. Let the MILP algorithm return $\mathcal{L}_{MILP}$, frame deadlines and separations. If we fix the deadline and separation variables of the concave programming formulation to be the values returned by the

MILP, we calculate the value of $\mathcal{L}^{'}_{concave}$. We will prove $\mathcal{L}_{concave} \leq \mathcal{L}^{'}_{concave} < \mathcal{L}_{MILP} \cdot (1+\delta)^2$. $\mathcal{L}_{concave} \leq \mathcal{L}^{'}_{concave}$ is proved in Lemma 5. Based on the demand bound functions defined in Equations 8 and 9, we prove $\mathcal{L}^{'}_{concave} < \mathcal{L}_{MILP} \cdot (1+\delta)^2$ in Lemma 8.

▶ **Lemma 5.** *Let $\mathcal{L}_{concave}$ be the optimal value returned by the concave programming algorithm, and $\mathcal{L}^{'}_{concave}$ be the value calculated by the concave programming algorithm using the frame parameters returned by the MILP. We have:*

$$\mathcal{L}_{concave} \leq \mathcal{L}^{'}_{concave}. \tag{5}$$

Proof. Since the concave programming algorithm minimizes $\mathcal{L}_{concave}$, $\mathcal{L}_{concave}$ must be the smallest value over all feasible-assigned/preassigned frame parameters, and $\mathcal{L}_{concave} < \mathcal{L}^{'}_{concave}$. If frame parameters returned by the MILP and concave programming algorithms are same, $\mathcal{L}_{concave} = \mathcal{L}^{'}_{concave}$. In all, $\mathcal{L}_{concave} \leq \mathcal{L}^{'}_{concave}$. ◀

For ease of proof, we consider a staircase approximation function $\mathsf{dbf}^a_i(t, D^{j,k}_i)$ illustrated by the red dotted line in Figure 3 for task τ_i over the t-length interval, and the solid line shows an example of the staircase demand $\mathsf{dbf}_i(t, D^{j,k}_i)$.

Equation 6 shows $\mathsf{dbf}_i(t, D^{j,k}_i)$ as the k'th frame-demand function of task τ_i over the t-length interval that starts with the j'th frame. The corresponding task demand $\mathsf{dbf}_i(t, \vec{F}_i)$ is shown in Equation 8, and the reasoning is same as to the relationship between $y_i(t)$ and $y^{j,k}_i(t)$ in the MILP algorithm. I.e., we take the maximum demand over all sequences as the task demand. The approximate frame-demand $\mathsf{dbf}^a_i(t, D^{j,k}_i)$ and task-demand $\mathsf{dbf}^a_i(t, \vec{F}_i)$ (for $\mathsf{dbf}_i(t, D^{j,k}_i)$ and $\mathsf{dbf}_i(t, \vec{F}_i)$, respectively) are defined in Equations 7 and 9, respectively. We prove that the approximation demand over-approximates the concave demand in Lemma 6.

$$\mathsf{dbf}_i(t, D^{j,k}_i) = \begin{cases} 0, & 0 \leq t < D^{j,k}_i \\ E^k_i, & D^{j,k}_i \leq t \leq \mathcal{P}_i \\ E^k_i \cdot \lfloor \frac{t}{\mathcal{P}_i} \rfloor + \mathsf{dbf}_i(t - \mathcal{P}_i \cdot \lfloor \frac{t}{\mathcal{P}_i} \rfloor, D^{j,k}_i), & t > \mathcal{P}_i \end{cases} \tag{6}$$

$$\mathsf{dbf}^a_i(t, D^{j,k}_i) = \begin{cases} 0, & 0 \leq t < \frac{D^{j,k}_i}{(1+\delta)} \\ (1+\delta) \cdot E^k_i, & \frac{D^{j,k}_i}{(1+\delta)} \leq t \leq \mathcal{P}_i \\ E^k_i \cdot \lfloor \frac{t}{\mathcal{P}_i} \rfloor + \mathsf{dbf}^a_i(t - \mathcal{P}_i \cdot \lfloor \frac{t}{\mathcal{P}_i} \rfloor, D^{j,k}_i), & t > \mathcal{P}_i \end{cases} \tag{7}$$

$$\mathsf{dbf}_i(t, \vec{F}_i) = \max_{j=0}^{N_i-1} \{ \sum_{k=0}^{N_i-1} \mathsf{dbf}_i(t, D^{j,k}_i) \} \tag{8}$$

$$\mathsf{dbf}^a_i(t, \vec{F}_i) = \max_{j=0}^{N_i-1} \{ \sum_{k=0}^{N_i-1} \mathsf{dbf}^a_i(t, D^{j,k}_i) \} \tag{9}$$

▶ **Lemma 6.** *The demand of task τ_i over any interval length t in Equation 9 is an upper bound of its corresponding concave approximation demand.*

Proof. In Lemma 3, we proved that $\delta_d \leq \delta$. Let $t_\Delta = t - \mathcal{P}_i \cdot \lfloor \frac{t}{\mathcal{P}_i} \rfloor$. From Equation 2 and the definition of δ_d, the concave demand with any value assigned for $D^{j,k}_i \in [0, t_\Delta \cdot (1+\delta_d)]$ is smaller than $E^k_i \cdot (1+\delta)$, and the demand is zero when $D^{j,k}_i > t_\Delta \cdot (1+\delta_d)$. Since $\mathsf{dbf}^a_i(t, D^{j,k}_i) = E^k_i \cdot (1+\delta)$ when $D^{j,k}_i \leq t_\Delta \cdot (1+\delta)$ and $\delta_d \leq \delta$, the demand function $\mathsf{dbf}^a_i(t, D^{j,k}_i)$ over approximates the concave demand. For task-wise demand $\mathsf{dbf}^a_i(t, \vec{F}_i)$, we take the summation of all frame demand $\mathsf{dbf}^a_i(t, D^{j,k}_i)$ of task τ_i over all sequences (sequences differ from the starting j'th frame in the t-length interval), and take the maximum demand over all sequences as the task demand. The task demand $\mathsf{dbf}^a_i(t, \vec{F}_i)$ also over approximates the corresponding concave demand. In all, we have proved this lemma. ◀

With the demand bound functions shown in Equations 8-9, we prove $\mathcal{L}'_{concave} < \mathcal{L}_{MILP} \cdot (1+\delta)^2$ in Lemmas 7-8.

▶ **Lemma 7.** *For the task τ_i's demand $\mathsf{dbf}_i(t, \vec{F}_i)$ and its approximation demand $\mathsf{dbf}^a_i(t, \vec{F}_i)$ in the t-length time interval, we have: $\mathsf{dbf}_i((1+\delta)\cdot t, \vec{F}_i)\cdot(1+\delta) \geq \mathsf{dbf}^a_i(t, \vec{F}_i)$.*

Proof. We first prove $\mathsf{dbf}_i((1+\delta)\cdot t, D_i^{j,k})\cdot(1+\delta) \geq \mathsf{dbf}^a_i(t, D_i^{j,k})$, and $\mathsf{dbf}_i((1+\delta)\cdot t, \vec{F}_i)\cdot(1+\delta) \geq \mathsf{dbf}^a_i(t, \vec{F}_i)$ can be extended by Equations 8 and 9. We classify all interval lengths t in three sets:

$T_1 : 0 \leq t < D_i^{j,k}/(1+\delta)$,
$T_2 : D_i^{j,k}/(1+\delta) \leq t \leq \mathcal{P}_i$,
T_3 : otherwise.

When $t \in T_1$, $\mathsf{dbf}_i(t, D_i^{j,k}) = \mathsf{dbf}^a_i(t, D_i^{j,k}) = 0$. Since demand bound functions are monotonically increasing functions, $\mathsf{dbf}_i((1+\delta)\cdot t, D_i^{j,k})\cdot(1+\delta) \geq \mathsf{dbf}_i(t, D_i^{j,k}) = \mathsf{dbf}^a_i(t, D_i^{j,k})$.

When $t \in T_2$, we know that $\mathsf{dbf}_i(t^*, D_i^{j,k}) = E_i^k$ at $D_i^{j,k} \leq t^* \leq \mathcal{P}_i$ from Equation 6. Let $t^* = t\cdot(1+\delta)$, we have $\mathsf{dbf}_i(t\cdot(1+\delta), D_i^{j,k}) = E_i^k$ at $D_i^{j,k}/(1+\delta) \leq t \leq \mathcal{P}_i$. From Equations 6 and 7, we know that $\mathsf{dbf}_i(t\cdot(1+\delta), D_i^{j,k})\cdot(1+\delta) = \mathsf{dbf}^a_i(t, D_i^{j,k})$ at $D_i^{j,k}/(1+\delta) \leq t \leq \mathcal{P}_i$.

When $t \in T_3$, it is trivial to see the fact that $\mathsf{dbf}_i((1+\delta)\cdot t, D_i^{j,k})\cdot(1+\delta) \geq \mathsf{dbf}^a_i(t, D_i^{j,k})$ since the demand is iteratively calculated from the demand when $t \in T_1 \cup T_2$. ◀

▶ **Lemma 8.** *Let $\mathcal{L}_{MILP}$ be the optimal value returned by the MILP algorithm, and $\mathcal{L}'_{concave}$ be the value calculated by the frame parameters returned by the MILP. We have:*

$$\mathcal{L}'_{concave} < \mathcal{L}_{MILP}\cdot(1+\delta)^2. \tag{10}$$

Proof. Line 9 of Figure 2 shows that $\mathcal{L}$ is the largest value of $\frac{\sum_{i=0}^{n-1} y_i(t)}{t}$ for all values of t in the MILP algorithm (can be derived from Lemma 1). We also require this line in the concave programming algorithm. From Lemma 7, we know that $\mathsf{dbf}_i((1+\delta)\cdot t, \vec{F}_i)\cdot(1+\delta) \geq \mathsf{dbf}^a_i(t, \vec{F}_i)$ for any task τ_i over any t-length interval. Let $t = (1+\delta)\cdot t^*$, we have:

$$\begin{aligned}
\mathcal{L}_{MILP} &= \max_{t>0} \frac{\sum_{\tau_i\in\mathcal{T}} \mathsf{dbf}_i(t, \vec{F}_i)}{t} \quad \text{By Lemma 1}\\
&= \frac{\sum_{\tau_i\in\mathcal{T}} \mathsf{dbf}_i((1+\delta)\cdot t^*, \vec{F}_i)}{(1+\delta)\cdot t^*}\\
&= \frac{\sum_{\tau_i\in\mathcal{T}} \mathsf{dbf}_i((1+\delta)\cdot t^*, \vec{F}_i)\cdot(1+\delta)}{(1+\delta)^2\cdot t^*}\\
&\geq \frac{\sum_{\tau_i\in\mathcal{T}} \mathsf{dbf}^a_i(t^*, \vec{F}_i)}{(1+\delta)^2\cdot t^*} \quad \text{By Lemma 7}\\
&\geq \frac{\mathcal{L}'_{concave}}{(1+\delta)^2} \quad \text{By Lemma 6}
\end{aligned} \tag{11}$$

◀

▶ **Theorem 9.** *When the concave programming algorithm returns integer frame deadlines and separation times, the speed-up factor of our concave programming algorithm with respect to the MILP algorithm is $(1+\delta)^2$.*

Proof. In Lemmas 4, 5, and 8, we have proved that $\mathcal{L}_{MILP} < \mathcal{L}_{concave} < \mathcal{L}_{MILP}\cdot(1+\delta)^2$. $\mathcal{L}_{MILP} < \mathcal{L}_{concave}$ indicates that a task system is deemed schedulable (with integer frame parameters) by the MILP if the task system is deemed schedulable (with integer parameters)

The LP-based Algorithm for GMF-PA tasks.

```
1  Initialize $D$ as $D_i^k \leftarrow (E_i^k/E_i) \cdot \mathcal{P}_i$, $\mathcal{L}^{\text{last}} \leftarrow \infty$, and $\mathcal{L}^{\text{cur}} \leftarrow \infty$
2  **repeat**
3          $\mathcal{L}^{\text{last}} \leftarrow \mathcal{L}^{\text{cur}}$
4          $S \leftarrow$ *computeSlope(D)*
5          $[D, \mathcal{L}^{\text{cur}}] \leftarrow Heuristic\text{-}LP(D, S)$
6     **until** $\mathcal{L}^{\text{last}} - \mathcal{L}^{\text{cur}} < \epsilon$
7  Process frame deadlines D to integers.
8  $[\mathcal{L}^{\text{cur}}] \leftarrow Heuristic\text{-}LP\text{-}fixedDeadline(D, S)$
9  **if** $\mathcal{L}^{\text{cur}} \leq 1$
10    **then return** *schedulable*
11 **else return** *unschedulable*
```

Figure 4 The LP-based algorithm for GMF-PA tasks.

by the concave programming algorithm. $\mathcal{L}_{MILP} < \mathcal{L}_{concave}$ shows our concave programming algorithm is an approximation algorithm for the MILP.

We divide $(1+\delta)^2$ on both sides of the inequality $\mathcal{L}_{concave} < \mathcal{L}_{MILP} \cdot (1+\delta)^2$ to get $\mathcal{L}_{concave}/(1+\delta)^2 < \mathcal{L}_{MILP}$. $\mathcal{L}_{concave}/(1+\delta)^2$ represents that we change the speed of the processor from one to $(1+\delta)^2$. Thus, a task system must be scheduled by the concave programming algorithm with a $(1+\delta)^2$-speed processor if the task system is scheduled by the MILP on a single speed processor. From the definition of the speed-up factor, we have proved that the speed-up factor of our concave programming algorithm with respect to the MILP is $(1+\delta)^2$. ◀

7 The Linear Programming-Based Heuristic Algorithm and its Application to One-Suspension Self-Suspending Tasks

Until now, we have constructed the concave programming approximation algorithm for the MILP-based algorithm. Due to the difficulties in solving concave programming (or non-convex optimization) problems in general, we use a heuristic LP-based scheme to efficiently select frame parameters of GMF-PA tasks, and apply it to self-suspending tasks. For ease of presentation, we let $\underline{D}_i^k = E_i^k$, $\overline{D}_i^k = \mathcal{P}_i$, and $P_i^k = D_i^k$. In this case, frames deadlines are constrained by frame execution time and the l-MAD property. We present the LP-based heuristic algorithm in Section 7.1, and further to optimize the LP-based algorithm to schedule one-suspension self-suspending tasks in Section 7.2.

7.1 The Linear Programming-Based Heuristic Algorithm

The general routine of the LP-based scheme for GMF-PA tasks is: 1) We initialize frame parameters of GMF-PA tasks. 2) Given the frame parameters, we recalculate a set of linear functions, which approximate the staircase functions for frame demands in the MILP, guided by the concave programming algorithm. 3) We run the LP algorithm (shown later) based on the assigned linear functions, and receive frame parameters as outputs. If the difference in $\mathcal{L}$ values between the current and the last iterations is no smaller than some threshold, the program goes back to Step 2. 4) We round frame parameters to integers and run the LP algorithm with the fixed integer-valued parameters to get the final assignment.

$$\boxed{\mathsf{dbf}_i^{\text{linear}}(t, D_i^{j,k}) = \max\{0, s_i^{j,k}(t) \cdot (D_i^{j,k} - t') + E_i^k\} + \lfloor \frac{t}{\mathcal{P}_i} \rfloor \cdot E_i^k, \quad t' = t - \lfloor \frac{t}{\mathcal{P}_i} \rfloor \cdot \mathcal{P}_i} \qquad (12)$$

computeSlope(D).

```
Calculate all $D_i^{j,k'}$ from $D$
$t' \leftarrow t - \lfloor \frac{t}{\mathcal{P}_i} \rfloor \cdot \mathcal{P}_i$
$y_i^{j,k'}(t') \leftarrow E_i^k \cdot (1+\delta) - E_i^k \cdot \delta \cdot e^{\mu \cdot (D_i^{j,k'} - t')}$
**if** $D_i^{j,k'} > t'$
    **then** $s_i^{j,k}(t) \leftarrow (0 - E_i^k)/(\frac{1}{\mu} \cdot \ln(1+\frac{1}{\delta}) + t' - t')$
**elseif** $D_i^{j,k'} == t'$
    **then** $s_i^{j,k}(t) \leftarrow \frac{\partial}{\partial D_i^{j,k}} \left[\mathsf{dbf}_i^{\text{concave}}(t', D_i^{j,k})\right]$
**else** $s_i^{j,k}(t) \leftarrow (y_i^{j,k'}(t') - E_i^k)/(D_i^{j,k'} - t')$
**return** $S$
▷ $S$ is the matrix that stores all slopes $s_i^{j,k}(t)$.
```

Figure 5 This algorithm calculates all slopes given all frame deadlines.

In *The LP-based Algorithm for GMF-PA Tasks* (Figure 4), we initialize frame deadlines by proportional deadline assignment (PDA [15]) to $D_i^k = (E_i^k/E_i) \cdot \mathcal{P}_i$. Given the deadline matrix D which stores all D_i^k, we calculate all slopes and store them in matrix S. We replace Line 6 of Figure 2 with Equation 12 to transform the algorithm into a LP algorithm $\mathit{Heuristic\text{-}LP}(D,S)$ (Line 5 of Figure 4). The slope element $s_i^{j,k}(t)$ of S, which corresponds to $y_i^{j,k}(t)$, is calculated in the algorithm shown in Figure 5 and all lines pass the point (t', E_i^k). The linear functions are illustrated by the red lines in Figures 6-8. If the deadline $D_i^{j,k'}$ (generated from the previous iteration) is smaller than $t' = t - \lfloor \frac{t}{\mathcal{P}_i} \rfloor \cdot \mathcal{P}_i$, we calculate the demand $y_i^{j,k'}(t')$ of the concave function at $D_i^{j,k'}$. The slope of the line is calculated by two points $(D_i^{j,k'}, y_i^{j,k'}(t'))$ and (t', E_i^k) illustrated in Figure 6. If the deadline $D_i^{j,k'}$ equals t', we calculate the slope by taking the tangent of the concave function at the point (t', E_i^k) shown in Figure 7. If the deadline is larger than t', we use two points (t', E_i^k) and $\left(\frac{1}{\mu} \cdot \ln(1+\frac{1}{\delta}) + t', 0\right)$, which is the cross point of the x-axis and the concave function, to calculate the slope, and the line with the slope is shown in Figure 8. The slope matrix S is adjusted in each iteration of the loop in Figure 4.

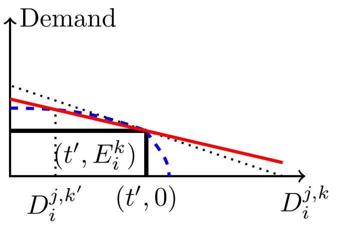


Figure 6 The frame deadline $D_i^{j,k'}$ of the last iteration is smaller than t' in this case.

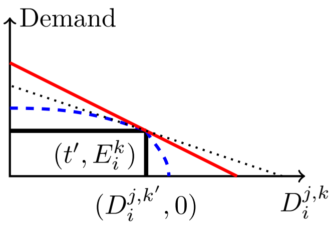


Figure 7 The frame deadline $D_i^{j,k'}$ of the last iteration equals t' in this case.

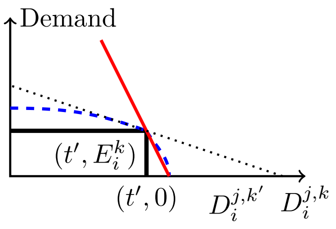


Figure 8 The frame deadline $D_i^{j,k'}$ of the last iteration is larger than t' in this case.

The loop in Figure 4 will not stop recursively calling function $\mathit{Heuristic\text{-}LP}(D,S)$ until the difference of the $\mathcal{L}$ values in two consecutive iterations is smaller than the positive threshold ϵ. $\mathcal{L}^{\text{last}}$ and $\mathcal{L}^{\text{cur}}$ represent the $\mathcal{L}$ values of the last and current iterations, respectively. The $\mathit{Heuristic\text{-}LP\text{-}fixedDeadline}(D,S)$ algorithm (Line 8 of Figure 4) uses the integer deadlines to maintain sufficiency for schedulability, which is proved in Theorem 11. We first round up frame deadlines to be integers. For each task, we keep reducing the largest frame deadline by one until the summation of them equals to its task deadline/period. We assign the deadline variables to the integer values in Line 7 of Figure 4 and the other parts are the same as in the *Heuristic-LP*(D, S) algorithm. The system is schedulable if $\mathcal{L} \leq 1$.

We prove in Theorem 10 that the `while` loop of the algorithm *The LP-based Algorithm for GMF-PA Tasks* function stops after a finite number of iterations. The sufficiency of the LP-based algorithm for schedulability is proved in Theorem 11.

▶ **Theorem 10.** *The* `while` *loop of the function* The LP-based Algorithm for GMF-PA Tasks *stops in a finite number of iterations.*

Proof. We first prove that $\mathcal{L}$ decreases from one iteration to the next. Before each iteration of the algorithm *Heuristic-LP*(D, S), we use the deadline assignment D' from the last iteration to calculate the slopes S of frame functions in the current iteration. Let $\mathcal{L}^{\text{last}}$ be the value of $\mathcal{L}$ in the last iteration. In the current iteration, let us assume that we use the same set of the deadlines D' to calculated the value $\mathcal{L}^{\text{cur}}$.

In the first and third cases shown in Figures 6 and 8, the frame demand is either smaller (if the last iteration is the first iteration) or equal to the one in the last iteration. In the second case, the frame demand is the same as the one in the last iteration. From all cases, we know that the same set of deadlines causes $\mathcal{L}^{\text{cur}} \leq \mathcal{L}^{\text{last}}$. Since we minimize $\mathcal{L}$ in the algorithm, the returned deadlines by the algorithm *Heuristic-LP*(D, S) must generate a value of $\mathcal{L}$ that is smaller than $\mathcal{L}^{\text{cur}}$. Thus, we have proved that $\mathcal{L}$ decreases from one iteration to the next. We also set a threshold to be the difference of the $\mathcal{L}$ values in two consecutive iterations, and we know that the lower bound of $\mathcal{L}$ equals $\sum_{i=1}^{n} U_i$. In either cases, the loop of the function *The LP-based Algorithm for GMF-PA Tasks* stops in a finite number of iterations. ◀

▶ **Theorem 11.** *The LP-based algorithm is a sufficient schedulability test when $\mathcal{L} \leq 1$.*

Proof. This proof is similar to Theorem 2. The sufficiency of any approximation/heuristic algorithm (w.r.t. the MILP algorithm) for schedulability requires two conditions: 1) the demand of the algorithm over any t-length interval is larger than the one in the MILP. 2) frame parameters must take integer values. The first condition ensures that the demand over approximates on any t, and the second condition ensures that the demand only changes at integer values. We require the second condition since all lengths (represented by t) can only be integers in the MILP algorithm. The LP-based algorithm over approximates system demand among all t, and the algorithm adjusts frame deadlines to be integers in the last iteration. ◀

7.2 The Application of the LP-Based Algorithm to One-Suspension Self-Suspending Tasks

The LP-based scheme can be applied to multiple-segment self-suspending tasks directly. In this section, we further optimize the algorithm for one-suspension self-suspending tasks by reducing the number of free variables and equations. Given that n is the number of tasks and H is the maximum interval length, the algorithm uses $8 \cdot n \cdot H + n$ fewer variables and $15 \cdot n \cdot H + n$ fewer number of constraints than the ones in the standard LP-based scheme.

For each task τ_i, we use variables D_i^1 and $\mathcal{P}_i - S_i - D_i^1$ (instead of D_i^1 and D_i^2) to denote frame deadlines to reduce the number of variables and constraints. S_i is the suspension length of task τ_i. In this case, the demand bound function only relies on D_i^1 and t and $\vec{F}_i = [D_i^1, D_i^1, \mathcal{P}_i - S_i - D_i^1, \mathcal{P}_i - S_i - D_i^1]$ since we let $P_i^k = D_i^k$. A task demand falls in four cases which are shown and proved in Theorem 12.

▶ **Theorem 12.** *The demand bound function of a task τ_i lies in one of the following four cases:*

$$\mathsf{dbf}_i(t,\vec{F}_i) = \begin{cases} \mathsf{dbf}_i^1(t,\vec{F}_i) & = \begin{cases} E_i^1, & 0 < D_i^1 \le t \\ 0, & t < D_i^1 < \mathcal{P}_i - S_i - t \\ E_i^2, & \mathcal{P}_i - S_i - t < D_i^1 \le \mathcal{P}_i - S_i \end{cases} \\ & \textit{when } 0 < t < (\mathcal{P}_i - S_i)/2 \\ \mathsf{dbf}_i^2(t,\vec{F}_i) & = \begin{cases} E_i^1, & 0 < D_i^1 < \mathcal{P}_i - S_i - t \\ \max\{E_i^1, E_i^2\}, & \mathcal{P}_i - S_i - t \le D_i^1 \le t \\ E_i^2, & t < D_i^1 < \mathcal{P}_i - S_i \end{cases} \\ & \textit{when } (\mathcal{P}_i - S_i)/2 \le t < \mathcal{P}_i - S_i \\ \mathsf{dbf}_i^3(t,\vec{F}_i) & = E_i^1 + E_i^2, \\ & \textit{when } \mathcal{P}_i - S_i \le t \le \mathcal{P}_i \\ \mathsf{dbf}_i^4(t,\vec{F}_i) & = \lfloor \frac{t}{\mathcal{P}_i} \rfloor \cdot (E_i^1 + E_i^2) + \mathsf{dbf}_i(t - \lfloor \frac{t}{\mathcal{P}_i} \rfloor \cdot \mathcal{P}_i, D_i^1), \\ & \textit{when } t > \mathcal{P}_i \end{cases} \tag{13}$$

Proof. Figures 9-10 show an example of the staircase demand of $\mathsf{dbf}_i^1(t,\vec{F}_i)$ and $\mathsf{dbf}_i^2(t,\vec{F}_i)$ with black solid lines, respectively. Roughly, the two staircase/concave demand curves head toward each other when t increases. The first two cases differ when the two staircase functions meet as t increases. The demand $\mathsf{dbf}_i^3(t,\vec{F}_i)$ considers the total task demand and $\mathsf{dbf}_i^4(t,\vec{F}_i)$ iterates over the first three cases.

For the demand $\mathsf{dbf}_i^1(t,\vec{F}_i)$ in the first case, when $0 < t < (\mathcal{P}_i - S_i)/2$, we know that $t < \mathcal{P}_i - S_i - t$ by simple mathematical transformation. In this case, we have two separate staircase functions as shown in Figure 9. When $D_i^1 \le t$, the demand of the first frame is E_i^1, the demand of the second frame is zero because $D_i^1 \le t < \mathcal{P}_i - S_i - t$. $D_i^1 < \mathcal{P}_i - S_i - t$ means $t < \mathcal{P}_i - S_i - D_i^1$ which indicates the deadline of the second frame is larger than t. Thus, $\mathsf{dbf}_i^1(t,\vec{F}_i) = E_i^1$ when $D_i^1 \le t$. When $t < D_i^1 < \mathcal{P}_i - S_i - t$, $\mathsf{dbf}_i^1(t,\vec{F}_i) = 0$ because $t < D_i^1$ and $t < \mathcal{P}_i - S_i - D_i^1$. When $D_i^1 \ge \mathcal{P}_i - S_i - t$, i.e., $t \ge \mathcal{P}_i - S_i - D_i^1$, the demand $\mathsf{dbf}_i^1(t,\vec{F}_i)$ equals E_i^2. Thus, we have proved that the demand of task τ_i is this case when $0 < t < (\mathcal{P}_i - S_i)/2$.

For the demand $\mathsf{dbf}_i^2(t,\vec{F}_i)$, the proof is similar to the one of the demand $\mathsf{dbf}_i^1(t,\vec{F}_i)$. We know $\mathcal{P}_i - S_i - t \le t$ since $(\mathcal{P}_i - S_i)/2 \le t$. By comparing the deadline and length t, $\mathsf{dbf}_i^2(t,\vec{F}_i) = E_i^1$ when $0 < D_i^1 < \mathcal{P}_i - S_i - t$ and $\mathsf{dbf}_i^2(t,\vec{F}_i) = E_i^2$ when $t < D_i^1 < \mathcal{P}_i - S_i$. When $\mathcal{P}_i - S_i \le t \le \mathcal{P}_i$, we know that either frame can contribute to the demand. However, the two frames cannot contribute together since $t < \mathcal{P}_i - S_i$. In other words, the interval length t cannot fit both frames. Thus, we take the maximum execution of the two frames as the demand when $\mathcal{P}_i - S_i \le t \le \mathcal{P}_i$.

It is trivial to see that $\mathsf{dbf}_i^3(t,\vec{F}_i) = E_i^1 + E_i^2$ when $\mathcal{P}_i - S_i \le t \le \mathcal{P}_i$, and the fourth case iterates over the first three cases. In all, we have proved this theorem. ◀

The LP-based algorithm for one-suspension tasks is based on approximating the exact demand in Theorem 12 and the algorithm *The LP-based Algorithm for GMF-PA Tasks* in Figure 4. We replace Lines 6-8 in the MILP algorithm with the linear functions shown in Equation 16 to get the LP algorithm *Heuristic-LP*(D,S) in Line 5 of Figure 4. The linear functions shown in Equations 14-15 are to approximate the two concave portions of the task demand for $\mathsf{dbf}_i^1(t,\vec{F}_i)$ and $\mathsf{dbf}_i^2(t,\vec{F}_i)$, respectively, illustrated by the red dotted lines in Figures 9-10.

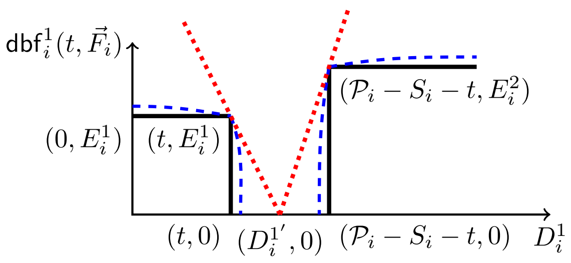


Figure 9 The black solid line shows the demand $\mathsf{dbf}_i^1(t, \vec{F}_i)$, the blue dashed line shows its concave approximation, and the red dotted line shows its linear function when the deadline $D_i^{1'}$ of the last iteration lies between $(t, 0)$ and $(\mathcal{P}_i - S_i - t, 0)$.

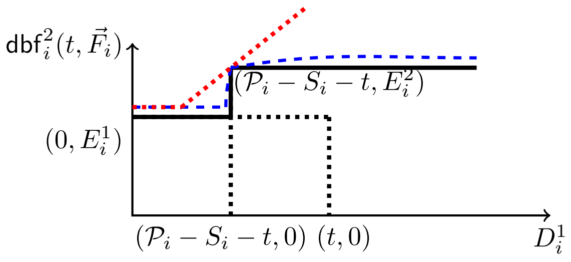


Figure 10 Similar to Figure 9, the dashed and dotted lines show the concave and linear functions of the demand $\mathsf{dbf}_i^2(t, \vec{F}_i)$, shown with the solid line, respectively. The black dotted line shows the frame-wise demand.

$$\mathsf{dbf}_i^{1,\text{linear}}(t, \vec{F}_i) = \begin{cases} \max \begin{cases} \mathsf{dbf}_i^{\text{linear}}(t, D_i^1) \\ \mathsf{dbf}_i^{\text{linear}}(t, \mathcal{P}_i - S_i - D_i^1) \end{cases} \\ \text{when } 0 < D_i^{1'} \leq t \\ \max \begin{cases} \frac{0 - E_i^1}{D_i^{1'} - t'} \cdot (D_i^1 - t) + E_i^1 \\ \frac{0 - E_i^2}{D_i^{1'} - (\mathcal{P}_i - S_i - t')} \cdot (D_i^1 - (\mathcal{P}_i - S_i - t)) + E_i^2 \\ 0 \end{cases} \\ \text{when } t < D_i^{1'} < \mathcal{P}_i - S_i - t \\ \max \begin{cases} \mathsf{dbf}_i^{\text{linear}}(t, D_i^1) \\ \mathsf{dbf}_i^{\text{linear}}(t, \mathcal{P}_i - S_i - D_i^1) \end{cases} \\ \text{when } \mathcal{P}_i - S_i - t \leq D_i^{1'} < \mathcal{P}_i - S_i \end{cases} \tag{14}$$

$$\mathsf{dbf}_i^{2,\text{linear}}(t, \vec{F}_i) = \begin{cases} \max \begin{cases} \mathsf{dbf}_i^{\text{linear}}(t, D_i^1) \\ E_i^2 \end{cases} \\ \text{when } E_i^1 \geq E_i^2 \\ \max \begin{cases} \mathsf{dbf}_i^{\text{linear}}(t, \mathcal{P}_i - S_i - D_i^1) \\ E_i^1 \end{cases} \\ \text{when } E_i^1 < E_i^2 \end{cases} \tag{15}$$

$$\mathsf{dbf}_i^{\text{linear}}(t,\vec{F_i}) = \begin{cases} \mathsf{dbf}_i^{1,\text{linear}}(t,\vec{F_i}) & \text{when } 0 < t < (\mathcal{P}_i - S_i)/2 \\ \mathsf{dbf}_i^{2,\text{linear}}(t,\vec{F_i}) & \text{when } (\mathcal{P}_i - S_i)/2 \le t < \mathcal{P}_i - S_i \\ \mathsf{dbf}_i^{3,\text{linear}}(t,\vec{F_i}) & = E_i^1 + E_i^2, \\ & \text{when } \mathcal{P}_i - S_i \le t \le \mathcal{P}_i \\ \mathsf{dbf}_i^{4,\text{linear}}(t,\vec{F_i}) & = \lfloor \frac{t}{\mathcal{P}_i} \rfloor \cdot (E_i^1 + E_i^2) + \mathsf{dbf}_i(t - \lfloor \frac{t}{\mathcal{P}_i} \rfloor \cdot \mathcal{P}_i, D_i^1), \\ & \text{when } t > \mathcal{P}_i \end{cases} \tag{16}$$

The approximation demand $\mathsf{dbf}_i^{\text{linear}}(t,\vec{F_i})$ is calculated based on the t-length interval. Equation 14 shows that the task demand is approximated when $0 < t < (\mathcal{P}_i - S_i)/2$. This case is illustrated by the red dashed lines shown in Figure 9. The functions are also based on the LP-based iterative process and the initial deadline D_i^1 is assigned by PDA $(\mathcal{P}_i - S_i) \cdot \frac{E_i^1}{E_i^1 + E_i^2}$. The slope of the linear function depends on the frame deadline $D_i^{1'}$ from the last iteration. If the deadline $D_i^{1'}$ lies in the region $(t, \mathcal{P}_i - S_i - t)$, we use the two red dotted lines shown in Figure 9 to approximate the staircase demand. The first line passes the points (t, E_i^1) and $(D_i^{1'}, 0)$, and the second line passes the points $(D_i^{1'}, 0)$ and $(\mathcal{P}_i - S_i - t, E_i^2)$. When the frame deadline $D_i^{1'}$ lies in the region $(0, t]$ or $[\mathcal{P}_i - S_i - t, \mathcal{P}_i - S_i)$, we reuse the linear function $\mathsf{dbf}_i^{\text{linear}}(t, D_i^1)$ shown in Equation 12 to calculate the slopes.

Equation 15 shows the task demand when $(\mathcal{P}_i - S_i)/2 \le t < \mathcal{P}_i - S_i$, the demand functions differ by the values of E_i^1 and E_i^2. In the case of the demand $\mathsf{dbf}_i^2(t,\vec{F_i})$, the first line equals $\min\{E_i^1, E_i^2\}$, and the second line uses the previous method *computeSlope(D)* to adjust the slope of the linear function as shown in Figure 10. Figure 10 shows the approximate lines when $E_i^1 < E_i^2$, and the case is similar when $E_i^1 \ge E_i^2$. When $t \ge \mathcal{P}_i - S_i$, the demand $\mathsf{dbf}_i^{3,\text{linear}}(t,\vec{F_i})$ and $\mathsf{dbf}_i^{4,\text{linear}}(t,\vec{F_i})$ are identical to $\mathsf{dbf}_i^3(t,\vec{F_i})$ and $\mathsf{dbf}_i^4(t,\vec{F_i})$ of Equation 13, respectively. Thus, we have created the LP-based algorithm for one-suspension tasks.

8 Experiments

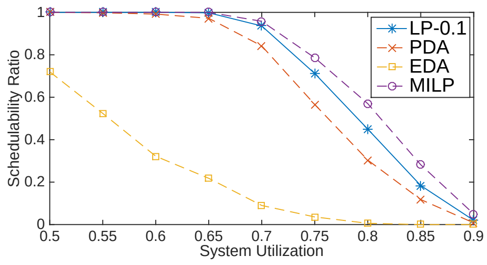
We implement our LP-based algorithms using the commercial solver GUROBI [17] in MATLAB on a 2 GHz Intel Core i5 processor and 8 GB memory machine. We compare our LP-based algorithm with the MILP algorithm [18] and its application to self-suspending tasks [9, 14] on uniprocessor systems. The algorithm LP-δ is the LP-based schedulability test given the maximum error δ of the concave programming algorithm. The algorithm n_{iter}-LP-δ limits the number of iterations to be n_{iter}. Note that we set $\delta = 0.1$, as the constant $\mu = \frac{1}{\delta} \cdot \ln\left(1 + \frac{1}{\delta}\right)$ (e.g. the exponential constants in Equation 2) will be out of range if δ is too small.

The MILP algorithm is introduced in Section 5. The algorithm EDA (equal deadline assignment [9, 3]) assigns each frame the same deadline ($D_i^k = (\mathcal{P}_i - \sum_{i=0}^{N_i-1} S_i^k)/N_i$), and the algorithm PDA [15, 3] assigns frame deadlines proportional to frame execution time ($D_i^k = (\mathcal{P}_i - \sum_{i=0}^{N_i-1} S_i^k) \cdot E_i^k/E_i$). Note that we use the schedulability test in the GMF model [3] with the EDA and PDA deadline assignment, since the upper bound of the maximum interval length is bounded [3]. The details of application from GMF-PA to self-suspending tasks can be found in a previous paper [18]. Comparative results on tasks with one suspension and multiple suspensions are shown in Section 8.1 and 8.2, respectively.

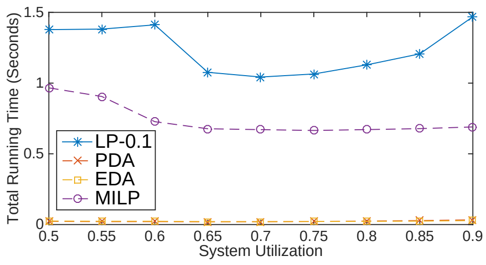
8.1 The Experiments for One-Suspension Self-Suspending Tasks

For one-suspension self-suspending tasks, we compare schedulability ratio and total running time among the algorithms in Figures 11a and 11b, respectively. Since the MILP algorithm does not scale well with an increasing number of tasks (Figure 12) and task periods, we test multiple-suspension self-suspending tasks in Figures 14a and 15a without the MILP algorithm. The schedulability ratio is the number of feasible systems over the total systems. The total running time consists of matrix building time and solver running time.

In the task systems, task periods $\mathcal{P}_i$ are randomly generated in the range $[P_{low}, P_{high}]$. P_{low} and P_{high} are the low and high bounds of the task periods. The UUniFast algorithm [5] divides the utilizations U_i of n tasks under system utilization U_{cap}. The total execution time is $E_i = \mathcal{P}_i \cdot U_i$, and the suspension delay is generated from $[S_{low} \cdot (1-U_i) \cdot \mathcal{P}_i, S_{high} \cdot (1-U_i) \cdot \mathcal{P}_i]$. S_{low} and S_{high} in suspension range $[S_{low}, S_{high}]$ are the low and high suspension index bounds, respectively. The UUniFast algorithm also divides the total execution time into frame execution times. ϵ represents the threshold in the LP-based algorithm shown in Figure 4 and is set to be 0.01. Since all algorithms perform well under small system utilization U_{cap}, we focus on the experiments whose system utilization $U_{cap} \geq 0.5$.



(a) The schedulability ratio of the algorithms at system utilization [0.5, 0.9].



(b) The average running time of the algorithms at system utilization [0.5, 0.9].

Figure 11 The comparison of our LP-based algorithm with the MILP and other polynomial-time algorithms on schedulability ratio and average running time.

In Figures 11a and 11b, the x-axes represent the system utilization $U_{cap} \in [0.5, 0.9]$ with a step size of 0.05. Each task system contains five tasks. The task configuration parameters are $P_{low} = 10$, $P_{high} = 100$, $S_{low} = 0.3$, and $S_{high} = 0.6$. The y-axes represent the schedulability ratio and total running time in Figures 11a and 11b, respectively. The data are the average numbers of 500 runs on each U_{cap}. Figure 11a shows that our LP-δ is better than PDA and EDA algorithms in terms of schedulability ratio. The iteration numbers of all tested LP-δ algorithm are smaller than five. The multiple runs of the LP algorithm make the LP-δ algorithm take slightly longer than the MILP algorithm shown in Figure 11b. The MILP can be relatively efficient for small enough task systems; however, as the number of tasks/frames increases, the MILP running time increases exponentially. Note that in Figure 11, we focus on a small system where we can gauge the effectiveness of the LP in comparison with the MILP and other algorithms. With $U_{cap} = 0.5$, Figure 12 shows that the execution time of the MILP algorithm increases dramatically when the number of tasks increases. Multiple input dimensions affect the execution time of the MILP algorithm, e.g., the task periods. Task periods directly affect the number of integer variables of the MILP algorithm and the running time is longer with higher task periods even when the number of tasks in the system is small. The running time of the LP-based algorithm scales relatively well.

Since we use the concave programming algorithm to guide the LP-based algorithm and have not proved a speed-up factor for the LP-based algorithm, we perform experiments on $\mathcal{L}$ value and maximum error ($\sqrt{\mathcal{L}/\mathcal{L}_{\text{MILP}}} - 1$ by the transformation of Theorem 9). $\mathcal{L}$ shows how close the value of the heuristic algorithm is to the MILP algorithm. $\mathcal{L}$ indicates the minimization of the maximum demands over all tested intervals. E.g., assume there exist two heuristic algorithms that generate $\mathcal{L} = 0.2$ and 0.9, respectively. Both algorithms

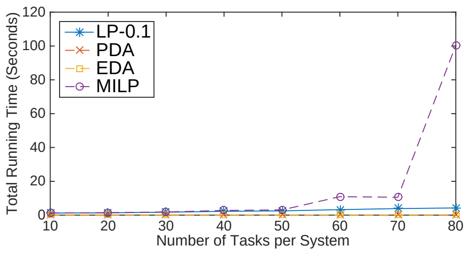
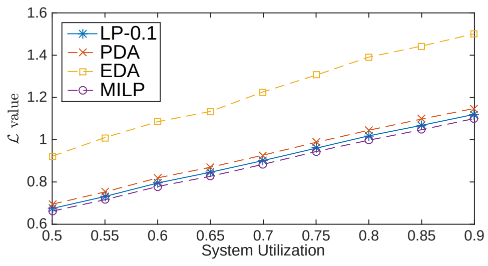


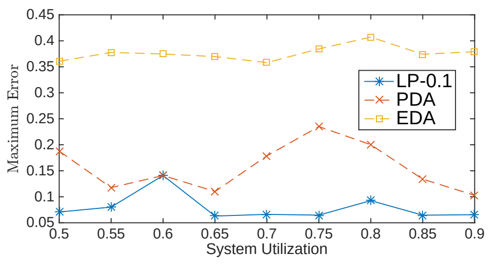
Figure 12 The average running time of the algorithms as the number of tasks increases.

will give successful schedules in the schedulability ratio test, but the one with $\mathcal{L} = 0.2$ is a tighter schedule compared to the other one. If $\mathcal{L} > 1$, the system is not schedulable. We also compare the maximum error of the LP-δ algorithm since the error can be larger than δ.

Figure 13a shows the average $\mathcal{L}$ value of the algorithms among all system utilization points. The LP-0.1 algorithm returns the closest values to the MILP algorithm. The maximum error values shown in Figure 13b take the maximum values among 500 runs in each utilization point. Our LP-based algorithm returns the smallest error across all algorithms.



(a) The $\mathcal{L}$ value of the algorithms at system utilization [0.5, 0.9].



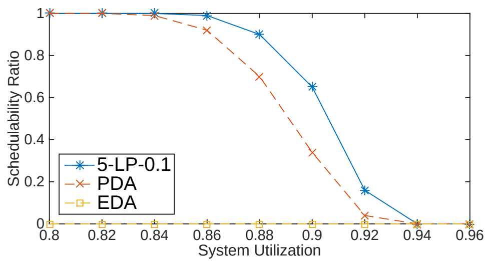
(b) The maximum error of the algorithms compared with the MILP algorithm.

Figure 13 The quality of the LP-based algorithm on the $\mathcal{L}$ value and the maximum system error.

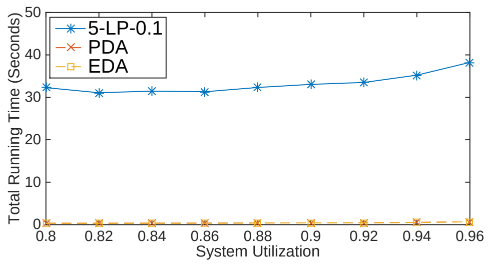
8.2 The Experiments for Multiple-Suspension Self-Suspending Tasks

Among the shown experiments on self-suspending tasks with one suspension frame, the average number of iterations of the LP-based algorithm is smaller than five among all system utilization U_{cap}. Since we believe that the algorithms can approach local optimal with a small number of iterations, we fix the number of iterations to five and test on multiple-suspending tasks. In Figures 14 and 15, the data for each system utilization point is based on 100 runs. Each run of the system contains 30 tasks and each task contains six execution frames separated by five suspending frames (11 frames in total). $P_{low} = 10$ and $P_{high} = 100$. Since the MILP-based approach in this setting takes much longer than the LP-based algorithm, we do not include the MILP-based approach in this experiment. The MILP-based approach takes more than $1.5 * 10^3$ (respectively, $3.0 * 10^3$) seconds with optimality gap (the gap between the lower and upper objective bounds) which is larger than 10% (respectively, 5%).

In Figure 14a, the system utilization $U_{cap} \in [0.8, 0.96]$ with step size of 0.02 is shown on the x-axis. Figure 14a has the suspension range with $S_{low} = 0.1$ and $S_{high} = 0.3$. In Figure 15a, the system utilization $U_{cap} \in [0.5, 0.9]$ with a step size of 0.05 is shown on the x-axis. Figure 15a has the suspension range with $S_{low} = 0.3$ and $S_{high} = 0.6$. Figures 14a and 15a show that our LP-δ is the best among all polynomial-time algorithms in terms of



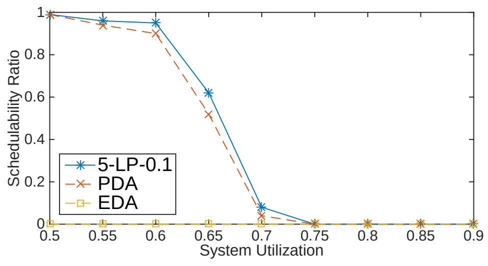
(a) The schedulability ratio of the algorithms at system utilization [0.8, 0.96].



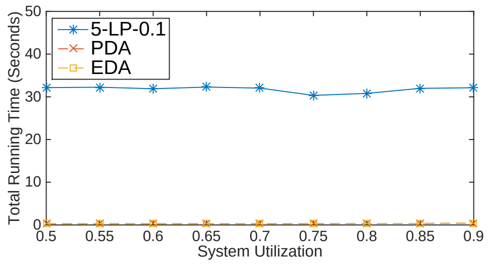
(b) The average running time of the algorithms at system utilization [0.8, 0.96].

Figure 14 Comparison of our LP-based algorithm with other polynomial-time algorithms on the schedulability ratio and average running time.

schedulability ratio. The running times in Figures 14b and 14b reveal that LP-δ also scales well. The improvements for low suspension range $[0.1, 0.3]$ are better than the one with long range $[0.3, 0.6]$. The reason is that when the system specification has more slack time (small frame execution time and short suspending length), the LP-based algorithms can be "trained" to get near optimal parameters during the five iterations. In other words, e.g., the frames deadlines will be equal to their corresponding execution times if there are no slacks for all tasks, and all algorithms will return identical frame deadlines.



(a) The schedulability ratio of the algorithms at system utilization [0.5, 0.9].



(b) The average running time of the algorithms at system utilization [0.5, 0.9].

Figure 15 The comparison of our LP-based algorithm with other polynomial-time algorithms on schedulability ratio and average running time.

Our LP-based algorithm always yields higher schedulability ratio compared to other polynomial-time algorithms. The average running time is competitive overall even when compared with non-mathematical-programming based algorithms such as EDA/PDA.

9 Conclusions

In this paper, we propose a concave programming approximation algorithm and prove its speed-up factor (can approach one) compared to the optimal MILP algorithm. Under the guidance of the tunable small speed-up factor, we present the general LP-based scheme to schedule GMF-PA tasks. We further optimize the LP-based algorithm and apply it to schedule one-suspension tasks. Extensive experiments show that our algorithms improve the schedulability ratio and have competitive running time compared to the previous results.

References

1 B. Andersson. Schedulability analysis of generalized multiframe traffic on multihop-networks comprising software-implemented ethernet-switches. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, April 2008.

2 S. Baruah. Dynamic- and Static-priority Scheduling of Recurring Real-time Tasks. *Real-Time Syst.*, 24(1):93–128, January 2003.

3 S. Baruah, D. Chen, S. Gorinsky, and A. Mok. Generalized Multiframe Tasks. *Real-Time Systems*, pages 5–22, 1999.

4 S. Baruah and N. Fisher. The Partitioned Multiprocessor Scheduling of Sporadic Task Systems. In *Proceedings of the 26th Real-Time Systems Symposium*, pages 321–329, 2005.

5 E. Bini and G. C. Buttazzo. Measuring the Performance of Schedulability Tests. *Real-Time Systems*, pages 129–154, 2005.

6 G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic Scheduling for Flexible Workload Management. *IEEE Transactions on Computers*, pages 289–302, March 2002.

7 D. Buttle. Real-Time in the Prime-Time. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages xii–xiii, July 2012. doi:10.1109/ECRTS.2012.7.

8 T. Chantem, X. Wang, M.D. Lemmon, and X.S. Hu. Period and Deadline Selection for Schedulability in Real-Time Systems. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, pages 168–177, July 2008.

9 J. J. Chen and C. Liu. Fixed-Relative-Deadline Scheduling of Hard Real-Time Tasks with Self-Suspensions. In *Proceedings of the Real Time Systems Symposium (RTSS)*, December 2014.

10 J. J. Chen, G. von der Bruggen, W. H. Huang, and C. Liu. State of the Art for Scheduling and Analyzing Self-Suspending Sporadic Real-Time Tasks. In *Proceedings of the Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2017.

11 S. Ding, H. Tomiyama, and H. Takada. Scheduling Algorithms for I/O Blockings with a Multi-frame Task Model. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, August 2007.

12 P. Ekberg and W. Yi. Uniprocessor Feasibility of Sporadic Tasks Remains coNP-complete Under Bounded Utilization. In *Proceedings of the 36th IEEE Real-Time Systems Symposium (RTSS)*, 2015.

13 M. R. Garey, D. S. Johnson, and Ravi Sethi. The Complexity of Flowshop and Jobshop Scheduling. *Math. Oper. Res.*, 1(2):117–129, May 1976. doi:10.1287/moor.1.2.117.

14 W.H. Huang and J.J. Chen. Self-Suspension Real-Time Tasks under Fixed-Relative-Deadline Fixed-Priority Scheduling. In *Proceedings of the Design, Automation, and Test in Europe (DATE)*, March 2016.

15 J. Liu. Real-Time Systems. *Prentice Hall*, 2000.

16 A.K. Mok and D. Chen. A multiframe model for real-time tasks. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 22–29, December 1996.

17 Gurobi Optimization. GUROBI: The state-of-the-art mathematical programming solver. URL: http://www.gurobi.com/.

18 B. Peng and N. Fisher. Parameter Adaptation for Generalized Multiframe Tasks and Applications to Self-Suspending Tasks. In *Proceedings of the 22nd Embedded and Real-Time Computing Systems and Applications (RTCSA)*, August 2016.

19 B. Peng, N. Fisher, and T. Chantem. MILP-based deadline assignment for end-to-end flows in distributed real-time systems. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, RTNS '16, pages 13–22, New York, NY, USA, 2016. ACM. doi:10.1145/2997465.2997498.

20 Bo Peng and Nathan Fisher. Parameter adaptation for generalized multiframe tasks: schedulability analysis, case study, and applications to self-suspending tasks. *Real-Time Systems*, 2017.

21 F. Ridouard, P. Richard, and F. Cottet. Negative results for scheduling independent hard real-time tasks with self-suspensions. In *Proceedings of the 25th Real-Time Systems Symposium*, pages 47–56, December 2004. `doi:10.1109/REAL.2004.35`.

22 J. M. Rivas, J. J. Gutiérrez, J. C. Palencia, and M. G. Harbour. Schedulability Analysis and Optimization of Heterogeneous EDF and FP Distributed Real-Time Systems. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 195–204, July 2011. `doi:10.1109/ECRTS.2011.26`.

23 M. Stigge, P. Ekberg, N. Guan, and W. Yi. The Digraph Real-Time Task Model. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 71–80, April 2011. `doi:10.1109/RTAS.2011.15`.

24 Martin Stigge and Wang Yi. Graph-based models for real-time workload: a survey. *Real-Time Systems*, 2015.

Response-Time Analysis of Limited-Preemptive Parallel DAG Tasks Under Global Scheduling

Mitra Nasri
Delft University of Technology (TUDelft), Delft, The Netherlands

Geoffrey Nelissen
CISTER Research Centre, Polytechnic Institute of Porto (ISEP-IPP), Portugal

Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

Abstract

Most recurrent real-time applications can be modeled as a set of sequential code segments (or blocks) that must be (repeatedly) executed in a specific order. This paper provides a schedulability analysis for such systems modeled as a set of parallel DAG tasks executed under any limited-preemptive global job-level fixed priority scheduling policy. More precisely, we derive response-time bounds for a set of jobs subject to precedence constraints, release jitter, and execution-time uncertainty, which enables support for a wide variety of parallel, limited-preemptive execution models (e.g., periodic DAG tasks, transactional tasks, generalized multi-frame tasks, etc.). Our analysis explores the space of all possible schedules using a powerful new state abstraction and state-pruning technique. An empirical evaluation shows the analysis to identify between 10 to 90 percentage points more schedulable task sets than the state-of-the-art schedulability test for limited-preemptive sporadic DAG tasks. It scales to systems of up to 64 cores with 20 DAG tasks. Moreover, while our analysis is almost as accurate as the state-of-the-art exact schedulability test based on model checking (for sequential non-preemptive tasks), it is three orders of magnitude faster and hence capable of analyzing task sets with more than 60 tasks on 8 cores in a few seconds.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Software and its engineering → Real-time schedulability

Keywords and phrases parallel DAG tasks, global multiprocessor scheduling, schedulability analysis, non-preemptive jobs, precedence constraints, worst-case response time, OpenMP

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.21

Supplement Material The source code of the analysis tool is available at `https://github.com/brandenburg/np-schedulability-analysis`.

Funding This work was partially supported by national funds through FCT/MCTES (Portuguese Foundation for Science and Technology), within the CISTER Research Unit (UID/CEC/04234); by the Operational Competitiveness Programme and Internationalization (COMPETE 2020) under the PT2020 Partnership Agreement, through the European Regional Development Fund (ERDF), and by national funds through the FCT, within project POCI-01-0145-FEDER-029119 (PReFECT); as well as by the European Union through the Clean Sky 2 Joint Undertaking, under H2020 (H2020-CS2-CFP08-2018-01) grant agreement number 832011 (THERMAC).

1 Introduction

With the proliferation of multicore and many-core processing platforms, the embedded systems world is steadily moving towards developing critical applications as (highly) parallel programs. In embedded real-time systems in particular, parallel programming approaches allow for more efficient use of a computing platform's resources, resulting in lower response times and improved power consumption. For instance, the automotive industry adopted

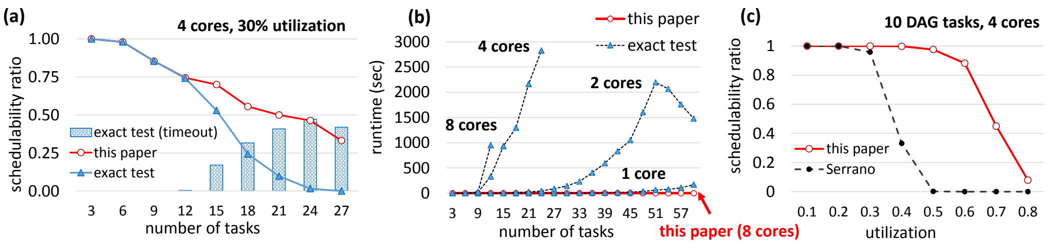


Figure 1 **(a)** schedulability ratio of the exact test [46] vs. our test for independent non-preemptive periodic tasks, **(b)** average runtime of the exact test [46] vs. our test for *schedulable* independent non-preemptive periodic task sets with total utilization of 30%, **(c)** schedulability ratio of Serrano et al.'s test [39] vs our test for DAG tasks. In inset (a), given a one-hour budget, starting from 12 tasks the *exact* test reports *lower* schedulability than the proposed test due to frequent timeouts. See Sec. 5 for a detailed description of the experimental setup.

multicore processors already more than six years ago, and their applications are routinely composed of thousands of *runnables* executing in parallel [24]. Such runnables are sequential code segments that perform simple operations, which are composed to produce complex applications by imposing precedence constraints that must be respected at runtime (to enforce a predictable ordering and to respect data dependencies). Because of the application domain, additional constraints on release and completion times are also associated with runnables to ensure temporal correctness, control performance, ease of synchronization and, in consequence, ease of integration of concurrent applications on multicore platforms.

Similarly to the automotive case, a wide variety of industrially relevant systems boil down to the execution of a set of functions with precedence constraints where a function is simply a *sequential* execution segment of a bigger, potentially *parallel*, application. Such applications (henceforth called *tasks*) may be modeled with *Directed Acyclic Graphs* (DAGs). Nodes of a DAG represent sequential code segments, and edges model their precedence constraints. Each application represented by a DAG releases *jobs* based on timer events or inputs regularly received from the environment following periodic or non-periodic activation patterns (e.g., multi-frame or multi-rate tasks [16, 6, 20]). Robotics applications executed upon the ROS middleware, machine learning algorithms developed with the TensorFlow or Pytorch frameworks, or applications developed with OpenMP are other notable examples of systems that are often time-driven and that may be naturally modeled with DAGs. To summarize, parallel DAG tasks are the characteristic real-time workload of the multicore age and thus of central interest for schedulability analysis.

In this work, we consider a limited-preemptive task model, where nodes of a DAG must execute non-preemptively, but higher-priority workload may preempt the execution of a DAG between the execution of any two of its nodes. This execution model is motivated by many previous studies [11, 34, 2, 29, 37, 39] that have shown that non-preemptive (or limited-preemptive) scheduling improves the timing predictability of jobs running on a multicore platform, since it reduces the number of context switches, increases cache predictability [45], and improves the accuracy of *worst-case execution time* (WCET) estimates and worst-case blocking bounds (e.g., due to contention for shared resources).

Two types of frameworks exist for the schedulability analysis of such systems today. Exact solutions based on model checkers or constraint programming [46, 42], and sufficient (but inexact) solutions usually based on some sort of response-time analysis [39, 15, 13, 14].

It has been demonstrated that exact analyses based on constrained programming or model checkers such as Uppaal do not scale well [46, 42]. For example, Figure 1(b) shows the time required to deem a simple non-preemptive periodic task set schedulable using Uppaal as a function of the number of tasks for several different core counts on a 3.3 GHz Intel Xeon machine with 256 GiB RAM. Even for such a simple model without intra-task parallelism or precedence constraints, Uppaal requires an average of 45 minutes to analyze 24 tasks on 4 cores used at only 30% of their capacity (i.e., the total platform utilization is 30%), with nearly 50% of the tested workloads timing out after 1 hour (see Figure 1(a)). Worse, it cannot solve the problem at all in less than one hour for 12 tasks (or more) on 8 cores with a total utilization of only 30%. Clearly, such a solution can realistically be used only for very small systems, which limits practicality.

On the other hand, classic sufficient schedulability analyses following the standard response-time analysis paradigms are usually fast but very pessimistic. For instance, as seen in Figure 1(c), the only sufficient test existing for the schedulability of limited-preemptive DAGs scheduled by a global fixed-priority scheduler (proposed by Serrano et al. [39]) cannot detect that *any* of the generated task sets with a total utilization larger than 50% (4 cores, 10 DAG tasks) is schedulable, when in fact at least 90% of them are. This pessimism reaches a level that calls into question the utility of such tests in industrial settings.

In this paper, we propose a new approach for the schedulability analysis of *limited-preemptive DAG tasks* that presents a more balanced tradeoff between runtime and accuracy. Case in point, w.r.t. analysis speed, in the scenario shown in Figure 1(a), our solution solves the schedulability problem of non-preemptive tasks *almost* optimally (empirically, almost all schedulable workloads are in fact deemed schedulable) in less than 10 seconds on average, while Uppaal needed tens of minutes to reach the same conclusion (and frequently exceeded the one-hour timeout). Furthermore, w.r.t. analysis accuracy for DAG tasks, the proposed analysis clearly increases the number of workloads successfully detected as being schedulable in comparison to the solution of Serrano et al. by a substantial margin (see Figure 1(c)).

The analysis presented in this paper covers any *global job-level fixed-priority* (JLFP) scheduler (e.g., global limited-preemptive earliest-deadline first (G-LP-EDF) or fixed-priority (G-LP-FP) scheduling). Specifically, each node of each DAG instance released by a task can have a distinct priority, a distinct release time, and is assumed to execute non-preemptively. We allow for the practical, but analytically challenging complication that each node may experience release jitter and execution-time uncertainty, which in combination with non-preemptivity results in scheduling anomalies that are notoriously difficult to analyze precisely.

To strike a good balance between accuracy and runtime, our analysis constructs a *schedule-abstraction graph* that abstracts all possible orderings of job dispatch times resulting from the underlying JLFP scheduling policy, based on which we derive bounds on the best- and worst-case response time of each job. This approach requires: **(i)** a *system-state abstraction* that represents the state in which the system may be after a given sequence of scheduling events, **(ii)** sound *exploration rules* that reflect how new system states may be reached from a given state, and **(iii)** *merging rules* for the aggregation of similar states to defer, as long as possible, the usual state-space explosion problem.

As a key technical contribution, this paper introduces a *new system-state abstraction* in which the number of newly created states at the end of each exploration step is *independent of the number of cores*, which ensures scalability to large multicore platforms. Furthermore, our new abstraction also allows for aggressive merging rules, and hence greatly reduces the number of system states that must be investigated to cover all relevant job schedules. Based on this novel technique, **(i)** we devise a schedule-abstraction graph generation algorithm that considers the precedence constraints of DAG tasks and ensures a small per-state memory footprint and low per-state computational costs, **(ii)** we prove the system state-space exploration and merging rules to be sound, and **(iii)** we report results on extensive experiments involving both synthetic DAGs and actual DAGs from parallel benchmark applications. The experiments show the proposed method to scale to systems with up to 64 cores, to be able to identify up to 90 percentage points more schedulable task sets in comparison to the state-of-the-art response-time analysis for limited-preemptive sporadic DAG tasks [39], and to be three orders of magnitude faster than model-checking approaches [46].

2 Related Work

The schedulability analysis of a set of independent non-preemptive *sporadic* tasks scheduled by a global scheduling policy such as G-LP-EDF or G-LP-FP has been studied in several works [5, 19, 23, 22, 11]. These analyses, however, do not account for release jitter and become needlessly pessimistic when applied to *periodic* tasks or jobs with regular, yet not necessarily periodic, activation patterns [33] as they fail to discount many execution scenarios that are impossible in such systems commonly found in industry.

In response to the need for supporting task models with more complicated job activation patterns, Stigge et al. [41] and Abdullah et al. [1] provided schedulability analyses for non-preemptive digraphs and digraphs with a mixed set of preemptive and non-preemptive nodes, respectively. The digraph model was later extended to support a rendezvous synchronization mechanism [31]. However, to the best of our knowledge, there is no result yet on digraphs with non-preemptive nodes and complex inter-task precedence constraints.

To work around the lack of a schedulability test for non-preemptive DAGs, Saifullah et al. [37] provided solutions to convert a DAG to a set of independent jobs whose arrival times and deadlines are assigned in a way that respects the DAG's given precedence constraints. This job set is then converted to an equivalent periodic task set and evaluated using Baruah's [5] or Guan's [19] schedulability analyses for independent, non-preemptive tasks. This approach, however, suffers from the pessimism inherent in the *decomposition* step, i.e., regardless of the accuracy of the underlying schedulability tests, many schedulable DAG tasks will be deemed unschedulable simply because the decomposition technique may not be able to find feasible parameters for the decomposed independent tasks.

Liu and Anderson extensively studied sporadic processing pipelines and DAGs under global scheduling in a soft real-time context [25, 26, 27, 28], showing that deadline tardiness remains bounded as long as the system is not overloaded (i.e., DAG instances may miss deadlines, but are guaranteed to complete within an *a priori* fixed interval after their deadline). In contrast to Liu and Anderson's focus on establishing (non-tight) tardiness bounds, our goal is to determine as accurate as possible response-time bounds given (possibly) hard deadlines.

Serrano et al. [39] proposed an analysis for limited-preemptive DAG tasks. This is the closest work to our problem as it explicitly considers precedence constraints and limited-preemptive global scheduling at the same time. Our work improves upon this result by: **(i)** providing a much more accurate analysis for periodic DAGs and other types of tasks with regular, yet non-periodic release patterns, **(ii)** including all JLFP global scheduling policies in one uniform analysis framework, and **(iii)** supporting inter-task dependencies (rather than only precedence constraints within individual DAG tasks).

Several works have proposed exact analyses for global *preemptive* sporadic tasks *without* precedence constraints [4, 8, 9, 18, 43]. These analyses generally explore all system states that can possibly be reached using model checking, timed automata, or linear-hybrid automata. These solutions, however, are limited to the preemptive execution model and have limited scalability w.r.t. the number of tasks, processors, and the granularity of time. For instance, the analysis of Sun et al. [43] is reported to be limited to 7 tasks and 4 cores, and Guan et al.'s approach [18] is applicable only if task periods are integers in the range from 8 to 20.

In our own prior work [33], we considered the schedulablity analysis of a set of *independent* (i.e., non-DAG), non-preemptive sequential jobs scheduled with a global JLFP scheduling policy. While this paper superficially resembles [33] in that it uses a similar general approach – namely, the generation of a schedule-abstraction graph [32] – it actually follows a substantially different design needed to support limited-preemptive parallel DAG tasks. Specifically, in order to scale to non-trivial DAG tasks, the system state abstraction, exploration rules, and merge rules presented in this paper are entirely novel, and in fact even incomparable, to those previously used in [33]. Case in point, extensive experiments (see Sec. 5) revealed that the solution presented in this paper is up to two orders-of-magnitude faster than [33] when non-preemptive sequential tasks are analyzed, which reflects the nontrivial scalability advantages of the novel approach introduced in this paper.

3 System Model and Definitions

We consider the problem of globally scheduling a set of limited-preemptive parallel tasks with known arrival patterns upon a multiprocessor platform composed of m unit-speed processors. Each task is modeled by a DAG (V, E), where V is the set of execution segments, and E is the set of precedence constraints between execution segments in V. Each execution segment $v_j \in V$ has an execution time, and may (or may not) be assigned a relative release offset and relative deadline with respect to the arrival time of the task. For each arrival of a task, every execution segment in V releases a *job*. Even though we assume that tasks have known arrival patterns, we allow their execution segments, and hence their jobs, to be subject to release jitter. Similarly, the exact execution time of each job is *a priori* unknown. In addition, we allow precedence constraints to be specified among execution segments of different DAGs, thereby allowing for arbitrary inter-task precedence constraints.

As the arrival pattern of each task is known, our problem reduces to the analysis of a finite set of non-preemptive jobs $\mathcal{J}$ on an observation window whose length can be computed *a priori*. For periodic tasks with constrained deadlines, release jitter and synchronous releases, the observation window is equal to one hyperperiod (i.e., the least common multiple of all periods). Bounds on the observation window length for periodic tasks with release offsets, precedence constraints, and arbitrary deadlines were established by Goossens et al. [17].

Each job $J_i = ([r_i^{min}, r_i^{max}], [C_i^{min}, C_i^{max}], d_i, p_i, pred_i))$ released in the observation window has an *earliest-release time* r_i^{min}, a *latest-release time* r_i^{max}, a *best-case execution time* (BCET) C_i^{min}, a WCET C_i^{max}, an *absolute* deadline d_i, a priority p_i, and a set of predecessors $pred_i \subset \mathcal{J}$, i.e., a set of jobs that must complete before J_i may start executing. The set of successors of a job J_i is denoted by $succ_i = \{J_x \mid J_i \in pred_x\}$.

Each job is assigned a priority by a given job-level fixed-priority (JLFP) scheduling policy. We assume that a numerically smaller value of p_i implies higher priority. Any ties in priority are broken arbitrarily in a deterministic way. For ease of notation, we assume that the "$<$" operator implicitly reflects this tie-breaking rule. We assume a discrete-time model, i.e., all job timing parameters are integer multiples of a basic time unit such as a processor cycle.

At runtime, each job is *released* at an *a priori* unknown time $r_i \in [r_i^{min}, r_i^{max}]$. The release bounds r_i^{min} and r_i^{max} are computed based on the arrival pattern (e.g., periodic, multi-rate, or bursty) of J_i's task, its offset, and its release jitter. We also assume that each job J_i has an *a priori* unknown execution time requirement $C_i \in [C_i^{min}, C_i^{max}]$. We assume that a job's absolute deadline d_i is fixed and not affected by release jitter. We say that a job J_i is *possibly released* at time t if $t \geq r_i^{min}$, and *certainly released* if $t \geq r_i^{max}$.

Any two jobs that are neither directly nor indirectly predecessor/successor of each other are said to be independent. Independent jobs may execute in parallel. Each individual job must execute sequentially, i.e., it cannot execute on more than one core at a time and must run to completion once started. A job J_i that starts its execution on a core at time t occupies that core during the interval $[t, t + C_i)$. In this case, we say that job J_i *finishes by* time $t + C_i$. At time $t + C_i$, the core used by J_i becomes available to start executing other jobs. A job's *response time* is defined as the difference between the earliest-release time and the actual completion time of the job[1], i.e., $t + C_i - r_i^{min}$. We say that a job is *ready* at time t if it is released, did not start its execution before time t, and all of its predecessors have finished by time t. Further, we assume that the system does not have a job-discarding policy, i.e., released jobs remain pending until their execution is finished.

The paper assumes that shared resources that must be accessed in mutual exclusion are protected by FIFO spin locks. Since jobs execute non-preemptively, it is easy to obtain a bound on the worst-case time that any job spends spinning while waiting to acquire a contested lock [44]; we assume the worst-case spin delay is included in each job's WCET.

For ease of notation, we use $\max_0\{X\}$ and $\min_\infty\{X\}$ over a set of positive values $X \subseteq \mathbb{N}$ that is completed by 0 and ∞, respectively. That is, if $X = \emptyset$, then $\max_0\{X\} = 0$ and $\min_\infty\{X\} = \infty$. Otherwise they yield the usual maximum and minimum values in X.

We consider any non-preemptive global JLFP scheduler upon an identical multiprocessor platform. The scheduler is invoked whenever a job is released or completed. To simplify the presentation of the proposed analysis, we make the modeling assumption that, without loss of generality, at any invocation of the scheduling algorithm, at most one of the pending jobs is picked by the scheduler and assigned to a core. The scheduler is invoked once for each event if two or more release or completion events occur at the same time. The actual scheduler implementation in the analyzed system need not adhere to this restriction and may process more than one event during a single invocation. Our analysis remains safe if the assumption is relaxed in this manner.

In this paper, we exclusively focus on priority-driven and work-conserving scheduling algorithms, i.e., the scheduler dispatches a job only if the job has the highest priority among all ready jobs, and it does not leave a core idle if there exists a ready job. We assume that the WCET of each job is padded to cover all scheduling overheads and to account for any micro-architectural interference (e.g., competition for shared caches or memory bandwidth).

A job set $\mathcal{J}$ is *schedulable* under a given scheduling policy if no execution scenario of $\mathcal{J}$ results in a deadline miss, where an execution scenario is defined as follows [32].

▶ **Definition 1.** An *execution scenario* $\gamma = \{(r_1, C_1), (r_2, C_2), \ldots, (r_n, C_n)\}$, where $n = |\mathcal{J}|$, is an assignment of execution times and release times to the jobs of $\mathcal{J}$ such that, for each job J_i, $C_i \in [C_i^{min}, C_i^{max}]$ and $r_i \in [r_i^{min}, r_i^{max}]$.

4 Schedulability Analysis

The schedulability analysis proceeds by exploring the space of all possible schedules using the notion of a *schedule-abstraction graph* [32]. Each path in this graph reflects a sequence of job-dispatch decisions made by the underlying scheduling policy. As discussed in Sec. 2, a key innovation of this paper is a new system-state abstraction that more richly aggregates the necessary information in each state and, ultimately, reduces the number of edges in the final graph. After introducing the new abstraction (Sec. 4.2), we explain how to build the graph (Sec. 4.3), define exploration rules for work-conserving global JLFP scheduling policies (Sec. 4.4), describe how to soundly construct a new state (Sec. 4.5), and finally show how to merge similar states to reduce the size of the graph (Sec. 4.6). A proof of correctness of the analysis is presented in Sec. 4.7.

[1] Any release jitter is counted as part of the job's response time, as introduced by Audsley et al. [3].

4.1 Job Finish Times and System-Availability Intervals

Because jobs experience release jitter and execution time variation, exponentially many execution scenarios exist, and the exact finishing time of each job cannot be known a priori. Therefore, we compute an interval $[EFT_i, LFT_i]$ in which a job J_i will finish after a given sequence of scheduling decisions taken by the scheduler. This interval is lower-bounded by J_i's *earliest finish time* EFT_i and upper-bounded by its *latest finish time* LFT_i, that is, J_i may possibly finish at or after EFT_i and is certainly finished at LFT_i. A key challenge is that this uncertainty in job finish times introduces uncertainty in processor availability, which in turn affects the finish-time intervals of subsequently scheduled jobs.

To address this challenge, in our new abstraction, a state represents the state of the system after a possible sequence of scheduling decisions (corresponding to a subset of execution scenarios) by indicating when one, two, three, ..., m cores will *possibly* and *certainly* become available. Namely, each state includes a set of *system-availability intervals*, denoted $A = \{A_1, A_2, \ldots, A_m\}$, where $A_x = [A_x^{min}, A_x^{max}]$ means that x cores are *possibly available* (PA) starting at time A_x^{min} and *certainly available* (CA) no later than at time A_x^{max}.

▶ **Example 1.** Consider a system with $m = 3$ cores and suppose that three jobs are scheduled, with the following finish-time intervals: $[10, 45]$, $[30, 40]$, and $[15, 25]$. In this example, one core becomes possibly available at time 10. Two cores can possibly be available from time 15 onward. Similarly, one core becomes certainly available at time 25, and two cores become certainly available at time 40. Thus, $A_1 = [10, 25]$, $A_2 = [15, 40]$, and $A_3 = [30, 45]$.

4.2 Graph Definition

We define the schedule-abstraction graph as a directed-acyclic graph $G = (V, E)$, where V is a set of system states and E is the set of labeled edges. An edge $e \in E$ is defined as $e = (v_p, v_q, J_i)$, where v_p and v_q are the source and destination vertices of the edge, and the label J_i is the job that, by being scheduled, evolves state v_p to state v_q. We say job J_i is dispatched *next* after v_p or *succeeds* v_p if it is on an outgoing edge from a state v_p.

A path P from the initial state v_1 to a state v_p represents a possible sequence of job-dispatching events (or scheduling decisions) that lead to state v_p from the initial state v_1, which represents the initial idle system at time zero before any job is scheduled. The length of a path refers to the number of jobs scheduled on that path, i.e., $|P| \triangleq |\mathcal{J}^P|$, where $\mathcal{J}^P$ is the set of jobs that appear as labels on the edges of path P.

In graph G, it is possible to have more than one incoming edge to a vertex v_p. However, in that case, the following property must hold for any two paths that connect v_1 to v_p.

▶ **Property 1.** *For any two arbitrary paths P and Q that connect v_1 to v_p, $\mathcal{J}^P = \mathcal{J}^Q$.*

Having defined edges and paths, we next define a system state $v \in V$ as a three-tuple that contains: **(i)** the set of m system-availability intervals as defined in Sec. 4.1, denoted $A(v)$, **(ii)** a set $\mathcal{X}(v)$ of jobs that are *certainly executing* on the platform in state v, and **(iii)** a set of finish-time intervals $\{[EFT_x(v), LFT_x(v)] \mid J_x \in \mathcal{X}(v)\}$, where $EFT_x(v)$ and $LFT_x(v)$ represent the time at which job J_x is possibly and certainly finished considering the sequence of job-dispatch events that led to state v.

The motivation for including the set of certainly running jobs $\mathcal{X}(v)$ is that, given precedence constraints, the ready time of a job depends on the completion time of its predecessors. This creates a challenge as storing the EFT and LFT of *every* job on *every* path would require an exponentially increasing amount of memory w.r.t. the number of jobs scheduled. As a tradeoff, to improve the accuracy of the analysis, we maintain the set of

Algorithm 1: Schedule Graph Construction Algorithm.

```
Input  : Job set $\mathcal{J}$
Output : Schedule graph $G = (V, E)$
1  $\forall J_i \in \mathcal{J}, BR_i \leftarrow \infty, WR_i \leftarrow 0$;
2  Initialize $G$ by adding $v_1 = \big(\{[0,0], \ldots, [0,0]\}, \mathcal{X} = \emptyset, \emptyset\big)$;
3  while $\exists$ path $P$ from $v_1$ to a leaf vertex s.th. $|P| < |\mathcal{J}|$ do
4      $P \leftarrow$ the shortest path from $v_1$ to a leaf vertex $v_p$;
5      $\mathcal{R}^P \leftarrow$ set of ready jobs obtained with Eq. (1);
6      for each job $J_i \in \mathcal{R}^P$ do
7          if $J_i$ can be dispatched after $v_p$ according to Eq. (9) then
8              Build $v'_p$ using Algorithm 2;
9              $BR_i \leftarrow \min\{EFT_i(v'_p) - r_i^{min},\ BR_i\}$;
10             $WR_i \leftarrow \max\{LFT_i(v'_p) - r_i^{min},\ WR_i\}$;
11             Connect $v_p$ to $v'_p$ by an edge with label $J_i$;
12             while $\exists$ path $Q$ that ends to $v_q$ such that Rule 1 is satisfied for $v'_p$ and $v_q$ do
13                 Merge $v'_p$ and $v_q$ by updating $v'_p$ using Eq. (15);
14                 Redirect all incoming edges of $v_q$ to $v'_p$;
15                 Remove $v_q$ from $V$;
16             end
17         end
18     end
19 end
```

certainly running jobs $\mathcal{X}(v)$ and their finishing time intervals in each system state v. Since there are at most m such jobs per state, the amount of memory required per state remains constant. This property of the algorithm is discussed in detail in Sec. 4.4.

4.3 Graph-Generation Algorithm

We next introduce the main state-space exploration algorithm for finding the schedule-abstraction graph for a given workload and platform. We first provide an informal high-level overview, and then present the algorithm more precisely as pseudocode in Algorithm 1.

The schedule-abstraction graph is built iteratively in two alternating phases: *expansion* and *merging*. The expansion phase, expands (one of) the shortest path(s) P in the graph by considering all jobs that can possibly be dispatched *next* in the job-dispatch sequence represented by P. For each such job J_i, a new vertex v'_p is created and added to the graph via a directed edge from v_p to v'_p. The new state v'_p is generated from v_p by updating the core availability intervals and the set of certainly running jobs (and their finish-time intervals) when the execution of J_i is considered.

The merge phase slows down the growth of the graph by merging, whenever possible, the terminal vertices of paths that have the same set of dispatched jobs. As a key soundness condition, the merge phase guarantees that any possible execution scenario that can be generated from two un-merged states v_p and v_q can still be generated after they are merged.

The search ends when there is no vertex left to expand, that is, when all paths represent a valid schedule of all jobs in $\mathcal{J}$, which implies that all possible schedules have been explored.

Algorithm 1 presents our iterative breadth-first method for generating the schedule-abstraction graph in full detail. A set of variables keeping track of the smallest and largest response times (BR_i and WR_i, respectively) observed for each job in all execution scenarios explored so far is initialized in line 1; these bounds are updated whenever a job J_i can possibly be dispatched on a core (lines 9 and 10). The graph is initialized in line 2 with a root vertex v_1 that represents m idle cores at time 0. The expansion phase corresponds to lines 6–18 and lines 12–16 implement the merge phase. These phases repeat until every path in the graph contains $|\mathcal{J}|$ distinct jobs (line 3). We next discuss each phase in detail.

4.4 Expansion Phase

In this section, we explain how to expand a path P ending in v_p, as found in line 4 in Algorithm 1, by dispatching an eligible job after the scheduling sequence represented by P.

Overview

The expansion phase starts by obtaining the set of potentially *ready jobs* for system state v_p, i.e., jobs whose predecessors have been dispatched previously on path P.

For each ready job J_i, we calculate the earliest and latest time at which J_i can be dispatched on the platform after state v_p. These times are called the *earliest start time* (EST) and the *latest start time* (LST) of the job, denoted by $EST_i(v_p)$ and $LST_i(v_p)$, respectively.

If the earliest time at which the job can potentially start executing, i.e., $EST_i(v_p)$, is earlier than the latest time at which a work-conserving JLFP scheduler would allow that job to start if it is to be the next scheduled job, i.e., $LST_i(v_p)$, then the job is *eligible* to be dispatched after state v_p. For each eligible job, a new state v'_p is created and appended to path P after state v_p.

We next explain in detail, and precisely define, each step of the expansion phase.

Ready Interval

As stated in Sec. 3, a job is ready only if it is released and all of its predecessors have been completed. Thus, potentially ready jobs for path P are those that are not yet dispatched and all of their predecessors are in $\mathcal{J}^P$, i.e.,

$$\mathcal{R}^P \triangleq \{J_i \mid J_i \in \mathcal{J} \setminus \mathcal{J}^P \ \wedge \ pred(J_i) \subseteq \mathcal{J}^P\}. \tag{1}$$

Since each job J_i may suffer release jitter and because the exact finish times of J_i's predecessors are not known, the exact time at which J_i becomes ready is also unknown. For that reason, we compute a lower bound on the time at which a job $J_i \in \mathcal{R}^P$ is *possibly ready*, denoted R_i^{min}, and an upper bound on the time at which J_i is *certainly ready*, denoted R_i^{max}. Since a job can start its execution only if **(i)** it is released, and **(ii)** all its predecessors have completed, R_i^{min} is the minimum of r_i^{min} and the earliest time at which all predecessors of J_i have possibly completed, and R_i^{max} is the maximum of r_i^{max} and the time at which all predecessors of J_i have certainly completed, i.e.,

$$R_i^{min} \triangleq \max\left\{r_i^{min}, \max_0\{EFT^*_x(v_p) \mid J_x \in pred(J_i)\}\right\}, \text{ and} \tag{2}$$

$$R_i^{max} \triangleq \max\left\{r_i^{max}, \max_0\{LFT^*_x(v_p) \mid J_x \in pred(J_i)\}\right\}, \tag{3}$$

where $EFT^*_x(v_p)$ and $LFT^*_x(v_p)$ are safe bounds (defined next) on the earliest and latest finish time of J_x for all execution scenarios that lead to v_p. The use of $\max_0$ in Eqs. (2) and (3) ensures that the ready interval of jobs with no precedence constraint is equal to their release jitter interval, i.e., $R_i^{min} = r_i^{min}$ and $R_i^{max} = r_i^{max}$ if J_i does not have predecessors.

For the predecessors of J_i that are *certainly running* in system state v_p, i.e., any job $J_x \in \mathcal{X}(v_p) \cap pred(J_i)$, the bounds $EFT^*_x(v_p)$ and $LFT^*_x(v_p)$ can safely assume the values $EFT_x(v_p)$ and $LFT_x(v_p)$ saved in state v_p. However, for predecessors of J_i that are not certainly running in state v_p, i.e., any job J_x that is not in $\mathcal{X}(v_p)$, there is no bound on

$EFT_x(v_p)$ and $LFT_x(v_p)$ saved in v_p (which, to recall, is an intentional space optimization). Therefore, we instead use the current values of BR_x and WR_x (see Algorithm 1) as they are safe bounds on the EFT and LFT of J_x for all system states explored up to this point (lines 9 and 10 of Algorithm 1), which also includes v_p.

To summarize, if a job J_x belongs to $\mathcal{X}(v_p)$, then EFT^*_x and LFT^*_x are equal to $EFT_x(v_p)$ and $LFT_x(v_p)$, respectively. Otherwise, they are equal to BR_x and WR_x, respectively.

Earliest and Latest Start Times

Consider a job $J_i \in \mathcal{R}^P$, i.e., all the precedence constraints of J_i are respected. Job J_i cannot start executing prior to the earliest time at which it may become ready, i.e., R_i^{min}, nor can it start executing before the earliest time at which a core may become available, which is given by A_1^{min}. Thus, the earliest time at which J_i can start its execution after path P is given by

$$EST_i = \max\{R_i^{min}, A_1^{min}\}. \tag{4}$$

The latest start time of J_i after path P is decided by two factors: **(i)** the scheduler follows a JLFP scheduling policy, and **(ii)** the scheduler is work-conserving.

Considering factor (i), since a JLFP scheduling policy always dispatches the highest-priority ready job, the latest start time of J_i is upper-bounded by $t_{high} - 1$, where t_{high} is the earliest point in time from which on J_i certainly is not the highest-priority ready job anymore. An upper bound on t_{high} is given by Eq. (5) as proven in Lemma 2.

$$t_{high} \triangleq \min_{\infty}\{th_x(J_i) \mid J_x \in \mathcal{R}^P \ \wedge \ p_x < p_i\}, \text{ where} \tag{5}$$

$$th_x(J_i) \triangleq \max\left\{r_x^{max}, \max_0\{LFT^*_y(v_p) \mid J_y \in pred(J_x) \setminus pred(J_i)\}\right\}. \tag{6}$$

▶ **Lemma 2.** *Job J_i will not be the highest-priority ready job in $\mathcal{R}^P$ for system state v_p at any time later than $t_{high} - 1$.*

Proof. Suppose that $t_{high} \neq \infty$ (otherwise the claim is trivially true as it does not actually constrain J_i). Let $J_x \in \mathcal{R}^P$ be the job with higher priority than J_i such that $th_x(J_i) = t_{high}$.

At time $th_x(J_i)$, job J_x is certainly released (since according to Eq. (6), $th_x(J_i) \geq r_x^{max}$) and all predecessors of J_x that are not predecessors of J_i have been certainly completed (since $\forall J_y \in pred(J_x) \setminus pred(J_i)$, $th_x(J_i) \geq LFT^*_y(v_p)$ according to Eq. (6)). If $pred(J_x) \cap pred(J_i) = \emptyset$, then according to Eq. (3), J_x is certainly ready at $th_x(J_i)$ and J_i cannot be the highest-priority ready job from $th_x(J_i)$ onward.

If $pred(J_x) \cap pred(J_i) \neq \emptyset$, then, at the first point in time $t \geq th_x(J_i)$ such that all precedence constraints of J_i are respected, all precedence constraints of J_x are also respected (recall that the precedence constraints of J_x that are not common with J_i were already respected before or at time $th_x(J_i)$). In other words, if J_i becomes ready at or after $th_x(J_i)$ then J_x also becomes ready and J_i is not the highest-priority ready job. ◀

Additionally, considering factor (ii), if there is a time where a core is certainly available (which is the case from time A_1^{max} onward), and a job is certainly ready, a work-conserving scheduler must dispatch the job at that time, which is denoted t_{wc} and obtained as follows.

$$t_{wc} \triangleq \max\left\{A_1^{max}, \ \min_{\infty}\{R_x^{max} \mid J_x \in \mathcal{R}^P\}\right\} \tag{7}$$

▶ **Lemma 3.** *Job $J_i \in \mathcal{R}^P$ will not be dispatched next after v_p at any time later than t_{wc}.*

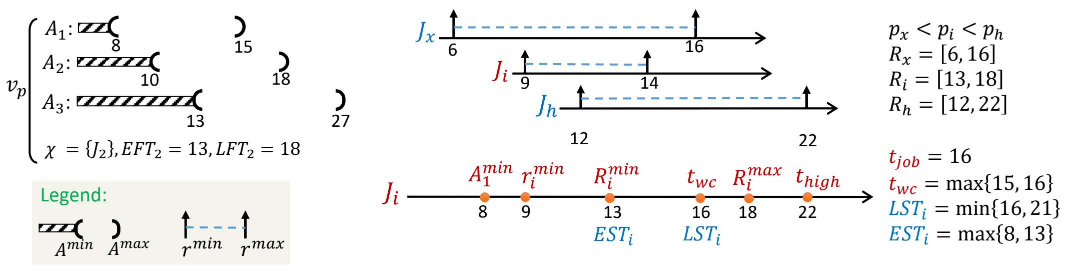


Figure 2 Calculating EST_i and LST_i for a successor job J_i of a certainly running job J_2.

Proof. Assume that $t_{wc} \neq \infty$; otherwise the claim is trivial. At time t_{wc}, a not-yet-dispatched job J_x whose precedence constraints are satisfied is certainly ready (because $t_{wc} \geq \min_\infty\{R_x^{max} \mid J_x \in \mathcal{R}^P\}$), and a core is certainly available (because $t_{wc} \geq A_1^{max}$). Hence, a work-conserving scheduler will dispatch J_x at t_{wc}. Consequently, J_i will be a direct successor of state v_p *only if* it starts no later than t_{wc}. ◀

Combining the facts that $LST_i \leq t_{high} - 1$ (Lemma 2) and $LST_i \leq t_{wc}$ (Lemma 3), we observe that J_i may be the next job scheduled after path P only if it starts no later than

$$LST_i = \min\{t_{wc},\ t_{high} - 1\}. \tag{8}$$

▶ **Example 4.** Figure 2 shows how EST_i and LST_i are calculated for J_i. The earliest time at which one core becomes ready is 8, and J_i is released at the earliest at time $r_i^{min} = 9$. However, since J_i must wait for its predecessor J_2 to finish before it becomes ready, we have $EST_i = 13$, which is the earliest finish time of J_2. Since J_x is certainly ready at time 16, and since at least one core is certainly available from time 15 onward, the latest time at which job J_i can be dispatched next after v_p is 16; otherwise, a work-conserving scheduler would schedule J_x after v_p instead. In this example, t_{high} is 22, where a higher priority job J_h is certainly released. However, since $t_{wc} < t_{high} - 1$, the LST_i is bounded by $t_{wc} = 16$.

Eligibility Condition

A job $J_i \in \mathcal{R}^P$ can be dispatched next after path P if its earliest start time EST_i is not later than its latest start time LST_i, i.e., if

$$EST_i \leq LST_i. \tag{9}$$

▶ **Lemma 5.** *Job J_i is a direct successor of v_p only if Inequality* (9) *holds.*

Proof. According to Lemmas 2 and 3, LST_i is an upper bound on the time at which J_i can be dispatched after v_p. Therefore, if J_i cannot be dispatched by LST_i, then it cannot be a direct successor of v_p. Since EST_i is the earliest time at which J_i can be dispatched after v_p, if $EST_i > LST_i$, J_i cannot be a direct successor of v_p. ◀

If a job J_i is dispatched next after v_p, its earliest and latest finish times are trivially

$$EFT_i = EST_i + C_i^{min} \text{ and} \tag{10}$$
$$LFT_i = LST_i + C_i^{max}. \tag{11}$$

Algorithm 2: Create a new state v'_p by dispatching job J_i after state v_p.

```
1 Initialize $PA$ and $CA$ using Eqs. (12) and (13);
2 for each $J_x \in \mathcal{X}(v_p) \cap pred(J_i)$ do
3     if $LST_i < LFT_x(v_p) \ \wedge \ LFT_x(v_p) \in CA$ then
4         replace $LFT_x(v_p)$ with $LST_i$ in $CA$;
5     end
6 end
7 Sort $PA$ and $CA$ in non-decreasing order;
8 $\forall x, 1 \leq x \leq m, \ A_x(v'_p) \leftarrow [PA_x, CA_x]$;
9 $\mathcal{X}(v'_p)$ is obtained from Eq. (14);
```

4.5 Creating a New State

If job $J_i \in \mathcal{R}^P$ satisfies Inequality (9), it can be dispatched next after v_p and a new system state v'_p is created to reflect this possibility. Algorithm 2 presents the procedure for creating a new state v'_p for job J_i. Line 1 creates two lists called PA and CA that contain bounds on the instants at which each core becomes possibly and certainly available after dispatching job J_i, respectively. Those lists are built using the following two lemmas.

▶ **Lemma 6.** *Lower bounds (respectively, upper bounds) on the instants at which each core becomes possibly (respectively, certainly) available after dispatching job J_i in system state v_p are given by PA (respectively, CA) defined as follows.*

$$PA \triangleq \left\{ \max\{EST_i, A_x^{min}(v_p)\} \mid 2 \leq x \leq m \right\} \cup \{EFT_i\} \tag{12}$$

$$CA \triangleq \left\{ \max\{EST_i, A_x^{max}(v_p)\} \mid 2 \leq x \leq m \right\} \cup \{LFT_i\} \tag{13}$$

Proof. We rely on the following four facts:

Fact 1. Since J_i is the first job starting to execute after system state v_p is reached, and because J_i's earliest start time is $EST_i(v_p)$, either all cores are busy until $EST_i(v_p)$, or no other job is released until $EST_i(v_p)$. In either case, after J_i is dispatched and the new system state v'_p is reached, none of the cores start executing another job before $EST_i(v_p)$. Therefore, for each core, its earliest and latest availability times for jobs other than J_i in the new state v'_p are no smaller than $EST_i(v_p)$.

Fact 2. At most x cores are available in the interval $[A_x^{min}(v_p), A_{x+1}^{min}(v_p))$ for $1 \leq x < m$, and no core is available for J_i to execute prior to $A_1^{min}(v_p)$ (by definition of $A_x^{min}(v_p)$). Therefore, each instant $A_x^{min}(v_p)$ is a lower bound on the availability time of a different core.

Fact 3. x cores are certainly available in the interval $[A_x^{max}(v_p), A_{x+1}^{max}(v_p))$ for $1 \leq x < m$, and all cores are certainly available after $A_m^{max}(v_p)$, by definition of $A_x^{max}(v_p)$. Each instant $A_x^{max}(v_p)$ is thus an upper bound on the availability time of a different core.

Fact 4. When J_i starts executing, it starts on the *first* available core (whichever physical core it is), and will occupy it until its finish time.

From Facts 1 and 2, the availability times of the cores in the new state v'_p are lower-bounded by $\{\max\{EST_i, A_x^{min}(v_p)\} \mid 1 \leq x \leq m\}$. Furthermore, from Facts 2 and 4, J_i starts its execution at the earliest at time $A_1^{min}(v_p)$ and keeps the core that was potentially available at $A_1^{min}(v_p)$ *certainly busy* until $EFT_i(v_p)$. Equivalently, that core will be possibly available at the earliest at $EFT_i(v_p)$ in the new system state v'_p. Therefore, the earliest times at which cores are potentially available in the new state v'_p are lower-bounded by $\{\max\{EST_i, A_x^{min}(v_p)\} \mid 2 \leq x \leq m\} \cup \{EFT_i\}$. This proves Eq. (12).

Similarly, from Facts 3 and 4, J_i starts executing on the first available core, which becomes certainly available at the latest at time A_1^{max}. J_i keeps that core *possibly busy* until $LFT_i(v_p)$, or equivalently said, the core that became available no later than A_1^{max} will be certainly available at $LFT_i(v_p)$ in the new system state v'_p. Therefore, considering Facts 1 and 3, the times at which cores are certainly available in the new state v'_p are upper-bounded by $\{\max\{EST_i, A_x^{max}(v_p)\} \mid 2 \leq x \leq m\} \cup \{LFT_i\}$. This proves Eq. (13). ◀

▶ **Lemma 7.** *If J_i is the job scheduled after v_p, all cores running predecessors of J_i in system state v_p become available by time $LST_i(v_p)$.*

Proof. As all predecessors of J_i must complete before J_i starts executing, and as the latest start time of J_i is $LST_i(v_p)$, all running predecessors of J_i must complete before time $LST_i(v_p)$. Hence, all cores running predecessors of J_i become available by time $LST_i(v_p)$. ◀

Line 1 in Algorithm 2 computes PA and CA according to Lemma 6. Lines 2–6 further ensure that the availability times of cores that are certainly executing predecessors of J_i are not larger than $LST_i(v_p)$, hence complying with Lemma 7.

Finally, Algorithm 2 computes the system-availability intervals for v'_p by sorting the lists PA and CA in non-decreasing order (lines 7–8). The correctness of this step is proven next.

▶ **Lemma 8.** *For any state v'_p built with Algorithm 1, let us define $t(v'_p)$ as follows: if $v_p = v_1$ then $t(v'_p) = 0$, otherwise $t(v'_p)$ is the EST of the last job dispatched to reach v'_p. For $1 \leq x \leq m$, x cores cannot be simultaneously available within $[t(v'_p),\ A_x^{min}(v'_p))$, and x cores are certainly available after time $A_x^{max}(v'_p)$.*

Proof. We prove the claim by structural induction on the states in the schedule-abstraction graph. The base case is state v_1, in which all cores are idle and, for $1 \leq x \leq m$, $A_x^{min}(v_1) = A_x^{max}(v_1) = 0$. The claim trivially holds as the interval $[t(v_1),\ A_x^{min}(v_1)) = [0, 0)$ is empty, and x cores are certainly available at time $A_x^{max}(v_1) = 0$, for all $1 \leq x \leq m$.

Next, in the inductive step, assume the claim holds for state v_p, that is x cores cannot be simultaneously available within $[t(v_p),\ A_x^{min}(v_p))$, and x cores are certainly available after time $A_x^{max}(v_p)$ for all $1 \leq x \leq m$. We prove that the claim holds in state v'_p resulting from dispatching J_i after v_p.

Assuming that $A_x^{min}(v_p)$ and $A_x^{max}(v_p)$ were safe bounds in state v_p (which holds by the induction hypothesis), Lemmas 6 and 7 prove that PA and CA provide safe lower bounds (resp., upper bounds) on the potential (resp., certain) availability times of each core in system state v'_p following the dispatch of job J_i, which happens no earlier than $t(v'_p) = EST_i$. Therefore, the x^{th} smallest element in PA is a lower bound on the time at which the x^{th} core may become available after $t(v'_p)$. Hence, the x^{th} smallest element in PA is also a lower bound on the time at which x cores may be simultaneously available after $t(v'_p)$. Since Algorithm 2 assigns the x^{th} smallest element in PA to $A_x^{min}(v'_p)$, the inductive claim holds for $A_x^{min}(v'_p)$. Similarly, the x^{th} smallest element in CA is an upper bound on the time at which an x^{th} core becomes certainly available in state v'_p. Hence, the x^{th} smallest element in CA is an upper bound on the time at which x cores are certainly available in v'_p. Since $A_x^{max}(v'_p)$ is assigned the x^{th} smallest element in CA, the inductive claim holds for $A_x^{max}(v'_p)$. ◀

Finally, Algorithm 2 updates the set of jobs that are certainly running in system state v'_p using the following property.

▶ **Property 2.** *If the earliest finish time of a running job $J_x \in \mathcal{X}(v_p)$ is later than J_i's latest start time, then J_x is still certainly running after J_i starts executing.*

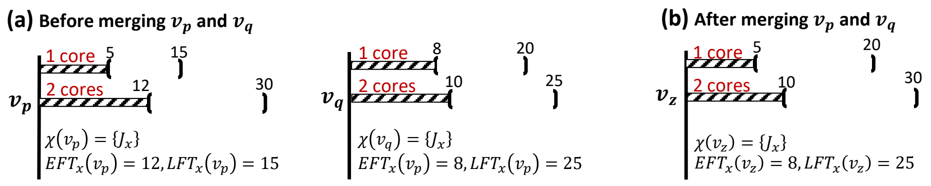

(a) Before merging v_p and v_q
1 core
5
15
2 cores
12
30
v_p
$\chi(v_p) = \{J_x\}$
$EFT_x(v_p) = 12, LFT_x(v_p) = 15$
1 core
8
20
2 cores
10
25
v_q
$\chi(v_q) = \{J_x\}$
$EFT_x(v_p) = 8, LFT_x(v_p) = 25$
(b) After merging v_p and v_q
1 core
5
20
2 cores
10
30
v_z
$\chi(v_z) = \{J_x\}$
$EFT_x(v_z) = 8, LFT_x(v_z) = 25$

Figure 3 States v_p and v_q **(a)** before and **(b)** after merging.

Therefore, certainly running jobs in state v_p' include J_i and all jobs that were running in state v_p and respect Property 2, i.e., $\{J_x \mid J_x \in \mathcal{X}(v_p) \ \wedge \ LST_i \leq EFT_x(v_p)\}$. Moreover, all predecessors of J_i must have been completed by LST_i. Hence, Eq. (14) below excludes the predecessors of J_i from the list of jobs that are certainly running in state v_p'.

$$\mathcal{X}(v_p') \leftarrow \{J_i\} \cup \{J_x \mid J_x \in \mathcal{X}(v_p) \setminus pred(J_i) \ \wedge \ LST_i \leq EFT_x(v_p)\} \tag{14}$$

4.6 Merge Phase

To slow down the growth of the graph (in terms of the number of system states generated), we merge paths with intersecting availability intervals that have the same set of jobs.

▶ **Rule 1** (Merge Rule). Two states v_p and v_q can be merged if $\mathcal{J}^P = \mathcal{J}^Q$ and $\forall x$, $1 \leq x \leq m$, $A_x(v_p) \cap A_x(v_q) \neq \emptyset$.

When two states v_p and v_q are merged, the system-availability intervals $A_x(v_z)$ in the merged state v_z are set to include the availability intervals of both v_p and v_q:

$$A_x(v_z) = \left[\min\{A_x^{min}(v_p), A_x^{min}(v_q)\}, \ \max\{A_x^{max}(v_p), A_x^{max}(v_q)\}\right]. \tag{15}$$

Eq. (15) expresses the fact that x cores become potentially available in the merged state v_z when x cores become potentially available in either of the original states v_p or v_q, and x core are certainly available in v_z when x cores are certainly available in both v_p and v_q.

Additionally, it is easy to see that the set of certainly running jobs in the merged state v_z comprises the jobs that were certainly running in both v_p and v_q, that is,

$$\mathcal{X}(v_z) = \{J_x \mid J_x \in \mathcal{X}(v_p) \cap \mathcal{X}(v_q)\}. \tag{16}$$

The finish time interval of each job J_x that is certainly running in v_z is updated to consider the bounds that were previously derived for all execution scenarios that lead to either v_p or v_q, and hence also to the merged state v_z. Therefore, we have that the EFT of J_x in v_z is the minimum between the EFTs in v_p and v_q. Similarly, the LST of J_x in v_z is the maximum LST reported for J_x in v_p and v_q, that is,

$$\begin{aligned} &EFT_x(v_z) = \min\{EFT_x(v_p), EFT_x(v_q)\} \text{ and} \\ &LFT_x(v_z) = \max\{LFT_x(v_p), LFT_x(v_q)\}. \end{aligned} \tag{17}$$

Figure 3 shows two states before and after merging. Lemma 9 proves that merging is safe.

▶ **Lemma 9.** *Merging two states v_p and v_q according to Rule 1 and Eqs.* (15), (16) *and* (17) *is* safe, *i.e., it does not remove any potentially reachable system state from the graph.*

Proof. First, Rule 1 enforces that the set of jobs scheduled on the path to v_p and v_q is identical for v_p and v_q. Therefore, the set of jobs that remain to be dispatched after v_z is the same as for v_p and v_q.

Second, removing jobs from the set of certainly running jobs $\mathcal{X}(\cdot)$ as done by Eq. (16), only increases the uncertainty in state v_z and therefore the set of system states reachable from v_z. Similarly, increasing the size of the possible finish intervals of the certainly running jobs (as done by Eq. (17)) increases the number of possible execution scenarios covered by v_z in comparison to v_p and v_q.

Finally, by Eq. (15) the system-availability intervals of the merged state v_z include the availability intervals of v_p and v_q. Therefore, all possible combinations of times at which a given number of cores is available either in state v_p or in state v_q are also possible in v_z. Thus, all sequences of dispatch events that are possible in v_p and v_q are possible in v_z and the system states reachable from v_z include all system states reachable from v_p and v_q. ◀

4.7 Correctness of the Proposed Solution

This section establishes the correctness of our analysis by showing that, for any possible execution scenario, there exists a path in the graph created by Algorithm 1 that represents the schedule of all jobs in the given scenario (i.e., Algorithm 1 is sound, but not exact).

▶ **Theorem 10.** *For any execution scenario such that a job $J_i \in \mathcal{J}$ finishes at some time t, there exists a path $P = \langle v_1, \ldots, v_p, v'_p \rangle$ in the schedule-abstraction graph generated by Algorithm 1 such that J_i is the label of the edge from the state v_p to the state v'_p and $t \in [EFT_i(v_p), LFT_i(v_p)]$.*

Proof. Initially, assume that the path $\langle v_1, \ldots, v_p \rangle$ respects the claim for all jobs dispatched before J_i in the execution scenario that led J_i to finish at time t. Furthermore, assume that **(i)** the availability intervals of state v_p correctly bound the availability time of x simultaneous cores in state v_p, $\forall x, 1 \leq x \leq m$, **(ii)** $\mathcal{X}(v_p)$ correctly includes a subset of the jobs that are certainly running on the platform before J_i is dispatched, and **(iii)** for each job $J_x \in \mathcal{X}(v_p)$, the interval $[EFT_x(v_p), LFT_x(v_p)]$ safely lower- and upper-bounds (i.e., contains) the completion time of J_x. We prove that there exists a vertex v'_p that is directly connected to v_p with an edge labeled J_i, that all three requirements (i)–(iii) hold for state v'_p, and that the interval $[EFT_i(v_p), LFT_i(v_p)]$ contains the completion time of J_i.

Under the assumption that hypotheses (i)–(iii) hold for v_p, Lemma 5 proves that Algorithm 1 expands the graph for any job that can possibly be dispatched next after v_p, hence also for J_i. Further, as proven in Sec. 4.4, Eq. (10) and Eq. (11) provide a lower and an upper bound on the completion time of J_i, respectively. Moreover, by Lemma 8, the availability intervals of v'_p correctly bound the simultaneous availability of x cores for all $1 \leq x \leq m$.

Eq. (14) computes the set $\mathcal{X}(v'_p)$ of certainly running jobs in state v'_p. Therefore, Requirement (ii) directly follows from Property 2 and the discussion of its role in obtaining Eq. (14) in Section 4.5. Requirement (iii) is the consequence of the assumption that the interval $[EFT_x(v_p), LFT_x(v_p)]$ computed for every job J_x dispatched before J_i in a state reached prior to v_p (and certainly running in v_p) is correct. Finally, according to Lemma 9, merging two states as in lines 12–16 of Algorithm 1 does not invalidate Requirements (i)-(iii).

Crucially, requirements (i)–(iii) are true for any state v_p' that is a direct successor of the initial system state v_1 because **(a)** in the initial state no job has been dispatched yet and all cores are available, and **(b)** Algorithm 1 initializes v_1's availability intervals to $[0, 0]$ (satisfying (i)), and sets the certainly running jobs set $\mathcal{X}(v_1)$ to $\emptyset$ (thus also satisfying (ii) and (iii)). The claim thus follows by induction on the states created by Algorithm 1. ◀

5 Empirical Evaluation

We conducted experiments to answer two main questions: **(i)** does the proposed test detect more schedulable workloads than state-of-the-art schedulability tests? And **(ii)** is the runtime of our analysis practical? We applied Algorithm 1 to the global limited-preemptive scheduling policy G-LP-FP. For the sake of simplicity, we used simple rate-monotonic priorities. As a baseline, we compared our results with the schedulability test of Serrano et al. [39] (identified as Serrano in the graphs) designed for *sporadic* limited-preemptive DAG tasks as it is the only available schedulability test in the state of the art for global limited-preemptive scheduling of DAG tasks based on the classical response-time analysis approach.

Since our test may also be used to analyze the special case of independent sequential non-preemptive tasks (NPR), we also performed experiments on such task sets, and compared our results to the test of Baruah for G-NP-EDF [5] (denoted by Baruah in the graphs), the test of Guan et al. for G-NP-FP [19] (denoted by Guan), the test of Lee for G-NP-FP [22] (denoted by Lee), and the test of Nasri et al. [33] for any G-NP-JLFP scheduling algorithm (denoted by Nasri18). Finally, we compare against the test of Yalcinkaya et al. [46] (denoted as Exact), an exact UPPAAL-based schedulability test for G-NP-FP (and EDF) that is designed for periodic tasks with fixed-preemption points and segmented self-suspensions.

We note that the Serrano, Baruah, Guan, and Lee tests are designed for sporadic DAG tasks; hence, we expect them to be pessimistic when applied to periodic workloads since sporadic tasks can generate more interference scenarios than periodic tasks. However, we believe that quantifying this pessimism serves to signify the need for schedulability tests that take task-activation patterns into account in order to provide more accurate results.

We implemented Algorithm 1 as a multi-threaded C++ program and performed the analysis on a cluster of machines each equipped with 256 GiB RAM and Intel Xeon E5-2667 v2 processors clocked at 3.3 GHz. We parallelized the breadth-first exploration of the schedule-abstraction graph using Intel's open-source Thread Building Blocks (TBB) library. Specifically, the while-loop in lines 3–19 can be easily parallelized since each iteration works on a different path. We report the *CPU time* as the runtime of the analysis, i.e., the total sum of the runtime of all threads used to analyze a task set. In the experiments, a task set was claimed unschedulable as soon as an execution scenario with a deadline miss was found.

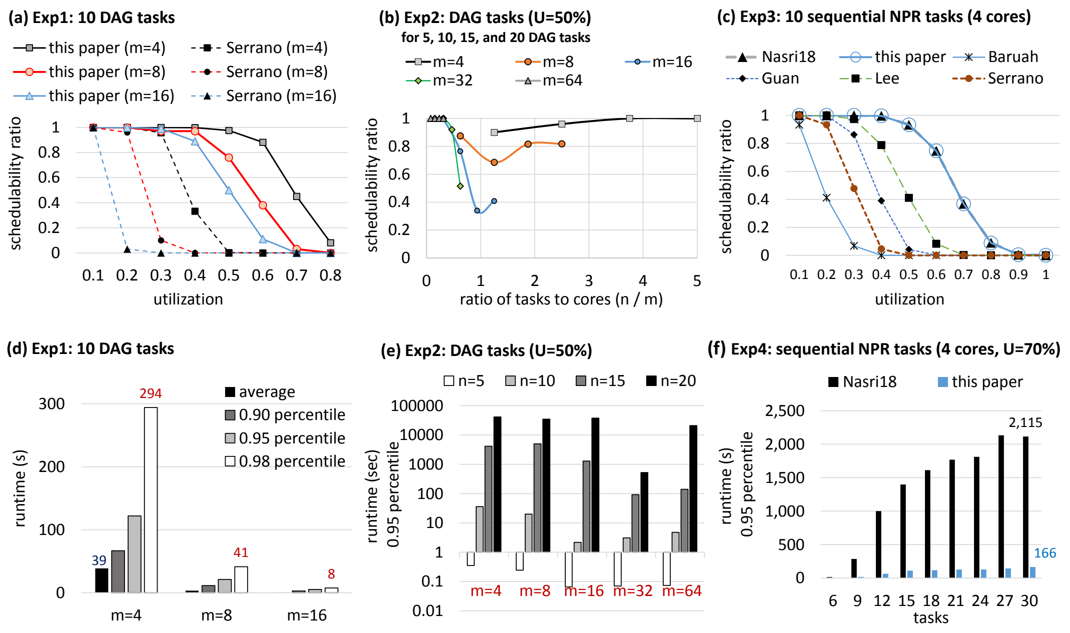
Experiments on synthetic task sets. We generated tasks using the same established techniques as used in prior studies [30, 35, 38, 39, 10]. The method generates series-parallel DAGs with nested fork-joins by recursively expanding blocks (a.k.a. non-terminal nodes) to either terminal nodes or parallel sub-graphs until a maximum depth of recursion (which limits the number of nested branches), a maximum length of the critical path, or a maximum number of nodes in the DAG is reached. The generation algorithm allows to define the branching factor, i.e., the maximum number of branches of parallel sub-graphs (denoted by n_{par}). In our experiment, the probability that a node is terminal, i.e., that it does not immediately fork a new branch, was set to $p_{term} = 0.4$, the probability of adding a random edge (precedence constraint) between two siblings was set to $p_{add} = 0.1$, the maximum number of nested branches was 3, the maximum number of nodes in the DAG was 50, and the maximum critical path length was set to 10 nodes. The WCET of each node was selected uniformly at random from the range [1, 50]. The BCET of each node was set to be 70% of the WCET.

To generate periodic DAG tasks with total utilization U, we used uUniFast [7] to generate random utilization values with a total sum U, and then assigned a period to each task using $\max\{\bar{C}_i/U_i, \bar{C}_i\}$, where $\bar{C}_i$ is the total sum of the WCET of all nodes of the task and U_i is the task utilization. To avoid cases where periods are co-prime and hence systems for which the hyperperiod is impractically large, we scaled the obtained periods so that they are contained in the interval $[500, 100000]$ with possible values given by the set $\{x \cdot 10^y \mid 1 \leq x \leq 9,\ 3 \leq y \leq 5\}$ms. This covers periods that are three orders of magnitude apart with a log-uniform distribution and includes periods commonly used by the automotive industry [21]. After assigning periods, we proportionally scale the WCET of the nodes so that tasks keep their intended utilization U_i (as assigned by uUniFast). We assumed that all tasks are first released at time 0 and that their deadlines are equal to their periods. Moreover, we assumed that all DAG nodes have an arrival time equal to the arrival time of the corresponding task. To filter out trivial task sets, we discarded task sets that cannot be successfully scheduled by G-LP-FP when each node of each task executes for its WCET. That is, we simulated the schedule of one hyperperiod using the WCET of each node and checked if there is a deadline miss (note that this initial test is only a necessary schedulability test, not a sufficient one because of the schedulability anomalies that exist under non-preemptive and limited preemptive scheduling).

The experiments were performed by varying **(Exp1)** the total system utilization U for 10 DAG tasks on 4 cores (Figures 4(a) and (d)), and **(Exp2)** the number of cores m and DAG tasks $n \in \{5, 10, 15, 20\}$ with $U = 0.5 \cdot m$ (Figures 4(b) and (e)). For each combination of parameters (e.g., DAG tasks with $U = 30\%, n = 10, m = 4$), more than 100 random task sets were generated. For each setup, we report the *schedulability ratio* (ratio of schedulable task sets to the number of task sets generated for that setup) and the *runtime* of Algorithm 1 for the task sets that were deemed *schedulable*. We excluded the runtime of unschedulable task sets since it would otherwise favor our solution and bias the results due to the fact that we stop the analysis as soon as a deadline miss is found. In other words, we only report the runtime of experiments that ran to the end, which is the worst case from an analysis runtime perspective. Since the runtime of the Serrano test never exceeded one second, it was omitted from all diagrams depicting runtimes.

Figure 4(a) shows a significant gap between the schedulability ratio determined by our solution and the baseline analysis for DAG tasks. For example, the Serrano test could only identify 10% of schedulable task sets for $U = 0.3$, while our test shows that at least 99% of them are schedulable. Furthermore, with the increase in the number of cores, the Serrano test becomes more pessimistic, e.g., it cannot find any schedulable task set with $U \geq 0.3$ when there are 16 cores, while the proposed test still finds schedulable task sets until $U = 0.6$.

Figure 4(b) shows the schedulability ratio as a function of the ratio between the number of DAG tasks and the number of cores (denoted by n/m). We observe that schedulability decreases when the number of tasks is close to the number of cores (i.e., the ratio n/m is around 1). We explain this trend by the fact that when there are more tasks than cores ($n/m > 1$), the per-task utilization and hence the blocking times caused by nodes of lower-priority tasks are smaller. As a result, the schedulability ratio is larger. This can be easily seen for $m = 4$ and $m = 8$ (since most values of n are larger than 4 or 8). The effect of smaller blocking times shows itself for $m = 16$, too, as an increase in the schedulability ratio for $n = 20$. When there are more cores than tasks ($n/m < 1$), there are enough cores to execute all tasks in parallel, hence the increase in schedulability. Further, more cores for a fixed number of tasks implies increased opportunity for tasks to execute their nodes in parallel; hence their response times approach their critical path lengths. This can be seen for larger values of m, e.g., 16 to 64. For instance the schedulability ratio for $m = 64$ is 100% for 10, 15 and 25 tasks, while it varies between 30% and 75% for $m = 16$.



■ **Figure 4** Experimental results for synthetic task sets for different experiments: **(a, d)** Exp1, **(b, e)** Exp2, **(c)** Exp3, and **(f)** Exp4. In (b), all task sets with $m = 64$ are schedulable. Hence, the curve overlaps with other curves (prior to the point $n/m < 0.4$).

Figures 4(d) and (e) show either a decrease or only a small linear increase in the runtime of the analysis w.r.t. to the increase in the number of cores in all experiments. Thanks to our new system state abstraction, the number of direct successors of a state does not depend on the number of cores in the system (unlike Nasri18 [33]) and hence the dependence on m is limited to the cost of calculating t_{high}, t_{wc}, etc. for each state.

For DAG tasks, with an increase in the number of tasks, the runtime of our analysis increases rapidly as the number of nodes and hence the number of jobs increases. While our analysis efficiently handles most task sets with up to 15 tasks within a couple of hours, it becomes notably slower for larger numbers of tasks. This, in particular, affects systems with a smaller number of cores, e.g., 4 and 8 cores, because when the system has insufficient cores to fully exploit the available task parallelism, the number of pending nodes in each system state increases. Since all nodes exhibit execution time variation, this drastically increases the number of possible scheduling decisions. As a result, the schedule abstraction graph grows rapidly since it must consider all possible interleavings.

In Figure 4(e) we observe a decrease in the runtime of the analysis from $m = 16$ to $m = 32$ and then an increase from $m = 32$ to $m = 64$. This decrease is due to the decrease in blocking times and an increase in the number of available cores (e.g., from 16 to 32 for 20 tasks). As a result, the busy windows become shorter, and hence paths merge very quickly as there are only relatively few interleavings to consider. On the other hand, the increase in the runtime of the analysis for $m = 64$ comes from the fact that, in a task set with 20 DAG tasks with $U = 50\%$, there are more tasks with large per-task utilizations. This situation increases the length of busy windows since tasks have only little slack. Moreover, due to the execution time variation of the tasks, there will be more scenarios that must be covered in the graph, which leads to an increase in the runtime of the analysis.

Experiments on non-preemptive sequential tasks (NPR). For the experiments on independent sequential non-preemptive tasks, we used the same task set generation setup as in [33]. To randomly generate a non-preemptive periodic task set with n tasks and a given utilization U, we used Emberson et al. [12] method to select the periods with a log-uniform distribution from the range [10000, 100000] microseconds with a granularity of $5000 \mu s$. We then used the RandFixSum [40] algorithm to generate n random task utilizations that sum to U. From the task utilization, we obtained C_i^{max} from the task utilization and the period and then set C_i^{min} to be 10% of C_i^{max}. Tasks were assumed to have implicit deadlines and any task set that had more than 100,000 jobs per hyperperiod was discarded from the experiment.

The experiments were performed by varying **(Exp3)** the total system utilization U ($n = 10$ and $m = 4$) for sequential non-preemptive tasks (Figure 4(c)), and **(Exp4)** the number of tasks n ($U = 70\%$ and $m = 4$) for sequential non-preemptive tasks (Figure 4(f)).

For sequential NPR tasks, as seen in Figure 4(c), our test performs similarly to Nasri18 (event though those tests are incomparable since task sets may be deemed schedulable by one and unschedulable by the other and vice versa). Both tests find many more schedulable task sets than the tests of Baruah, Guan, Lee, and Serrano. For example, for $U = 0.6$, our test and Nasri18 improve accuracy by 66 percentage points over the other baseline tests.

We have tried to run the exact test of Yalcinkaya et al. [46] on the data from Exp3. However, due to the scalability issue discussed in the introduction, the test could not complete the analysis of enough task sets to extract any meaningful results. Instead, we ran our analysis on the dataset used by Yalcinkaya et al. for sequential NPR task sets (see Exp2 in [46] for details). The results are depicted in Figure 1(a); the difference in accuracy between our test and the exact test is indistinguishable for the considered NPR task sets.

Figure 4(f) shows a neat improvement of our new analysis w.r.t. Nasri18, i.e., the best known analysis for G-NP-JLFP scheduling. For example, the 95th percentile runtime of Nasri18 [33] for 4 cores and 30 NPR tasks is more than 2,115 s while the 95th percentile runtime of the analysis presented in this paper is 166 s (i.e., a more than one order-of-magnitude difference). The maximum runtime of Nasri18 on all experiments that finished was 3027 s and one task set reached the time out of 1 h, while the maximum runtime of our new analysis was 275 s. The average runtime of the Nasri18 test was 327 s while our new analysis took an average of 25s only. These numbers strongly suggest that the proposed analysis is at least one order of magnitude faster than the Nasri18 test.

Experiments on benchmark task sets. We used the StreamIT benchmarks, which consist of a set of *digital signal processing* applications to evaluate the performance of our analysis on a realistic application workload. We used the DAG structure and WCET information of the tasks obtained by Rouxel et al. [36]. Table 1 reports the number of DAG nodes, width of the DAG graph (i.e., maximum number of parallel nodes), and the number of fork/join nodes. This table also presents the number of states, edges, and the runtime of the analysis for each of the benchmark applications when executed on a 4-core platform. As it can be seen, the analysis takes less than a minute even when there are more than 400 nodes in the DAG or when there are 80 fork/join constructs.

Discussion. Overall, we conclude that: **(i)** the proposed analysis is practical for realistic workload sizes and benchmarks, **(ii)** it has high accuracy when compared with the state-of-the-art exact schedulability analysis of sequential non-preemptive tasks with a global scheduling policy while being able to scale to much bigger systems (i.e., with more tasks

Table 1 Analysis of Benchmark Tasks.

Benchmark	#nodes	w	forks	**states**	**edges**	**runtime (ms)**
802.11a	119	7	17	10,164	28,656	**483.15**
Audiobeam	20	15	1	20	20	**0.18**
BeamFormer	56	12	2	6,036	29,686	**494.45**
CFAR	4	1	0	4	4	**0.05**
Complex-FIR	3	1	0	3	3	**0.04**
DCT2	40	16	2	40	40	**0.63**
DES	423	8	80	2,343	4,983	**849.63**
FFT2	26	2	1	74	122	**1.24**
FFT4	42	2	10	42	42	**0.27**
Filterbank	52	6	1	810,425	5,293,419	**25,339.02**
FMRadio	43	12	7	53,199	258,781	**1,402.83**

and more cores), **(iii)** it identifies a significantly larger portion of schedulable DAG tasks in comparison to the existing test, and **(iv)** the new system state abstraction allows a significant improvement in terms of scalability in comparison to the state-of-the-art test Nasri18.

However, even though the new abstraction allows scaling to much larger workloads, the treatment of execution time variation still needs further improvement. In the presence of precedence constraints, the impact of the response-time jitter of a job on its successors is the same as if the successors had a large release jitter. This induced jitter accumulates over chains of jobs with precedence constraints and greatly increases the degree of non-determinism in the graph exploration, and eventually forces the algorithm to consider all combinations of job orderings. This, for example, happens often in highly parallel DAG tasks or when the number of DAG tasks increases. Consequently, new techniques will have to be developed to allow the analysis to scale to highly parallel DAGs with large execution-time jitter.

6 Conclusion

We have considered the problem of analyzing the schedulability of a set of limited-preemptive DAG tasks with internal parallelism and precedence constraints scheduled upon a multicore platform using a global *job-level fixed-priority* (JLFP) scheduling policy. Our analysis conceptually enumerates all possible schedules using a novel system state abstraction that keeps track of the times at which a certain number of cores will become available. We have shown how the space of possible schedules can be explored with the abstraction, provided a proof of correctness, and conducted extensive experiments to assess the efficiency of the solution. Our analysis finds between 10 and 90 percentage points more schedulable task sets for most system configurations, in comparison with the best available baseline. It also scales to systems with up to 64 cores and 20 DAG tasks. A comparison with the state-of-the-art exact schedulability test for sequential non-preemptive tasks scheduled by a global JLFP scheduling policy has shown our analysis to scale much better while being almost as accurate as the exact test. The proposed analysis, however, does not yet scale to highly parallel DAG tasks or systems with a large number of cores (e.g., more than 64). In the future, we will investigate better ways of managing jitter, e.g., by applying partial-order reduction to skip over redundant paths that do not contribute to the worst-case response time of a task.

References

1 Jakaria Abdullah, Morteza Mohaqeqi, Gaoyang Dai, and Wang Yi. Schedulability Analysis and Software Synthesis for Graph-Based Task Models with Resource Sharing. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 261–270, 2018.

2 Ahmed Alhammad and Rodolfo Pellizzoni. Schedulability Analysis of Global Memory-predictable Scheduling. In *ACM International Conference on Embedded Software*, pages 20:1–20:10, 2014.

3 Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J. Wellings. Applying New scheduling Theory to Static Priority Preemptive Scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

4 Theodore P. Baker and Michele Cirinei. Brute-Force Determination of Multiprocessor Schedulability for Sets of Sporadic Hard-Deadline Tasks. In *International Conference on Principles of Distributed Systems (OPODIS)*, pages 62–75, 2007.

5 Sanjoy Baruah. The Non-preemptive Scheduling of Periodic Tasks upon Multiprocessors. *Real-Time Systems*, 32(1):9–20, 2006.

6 Sanjoy Baruah, Vincenzo Bonifaci, and Alberto Marchetti-Spaccamela. The Global EDF Scheduling of Systems of Conditional Sporadic DAG Tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 222–231, 2015.

7 Enrico Bini and Giorgio C. Buttazzo. Measuring the Performance of Schedulability Tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

8 Vincenzo Bonifaci and Alberto Marchetti-Spaccamela. Feasibility Analysis of Sporadic Real-time Multiprocessor Task Systems. In *ESA*, pages 230–241. Springer, 2010.

9 Artem Burmyakov, Enrico Bini, and Eduardo Tovar. An Exact Schedulability Test for Global FP Using State Space Pruning. In *International Conference on Real-Time Networks and Systems (RTNS)*, pages 225–234, 2015.

10 Daniel Casini, Alessandro Biondi, Geoffrey Nelissen, and Giorgio C. Buttazzo. Partitioned Fixed-Priority Scheduling of Parallel Tasks Without Preemptions. In *IEEE Real-Time Systems Symposium (RTSS)*, 2018.

11 UmaMaheswari Devi and James H. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. In *IEEE International Real-Time Systems Symposium (RTSS)*, pages 12–341, 2005.

12 Paul Emberson, Roger Stafford, and Robert I. Davis. Techniques For The Synthesis Of Multiprocessor Tasksets. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pages 6–11, 2010.

13 José Fonseca, Geoffrey Nelissen, and Vincent Nélis. Improved Response Time Analysis of Sporadic DAG Tasks for Global FP Scheduling. In *Proceedings of the 25th international conference on real-time networks and systems*, pages 28–37. ACM, 2017.

14 José Fonseca, Geoffrey Nelissen, and Vincent Nélis. Schedulability Analysis of DAG Tasks With Arbitrary Deadlines Under Global Fixed-Priority Scheduling. *Real-Time Systems*, 55(2):387–432, April 2019.

15 José Fonseca, Geoffrey Nelissen, Vincent Nelis, and Luís Miguel Pinho. Response Time Analysis of Sporadic DAG Tasks Under Partitioned Scheduling. In *11th IEEE Symposium on Industrial Embedded Systems (SIES)*. IEEE, 2016.

16 José Carlos Fonseca, Vincent Nélis, Gurulingesh Raravi, and Luís Miguel Pinho. A multi-DAG Model for Real-time Parallel Applications with Conditional Execution. In *Annual ACM Symposium on Applied Computing (SAC)*, pages 1925–1932, 2015.

17 Joël Goossens, Emmanuel Grolleau, and Liliana Cucu-Grosjean. Periodicity of real-time schedules for dependent periodic tasks on identical multiprocessor platforms. *Real-Time Syst.*, 52(6):808–832, 2016.

18 Nan Guan, Zonghua Gu, Qingxu Deng, Shuaihong Gao, and Ge Yu. Exact Schedulability Analysis for Static-Priority Global Multiprocessor Scheduling Using Model-Checking. In *Software Technologies for Embedded and Ubiquitous Systems (SEUS)*, pages 263–272, 2007.

19 Nan Guan, Wang Yi, Qingxu Deng, Zonghua Gu, and Ge Yu. Schedulability analysis for non-preemptive fixed-priority multiprocessor scheduling. *Journal of Systems Architecture*, 57(5):536–546, 2011.

20 Zhishan Guo, Ashikahmed Bhuiyan, Abusayeed Saifullah, Nan Guan, and Haoyi Xiong. Energy-Efficient Multi-Core Scheduling for Real-Time DAG Tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 22:1–22:21, 2017.

21 Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmark for free. In *International Workshop on Analysis Tools and Methodologies for Embedded Real-Time Systems (WATERS)*, 2015.

22 Jinkyu Lee. Improved Schedulability Analysis Using Carry-In Limitation for Non-Preemptive Fixed-Priority Multiprocessor Scheduling. *IEEE Transactions on Computers*, 66(10):1816–1823, 2017.

23 Jinkyu Lee and Kang G. Shin. Improvement of Real-Time Multi-CoreSchedulability with Forced Non-Preemption. *IEEE Transactions on Parallel and Distributed Systems*, 25(5):1233–1243, 2014.

24 Robert Leibinger. Software Architectures for Advanced Driver Assistance Systems (ADAS), 2015. URL: `https://people.mpi-sws.org/~bbb/events/ospert15/pdf/ospert15-talk-keynote.pdf`.

25 Cong Liu and James H Anderson. Supporting pipelines in soft real-time multiprocessor systems. In *21st Euromicro Conference on Real-Time Systems (ECRTS)*, pages 269–278. IEEE, 2009.

26 Cong Liu and James H Anderson. Scheduling suspendable, pipelined tasks with non-preemptive sections in soft real-time multiprocessor systems. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 23–32. IEEE, 2010.

27 Cong Liu and James H Anderson. Supporting soft real-time DAG-based systems on multiprocessors with no utilization loss. In *31st IEEE Real-Time Systems Symposium (RTSS)*, pages 3–13. IEEE, 2010.

28 Cong Liu and James H Anderson. Supporting graph-based real-time applications in distributed systems. In *17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, volume 1, pages 143–152. IEEE, 2011.

29 Cláudio Maia, Geoffrey Nelissen, Luis Nogueira, Luis Miguel Pinho, and Daniel Gracia Pérez. Schedulability analysis for global fixed-priority scheduling of the 3-phase task model. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10, 2017.

30 Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio C. Buttazzo. Response-Time Analysis of Conditional DAG Tasks in Multiprocessor Systems. In *Euromicro Conference on Real-Time Systems, (ECRTS)*, pages 211–221, 2015.

31 Morteza Mohaqeqi, Jakaria Abdullah, Nan Guan, and Wang Yi. Schedulability Analysis of Synchronous Digraph Real-Time Tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 176–186, 2016.

32 Mitra Nasri and Björn B. Brandenburg. An Exact and Sustainable Analysis of Non-Preemptive Scheduling. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 1–12, 2017.

33 Mitra Nasri, Geoffrey Nelissen, and Björn B. Brandenburg. A Response-Time Analysis for Non-Preemptive Job Sets under Global Scheduling. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 9:1–9:23, 2018.

34 Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A Predictable Execution Model for COTS-Based Embedded Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 269–279, 2011.

35 Bo Peng, Nathan Fisher, and Marko Bertogna. Explicit Preemption Placement for Real-Time Conditional Code. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 177–188, 2014.

36 Benjamin Rouxel and Isabelle Puaut. STR2RTS: Refactored streamit benchmarks into statically analysable parallel benchmarks for WCET estimation & real-time scheduling. In *OASIcs-OpenAccess Series in Informatics*, volume 57, pages 1:1–1:12, 2017.

37 Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill. Parallel Real-Time Scheduling of DAGs. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3242–3252, 2014.

38 Maria A. Serrano, Alessandra Melani, Marko Bertogna, and Eduardo Quiñones. Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions. In *Europe Conference on Design, Automation & Test & Exhibition (DATE)*, pages 1066–1071, 2016.

39 Maria A. Serrano, Alessandra Melani, Sebastian Kehr, Marko Bertogna, and Eduardo Quiñones. An Analysis of Lazy and Eager Limited Preemption Approaches under DAG-Based Global Fixed Priority Scheduling. In *IEEE International Symposium on Real-Time Distributed Computing (ISORC)*, pages 193–202, 2017.

40 Roger Stafford. Random vectors with fixed sum. Technical report, University of Oxford, 2006. URL: `http://www.mathworks.com/matlabcentral/fileexchange/9700`.

41 Martin Stigge and Wang Yi. Combinatorial Abstraction Refinement for Feasibility Analysis of Static Priorities. *Real-Time Systems*, 51(6):639–674, 2015.

42 Youcheng Sun and Marco Di Natale. Assessing the Pessimism of Current Multicore Global Fixed-Priority Schedulability Analysis. Research report, University of Oxford, 2017. URL: `https://hal.archives-ouvertes.fr/hal-01468067`.

43 Youcheng Sun and Giuseppe Lipari. A pre-order relation for exact schedulability test of sporadic tasks on multiprocessor Global Fixed-Priority scheduling. *Real-Time Systems Journal*, 52(3):323–355, 2016.

44 Alexander Wieder and Björn B. Brandenburg. On Spin Locks in AUTOSAR: Blocking Analysis of FIFO, Unordered, and Priority-Ordered Spin Locks. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 45–56, 2013.

45 Jun Xiao, Sebastian Altmeyer, and Andy Pimentel. Schedulability Analysis of Non-preemptive Real-time Scheduling for Multicore Processors with Shared Caches. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 199–208, 2017.

46 Beyazit Yalcinkaya, Mitra Nasri, and Björn B. Brandenburg. An Exact Schedulability Test for Non-Preemptive Self-Suspending Real-Time Tasks. In *IEEE/ACM Design, Automation and Test in Europe (DATE)*, pages 1222–1227, 2019.

Novel Methodologies for Predictable CPU-To-GPU Command Offloading

Roberto Cavicchioli
Università di Modena e Reggio Emilia, Italy
roberto.cavicchioli@unimore.it

Nicola Capodieci
Università di Modena e Reggio Emilia, Italy
nicola.capodieci@unimore.it

Marco Solieri
Università di Modena e Reggio Emilia, Italy
ms@xt3.it

Marko Bertogna
Università di Modena e Reggio Emilia, Italy
marko.bertogna@unimore.it

Abstract

There is an increasing industrial and academic interest towards a more predictable characterization of real-time tasks on high-performance heterogeneous embedded platforms, where a host system offloads parallel workloads to an integrated accelerator, such as General Purpose-Graphic Processing Units (GP-GPUs). In this paper, we analyze an important aspect that has not yet been considered in the real-time literature, and that may significantly affect real-time performance if not properly treated, i.e., the time spent by the CPU for submitting GP-GPU operations. We will show that the impact of CPU-to-GPU kernel submissions may be indeed relevant for typical real-time workloads, and that it should be properly factored in when deriving an integrated schedulability analysis for the considered platforms.

This is the case when an application is composed of many small and consecutive GPU compute/copy operations. While existing techniques mitigate this issue by batching kernel calls into a reduced number of *persistent kernel* invocations, in this work we present and evaluate three other approaches that are made possible by recently released versions of the NVIDIA CUDA GP-GPU API, and by Vulkan, a novel open standard GPU API that allows an improved control of GPU command submissions. We will show that this added control may significantly improve the application performance and predictability due to a substantial reduction in CPU-to-GPU driver interactions, making Vulkan an interesting candidate for becoming the state-of-the-art API for heterogeneous Real-Time systems.

Our findings are evaluated on a latest generation NVIDIA Jetson AGX Xavier embedded board, executing typical workloads involving Deep Neural Networks of parameterized complexity.

2012 ACM Subject Classification Computer systems organization → System on a chip; Computer systems organization → Real-time system architecture

Keywords and phrases Heterogeneous systems, GPU, CUDA, Vulkan

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.22

Supplement Material ECRTS 2019 Artifact Evaluation approved artifact available at
`https://dx.doi.org/10.4230/DARTS.5.1.4`

Funding Work supported by the CLASS Project, European Union's H2020 G.A. No. 780622.

1 Introduction

Modern high-performance embedded platforms feature heterogeneous systems, where a multicore CPU host is integrated with one or more parallel accelerators. These platforms are used to run cyber-physical real-time systems, requiring workload-intensive tasks to be executed within given deadlines. While there are many works addressing the schedulability analysis of real-time tasks on multi-core systems [4], there is an increasing interest in understanding and refining the adopted task models to better capture the timing behavior of real-time workloads in practical scheduling settings on heterogeneous embedded platforms. Thanks to the dramatic improvement of performance-per-Watt figures over multicore CPUs, GP-GPU (General Purpose GPU) computing is a widely adopted programming model to perform embarrassingly parallel computations in both embedded and discrete devices. Typical usage scenarios of heterogeneous embedded platforms are found in the domain of autonomous driving, avionics and industrial robotics, presenting important design challenges due to the safety-critical nature of these domains [12, 25].

To address this challenge, we first briefly summarize how computations are orchestrated in an embedded platform that features a multicore CPU (host) and an integrated GPU accelerator (device).

1.1 CPU-to-GPU interaction

The interactions between CPU-threads and the GPU are described through different APIs (Application Programming Interfaces) that allow the host to control the offloading of data and workload description, to be later executed by the parallel processing units within the GPU. A way to abstract a higher-level description of the processing units of a GPU is to think of it as being composed of two engines: the Execution Engine (EE) for computation, and the Copy Engine (CE) for high throughput DMA data transfers.

The Execution Engine enables a high level of SIMD parallelism by exploiting hundreds of ALU pipelines, that are usually grouped into processing clusters. Within each of these processing clusters, a hardware scheduler dispatches small groups of threads in lockstep to the ALU pipelines. A well-known API for modelling such interactions is CUDA [24], a proprietary API developed by NVIDIA. In such a model, ALU pipelines are called *CUDA cores*, and processing clusters are known as *Streaming Multiprocessors (SMs)*. SMs schedule *warps*, i.e. groups of 32 lockstep threads. Considering different GP-GPU APIs (such as OpenCL), the terminology may slightly vary, but these higher-level concepts remain unchanged. On the host side, the application developer has the responsibility to trigger data movements and the workload to be executed to the Execution Engine of the GPU by using API function calls. This operation is known as *kernel invocation*. When many workloads have to be dispatched within short time intervals, the time spent on the CPU for these offloading operations becomes non-negligible, and it should be properly accounted for when pursuing a sound schedulability analysis.

A task may be modeled using a Directed Acyclic Graph (DAG), where nodes represent workloads executing on a CPU core or on a GPU engine, while edges are precedence constraints between nodes. In a typical application, CPU nodes are constantly interleaved with GPU nodes. Despite its intrinsic parallelism, the GPU is considered as a single resource. This is due to its lockstep nature that limits task-level parallelism, especially for embedded GPUs with a smaller number of SMs. In other words, a GPU node represents a kernel occupying all the processing units of the considered GPU engine.

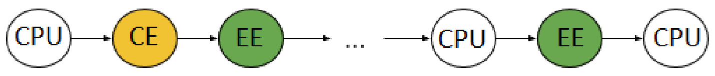


Figure 1 Sequences of sub-tasks of a CPU core submitting work to both compute and copy GPU engines. White nodes are CPU offloading operations, yellow and green nodes represent GPU copy and compute operations.

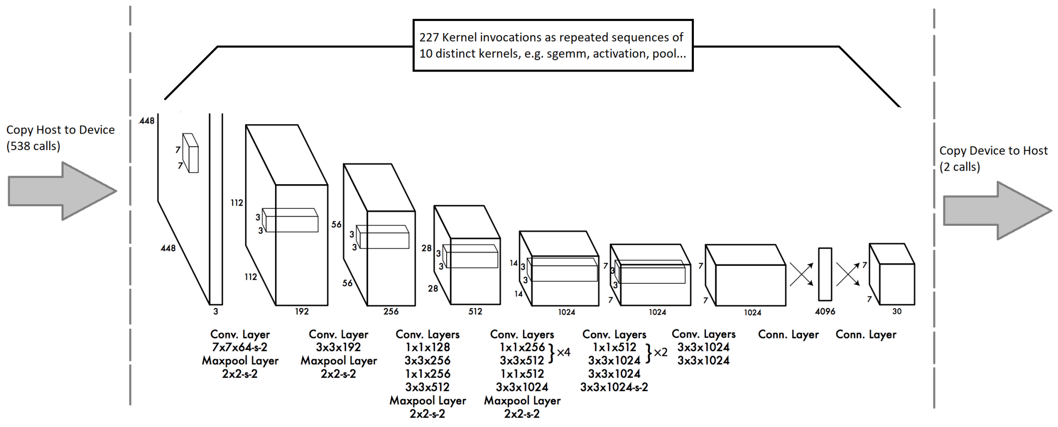

227 Kernel invocations as repeated sequences of 10 distinct kernels, e.g. sgemm, activation, pool...
Copy Host to Device (538 calls)
Copy Device to Host (2 calls)
448
7
7
448
3
112
3
3
112
192
56
3
3
56
256
28
3
3
28
512
14
3
3
14
1024
7
3
3
7
1024
7
7
1024
4096
7
7
30
Conv. Layer 7x7x64-s-2 Maxpool Layer 2x2-s-2
Conv. Layer 3x3x192 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x128 3x3x256 1x1x256 3x3x512 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x256 3x3x512 ×4 1x1x512 3x3x1024 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x512 3x3x1024 ×2 3x3x1024 3x3x1024-s-2
Conv. Layers 3x3x1024 3x3x1024
Conn. Layer
Conn. Layer

Figure 2 Data transfers and Layer-to-kernel invocations over the YOLO network [26].

Moreover, CPU nodes have to account for a variable cost of GPU commands' submission time, related to the translation of API function calls and to trigger the GPU operations, independently of this operation being synchronous (i.e., the CPU shall wait for the offloaded GPU kernel to complete before continuing) or asynchronous (i.e., the CPU may proceed to the next node without needing to wait for the offloaded kernel to complete).

The example depicted in Figure 1 shows a submission-intensive workload, in which a variable number of CPU-nodes offload-operations interleave among different kernels and copy invocations. Even if the represented chain may appear simplistic w.r.t. a fully-fledged DAG, it is not far from real world applications. In Figure 2, we profile CPU and GPU traces of a single iteration of the inference operation on a widely used YOLOv2 neural network [27]. The inference over YOLO's 28 network layers requires heavy data movements and the invocation of a very large number of kernels, each preceded by a CPU offloading phase. Using *nvprof*, the NVIDIA CUDA profiler, we discovered that kernels and copies are invoked as a sequence of implicitly synchronized operations.

There are two negative effects of interleaving CPU offloads and GPU workloads:

1. the completion of a submission node is a mandatory requirement for the activation of the subsequent GPU node. This implies that jitters or delays in the completion of the CPU node will postpone the actual release of the GPU sub-task, increasing the complexity of the schedulability analysis.
2. In case of shorter GPU kernels, CPU offloading becomes a performance bottleneck.

As a consequence of these considerations, we aim at analyzing and minimizing the time spent by the CPU for submitting commands related to GPU operations.

In the literature, these issues have been addressed using *persistent kernels* [15], i.e. a reduced number of kernel invocations that delegates the responsibilities of future workload invocations to GPU threads. We hereafter discuss the limitations of CUDA persistent threads, showing how novel features of the latest CUDA SDK release allows the programmer to mitigate the problem of submission-intensive operations. We characterize novel submission methods such as CUDA Dynamic Parallelism (CDP) and pre-constructed CUDA Graphs, and investigate a novel API for GPU programming called *Vulkan*. Vulkan is a next generation *bare-metal* API allowing a dramatic improvement in the control that the application programmer has over the interaction between the application layer and the GPU driver. Aspects that were commonly well hidden by traditional APIs are completely exposed to the application programmer: fine grained synchronization, low level mechanisms of memory allocation and coherency, state tracking and commands validation are all essential parts in direct control of the application developer. According to the Khronos Group, the industrial consortium that released and maintains the Vulkan technical specification [21], this increased level of control enhances applications performance and predictability due to substantially reduced CPU-to-GPU driver interaction. We will prove that this is indeed the case, especially for submission intensive workloads, making Vulkan a very promising open standard for real-time systems executing on heterogeneous platforms.

We experimentally validate our considerations executing representative kernels on an NVIDIA Jetson AGX Xavier[1], a recently released embedded development platform, featuring a Tegra System on Chip (Xavier SoC) composed of 8 NVIDIA Carmel CPU Cores (superscalar architecture compatible with ARMv8.2 ISA) and an integrated GPU based on the NVIDIA Volta architecture with 512 CUDA cores. On this platform, we executed a parametrized version of a Deep Neural Network, varying the number and position of convolutional and fully-connected layers.

The remainder of the paper is structured as follows. Section 2 presents a recent overview on the challenges of achieving timing predictability on GPU-accelerated SoCs, with specific emphasis on the impact of submission procedures. In Section 3, we describe the relevant CUDA features that have been introduced in the latest SDK versions, and how we leveraged them in our experiments. Section 4 provides a brief explanation of the Vulkan API peculiarities and related programming model, while Section 5 introduces our open source Vulkan Compute wrapper, that we implemented to simplify the generation (and easy reproduction for artifact evaluation) of the experiments. Experimental settings and related discussion on the results are provided in Sections 6 and 7. Section 8 concludes the paper with further remarks and proposals for future work.

2 Related Work

The recent literature on GPU-based real-time systems identified multiple sources of unpredictability, related to GPU scheduling [17], CPU interactions with the GPU API/driver [13, 14] and memory contention [28, 2, 9]. In this paper, we focus on minimizing driver interactions by exploiting recently proposed instruments to pre-record and pre-validate the GPU commands that characterize CPU-to-GPU offloading operations. Previously, a way to minimize CPU offloading times was to batch compute kernel calls into a single or reduced number of command submissions. This approach based on *persistent kernel* proved to have beneficial effects in terms of average performance [5, 15, 6, 16] and for obtaining a higher degree of control when scheduling blocks of GPU threads [31, 10]. The problem with persistent kernels is that they often require a substantial code rewriting effort and they are not able to properly manage Copy Engine operations mixed with computing kernels. Moreover, a persistent

[1] `https://developer.nvidia.com/embedded/buy/jetson-agx-xavier-devkit`

kernel approach would require managing GPU kernels with different safety criticality levels within a single application context, posing a significant threat to the overall system safety integrity assessment.

Our work tries to overcome these aspects by exploiting more recent methodologies, such as CUDA Dynamic Parallelism (CDP) and CUDA graphs. CDP allows the programmer to minimize kernel launch latencies in a similar way as a persistent kernel, without the need of substantial code rewriting and still allowing the different GPU tasks to reside in different process address spaces. CDP has been introduced in the CUDA SDK since version 5.0 (released in 2014).

More recently, the latest release of the CUDA SDK (version 10, released in late 2018) introduced the concept of CUDA graphs: a CUDA graph allows the developer to construct a graph of copy, compute and even host sub-tasks together with their related dependencies in advance. Then, at runtime, the offloading operation only involves a single call to the CUDA runtime driver. As pointed out in the introductory section, we model CPU and GPU interactions as sequences of operations that can be generalized to a DAG: this is similar to the OpenVX programming interface [32].

To the best of our knowledge, we are exploring for the first time not only these novel CUDA functionalities, but also the real-time capabilities of the Vulkan API, a recently released open standard for GPU programming. The work on novel low level and low overhead GPU APIs started in late 2013, with Apple's *Metal* and AMD's *Mantle* APIs representing the first concrete efforts toward the implementation of such a novel bare-metal approach. While the first one is a proprietary effort limited to Apple devices, Mantle development stopped around mid 2015, but served to the Khronos group as a base to start developing the Vulkan specifications (Version 1.0 was released in February 2016). In parallel, Microsoft's *Direct3D 12* saw its release at the same time as Windows 10. All these APIs share a common programming model that imposes minimal driver overhead as a result of a more explicit and direct control over GPU related commands.

The problem of minimizing CPU-GPU driver overhead is an issue that the Khronos group considered also for newer *OpenGL* releases. The so-called *AZDO* methodology (Approaching Zero Driver Overhead) aimed at minimizing the driver interaction by batching draw calls, state changes, reduced texture bindings and data factorization into command lists to be submitted within a reduced number of OpenGL API calls. As we already highlighted in the previous section, we are more interested on reducing submission latencies to improve the predictability of real-time applications. Hence, our work will focus on measuring how effective is the compute pipeline of Vulkan compared to traditional and to recently released CUDA-based submission models. We believe this document may be an instructive reference for real-time practitioners looking for more predictable settings and programming models on modern heterogeneous embedded platforms.

3 Alternative submission models for CUDA

In this section we provide an in-depth explanation of the approaches we tested for minimizing CPU-GPU driver interactions with relation to NVIDIA CUDA. These mechanisms are made possible by recent releases of the CUDA SDK, and they describe the motivation and typical usage scenarios of CDP and CUDA graphs.

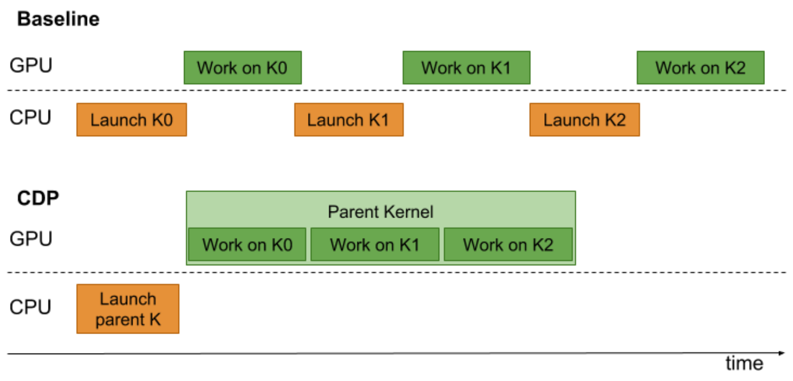


Figure 3 Timelines of the baseline CUDA submission model compared to CDP. Note that K0, K1 and K2 implementations are identical in both cases and the launch configuration of Ki depends on the result of $K(i-1)$.

3.1 CDP

GPU kernel invocations are characterized by a *launch configuration.* A launch configuration describes the degree of parallelism of the GPU computation. Implementation-wise, this implies that the programmer has to decide the size and number of CUDA blocks for the specific launch configuration. A CUDA block is a collection of concurrent threads that share the same L1 cache and/or scratchpad memory inside a single SM. The size of a block is the actual number of threads within each block. Trivially, if a GPU task is an algorithm in which each step presents a different parallelization degree, then the programmer has to break down this task into multiple kernel launches, each with a potentially different launch configuration. As discussed in the previous section, this can easily translate in additional overhead on the CPU side for performing offloading operations, especially when launch configurations between different launches are not known a priori.

Nested parallelism is a known construct in the parallel programming literature that allows the programmer to express variable parallelism as a series of nested kernel invocations with adaptive launch configurations. Nested parallelism has therefore the potential to fully exploit the GPU parallelism for workloads involving variable depth recursion (e.g: sorting [23], graph exploration [34], clustering algorithms [1] ...). In CUDA, this is implemented by having a parent kernel invoking child kernels with a varying block/thread count, without involving the CPU for launching the child kernels.

An example scenario is depicted in Figure 3, where a baseline offloading (top) is compared against a CDP sequence of invocations (bottom). In the depicted corner case, the CPU submission time is comparable to the GPU execution time for the considered kernels. In the baseline scenario, there is an interleaved work between host and GPU, where the CPU thread has to adjust the launch configuration for each subsequent kernel. A similar situation happens even for asynchronous submissions. Conversely, with the CPD scenario the CPU is only active during the launch of the parent kernel, while subsequent kernel invocations are managed by the device, allowing the concurrent execution of GPU workloads [3, 24]. This drastically reduces the response time of the entire task. Another interesting advantage is related to the simplification of the related worst-case response-time analysis, as the task graph to consider has significantly less nodes (CPU sub-tasks) and edges (CPU-GPU interactions). However, we will see in the evaluation section that CDP presents some limitations when data is requested by the CPU through the Copy Engine and when the sequence of kernels is not characterized by variable parallelism.

3.2 CUDA Graphs

CUDA graphs are the most recent innovation to the CUDA runtime set of functions. Graphs are a step forward compared to the more traditional CUDA streams: a stream in CUDA is a queue of copy and compute commands. Within a stream, enqueued operations are implicitly synchronized by the GPU so to execute them in the same order as they are placed into the stream by the programmer. Streams allow for asynchronous compute and copy, meaning that the CPU cores dispatch commands without waiting for their GPU-side completion: even in asynchronous submissions, little to no control is left to the programmer with respect to when commands are inserted/fetched to/from the stream and then dispatched to the GPU engines, with these operations potentially overlapping in time.

Graphs improve on this approach by allowing the programmer to construct a graph of compute, host and copy operations with arbitrary intra- and inter-stream synchronization, to then dispatch the previously described operations within a single CPU runtime function. Dispatching a CUDA graph can be an iterative or periodic operation, so to implement GPU-CPU tasksets as periodic DAGs. This aspect represents an appealing mechanism for the real-time system engineer. Legacy CUDA applications built on streams can be converted to CUDA graphs by capturing pre-existing stream operations, as it is shown in Listing 1.

In Listing 1 a baseline implementation of asynchronous offloading of copy, compute and CPU-side operations is shown (lines from 1 to 7): while the CPU is still inserting operations within the stream, the GPU fetches previously inserted commands on behalf of the GPU engines. This mechanism improves the average performance for long GPU-side workloads, but it is difficult to precisely model this CPU-GPU interaction for scheduling purposes. Moreover, the CPU can act as a bottleneck in case of a long sequences of small GPU operations. Also, in case of dispatching periodic work, CPU-GPU interaction will be repeated and the timing cost for each command submission and validation is bound to increase.

Lines from 9 to 15 show how to capture the same stream operations of the baseline approach to construct a CUDA graph. Lines from 18 to 32 show a graph construction that is equivalent to the previous methodology, but nodes and their dependencies are explicitly instantiated instead of being inferred from pre-existing stream operations. When building a graph, no operations are submitted to the GPU. This allows the developer to describe in advance complex work pipelines. Work is then dispatched with a single or periodic call to *cudaGraphLaunch* (lines 50 and 58). In the experimental section of this paper, we will show how submission latencies and CPU-GPU interaction timeline vary by exploiting these novel CUDA constructs.

4 The Vulkan API

Although Vulkan is defined as a graphics and compute API, the Vulkan model is agnostic to which of these two pipelines will be mostly used within an application. Initial benchmarks on the Vulkan API are related to graphics applications, hence measuring the performance of the same application with a Vulkan renderer and over an OpenGL/OpenGLES or Direct 3D 11 renderer [29]. One of the recent fields of applications that has been shown to provide sensible benefits from this new API paradigm is Virtual Reality (VR), due to the very stringent latency requirements required to mitigate motion/cyber sickness for users wearing VR-enabled devices [30]. We instead investigate the possibility of exploiting Vulkan to minimize driver interactions for allowing a more predictable response time of computing kernels. To the best of our knowledge, we are the first to propose the adoption of this newer generation API for real-time systems. The following overview of the Vulkan API from the programmer point of view can be followed on Figure 4.

Listing 1 CUDA baseline streams – node graphs and stream capture.

```
void baseline(cudaStream_t &s){
 cudaMemcpyAsync(....,s);
 ...
 cudaStreamAddCallback(s,..,cpuFunction);
 kernelCall<<<...,s>>>(...);
...
}

cudaGraph_t cudaGraphFromStreamCapture(&s){
 cudaGraph_t graph;
 cudaStreamBeginCapture(s);
  baseline(&s); //no modifications to baseline
 cudaStreamEndCapture(s,&graph);
 //stream operations are not yet submitted.
}

cudaGraph_t cudaGraphCreation(){
cudaGraph_t graph;
//node pointers
cudaGraphNode_t *memcpyNodes, *kernelNodes, *hostFuncNodes;
//node params
cudaKernelNodeParams *memcpyParams, *kerNodeParams,
   *hostNodeParams;
//define params for all the previously declared nodes
//(addesses, function names etc...)
cudaGraphCreate(&graph,0);
//for each host/device node and respective params ...
cudaGraphAdd<Memcpy/Kernel/Host>Node
(&nodePtrs, graph, ..., ..., &nodeParams);
//first param: node ptrs, second and third: depedencies
//info, forth: node params.
//No stream operations are submitted to the GPU
}

void mainPrepareAndOffload(){
 cudaStream_t s, sGraph;
 cudaStreamCreate(&s); //regular stream
 cudaStreamCreate(&sGraph); //stream for graph launch
 cudaGraphExec_t graphExec; //graph execution data structure

 //enqueue in stream s and launch with baseline behaviour:
 baseline(&s);
 //wait for finish
 //ALTERNATIVE METHOD 1:
 //create and define graph from pre-existing stream
 cudaGraph_t graph0 = cudaGraphFromStreamCapture(&s);
 //graph instantiation
 cudaGraphInstantiate(&graphExec, graph0, ....);
 //and launch
 cudaGraphLaunch(graphExec, sGraph);
 //wait for finish
 //ALTERNATIVE METHOD 2:
 //create and define graph from node structures
 cudaGraph_t graph1 = cudaGraphCreation();
 //graph instantiation
 cudaGraphInstantiate(&graphExec, graph1, ....);
 //and launch
 cudaGraphLaunch(graphExec, sGraph);
 // ...
}
```

The improved control over the thin driver layer exploited by a Vulkan application comes at the cost of a significantly increased implementation complexity. While in the CUDA runtime API establishing a context occurs transparently when the first CUDA API call is performed, a Vulkan context involves the explicit creation of a *vkInstance* object[2]. Such object stores global states information at application level. A *vkInstance* is created by default with no added debug/validation layers: in order to minimize the impact of driver level runtime checks and hidden optimizations (which are always performed in traditional APIs, such as CUDA), the application programmer has to explicitly add/activate third-party API layers at *vkInstance* creation time.

The layer-based mechanism is also one of the novel features enabled by these next generation APIs. Once the instance is created, the programmer retrieves a handle to the physical device(s) to use: from a single physical device (*vkPhysicalDevice* object), one or more logical devices (as software constructs able to interface with the chosen physical device) are created. From the logical device, memory for a *vkCommandPool* is allocated: a command pool in Vulkan is used to amortize the cost of resource creation across multiple command buffers. Finally, the programmer has to create one or more command buffers (*vkCommandBuffer*), which are objects that allow storing commands to be later sent to the GPU via a *vkQueue*. A *vkQueue* is the Vulkan abstraction of a queue of commands, with each set of created queues belonging to specialized family of capabilities (compute, graphics, data transfer etc...). A *vkQueue* is roughly the equivalent of a CUDA stream, but with a much finer granularity in terms of CPU-GPU and intra/inter queue synchronizations.

Compute kernels (more specifically, *Vulkan compute shaders*) are stored in Khronos's SPIR-V (Standard Portable Intermediate Representation [19]) binary cross-compatible format and compiled at runtime into device specific code (*vkShaderModule*). Then, the programmer has to explicitly describe how resources are bound to the compute shader, i.e. how input, output and constant buffers are mapped to the shader arguments. Vulkan *descriptor sets* are allocated within *descriptor pools* and further specified thorough *descriptor layouts*. This allows the developer to efficiently re-use the same bindings in different compute shaders.

For what refers to kernel arguments, constants are specified and updated through *Vulkan push and specialization constants*. Associating constants and descriptor layouts to one or more compute shaders occurs at pipeline creation time. During this phase, different pipeline stages are defined to allow associating different *vkShaderModule*s to different pre-allocated descriptor sets. The pipelines are then "baked" into static Pipeline State Objects (PSOs) for later use within a command buffer recording phase. Recording a command buffer allows the programmer to switch different pre-constructed PSOs, so to set in advance a plurality of compute kernels, each with its own descriptor set(s)/layout(s). Recorded commands are then executed when calling the *vkQueueSubmit* function. Once such function is called, little to no other driver interaction is necessary.

This long sequence of operations happens behind a context creation and subsequent kernel invocations in a Vulkan application, and it gives a very plausible overview of the lower-level details that are instead transparently managed in CUDA applications. The hidden handling of these aspects in the closed CUDA implementation makes it less suitable for predictable settings. Instead, we argue that *the explicit handling allowed by Vulkan may make this API much more suitable for real-time systems*, especially considering that aspects like bindings, sequence of kernel invocations and launch parameters are known in advance for predictable real-time applications. The Vulkan model is thus a perfect fit in this setting, since all these parameters are taken into account in pre-compiled structures such as PSOs.

[2] We use the term *object* for brevity, but this has nothing to do with object oriented programming: being the Vulkan client driver implemented in C, such objects are actually non dispatchable handles to C *structs*.

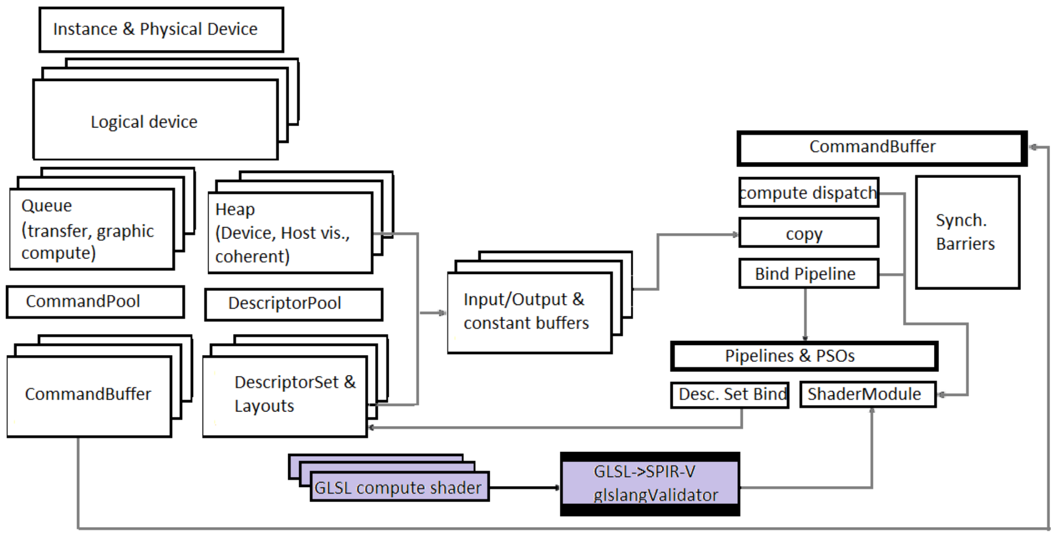


Figure 4 Schematic representation of the operations and modeling artifacts involved in a simplified Vulkan Compute pipeline. Shaded elements are external to the Khronos Vulkan specifications.

5 VkComp: an open-source wrapper for the Vulkan predictable compute pipeline

One of the strongest point of the CUDA model is its ease of use, which, as mentioned, is substantially better than Vulkan. We therefore decided to create a C++ wrapper to the Vulkan API that easily allows setting up operations such as context creation, buffer allocations, memory transfers and kernel invocations, to make these operations as easy as they were using the CUDA runtime API. Our wrapper allows the developer to instantiate pipeline creation blocks and command buffer recording blocks. These functionalities (roughly similar to CUDA graphs), are responsible for creating in advance reusable resources such as PSOs and pre-recorded command buffers. We plan to release this wrapper as an open-source contribution.

5.1 Wrapper structure

Our C++ wrapper to the Vulkan API exposes the functionalities described by the interface class represented in Figure 5. The figure shows the member functions are going to be called within a pipeline creation block, as opposed to command buffer recording blocks and application initialization.

In our experiments, we only use a single command buffer enqueued in a single queue. A generic application composed of many kernel invocations and multiple buffers can be summarized as shown in Pseudo-Algorithm 1.

In the pipeline creation block, a compute kernel is specified together with its launch configuration (set as specialization constants) and descriptor layouts. Argument buffers are pointers to SSBOs (Shader Storage Buffer Objects), whereas constants are specified as Vulkan *push constants*. Recording a command buffer allows us to order the sequence of kernel invocations (by selecting previously baked PSOs), data movements between CPU-GPU (any direction) and constant values updates.

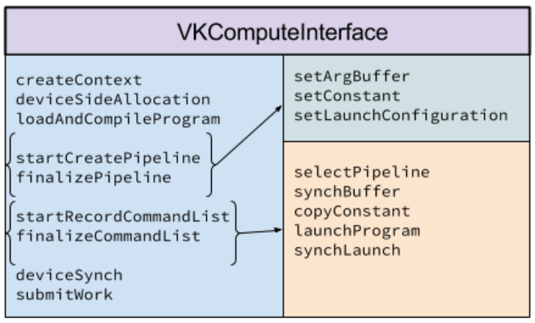


Figure 5 Vulkan based compute interface describing the exposed functionalities of our model.

Algorithm 1 Pseudocode of our VkComp Vulkan wrapper example application.

```
{application initialization:}
$createContext$
for each buffer do
    buffer ← $deviceSideAllocation$
end for
{pipeline creation block}
for each kernel do
    program ← $loadAndCompileProgram$
    startCreatePipeline(program)
        $setArgBuffer(buffer...)$ for each related buffer
        $setConstant$ for each related constant
        $setLaunchConfiguration$
    PSO ← finalizePipeline
end for
{Record command list block:}
startRecordCommandList
    for each PSO do
        $selectPipeline(PSO)$ for each related PSO
        $synchBuffer(buffer)$ for each related buffer
        $copyConstant$ for each related constant
        $launchProgram(kernel)$
    end for
cmdbuf ← finalizeCommandList
$submitWork(cmdbuf)$
$deviceSynch$
```

5.2 Buffers and programs

At context creation, a persistently mapped staging buffer is allocated. Its allocation size is large enough to contain all the data used by the application. This persistently mapped buffer is a driver-owned CPU-side buffer in which we can store buffer data to be later sent to the GPU. This allows the programmer to avoid mapping and unmapping buffers from CPU-only visible addresses to GPU-visible memory areas. An allocation table takes care of sub-allocation segmentation. In our interface, this is transparent to the application designer.

A compute kernel (called *program* in Figure 5 and Algorithm 1) is written in GLSL[3] [20] (OpenGL Shading Language) and translated into SPIR-V using the *glslangValidator* program that is provided by Valve's LunarG Vulkan SDK[4]. Then, a *vkShaderModule* (see section 5) is created and bound to a pipeline.

6 Experimental setting

In this section, we validate the proposed considerations on a representative hardware platform. We adopted an NVIDIA Jetson AGX Xavier development board, featuring a host processor with 8 NVIDIA Carmel cores, an ARM-based superscalar architecture with aggressive out-of-order execution and dynamic code optimizations. Carmel Instruction Set Architecture (ISA) is compatible with ARMv8.2. The Carmel CPU complex presents 4 islands of 2 cores each: each core has a dedicated L1, while each island presents a shared L2 Cache (2 MiB). Moreover, a L3 cache (4 MiB) is contended by all the CPU cores. In this SoC, the main accelerator is represented by an integrated GPU designed with the NVIDIA Volta architecture, featuring 512 CUDA cores distributed in 8 SMs. Each SM has a 128 KiB L1 cache, and a shared L2 Cache (512 KiB) is shared among all the SMs. Other application specific accelerators are present in this SoC, such as the DLA (Deep Learning Accelerator) and the PVA (Programmable Vision Accelerator): further discussions about these additional accelerators are beyond the scope of this paper.

The software stack used in our experiments is mostly contained within the latest version of the NVIDIA Jetpack (4.1.1). It contains the L4T (Linux For Tegra) Operating System (version 31.1), which is an NVIDIA custom version of Ubuntu, featuring a linux kernel version 4.9. The CUDA SDK version is 10, whereas the Vulkan driver is updated to version 1.1.76. Lunarg SDK for glslangValidator is updated to version 1.1.97.0.

6.1 Experimental use case

The application we selected for our tests relates to one of the most peculiar workloads to be accelerated on a GPU: Deep Neural Networks (DNNs). Inference of DNNs is an extremely common use case for heterogeneous SoCs. For example, latest Advanced Driving Assitance Systems (ADAS) and Autonomous Driving (AD) prototypes make a heavy use of neural networks for the detection/classification of objects on the road, making inferencing latencies a critical aspect of the system.

Analyzing the layers of a DNN, breaking them down into algorithmic steps, each layer presents algebraic computations that are typical of other generic signal processing tasks: basic algebraic operations such as matrix/matrix and matrix/vector multiplications, convolution filters and element-wise linear and/or non-linear functions are perfect example of compute building blocks for a wide variety of embarrassingly parallel computations.

For these reasons, we believe that parameterizing neural network topologies provides a synthetic benchmark that is realistic and general enough for our proposed approaches. More specifically, we characterize our neural network as a sequence of kernel invocations, with each invocation corresponding to a different layer. We therefore parameterize both the length of this sequence of kernels and the size of the input data, as shown in Figure 6.

[3] Our Vulkan compute wrapper allows the user to write compute shaders in GLSL. This facilitates porting from CUDA or OpenCL pre-existing kernels.
[4] `https://www.lunarg.com/vulkan-sdk/`

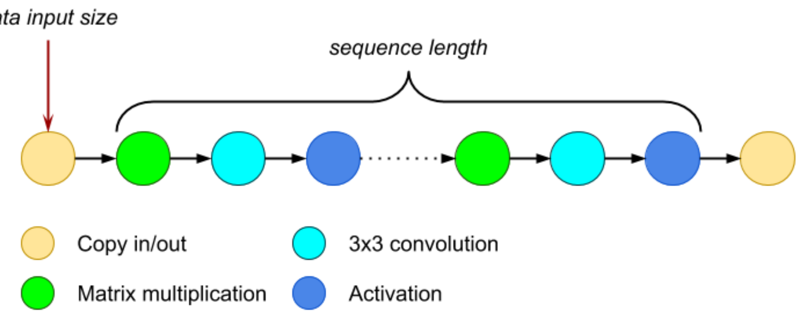


Figure 6 Schematic description of the application used for benchmarking the different submission models. Matrix multiplications are used to compute fully connected layers, as we assume dense matrices. The 3x3 convolution represents a convolutional layer. Simple activation functions (RELU and sigmoid) conclude the sequence.

In the DAG in Figure 6, copy-in operations are implemented as Copy Engine operations in which input layers, DNN weights and convolutional filters are made visible to the GPU (CUDA memcpy host to device). The copy-out operations only relates to the output buffer (CUDA memcpy device to host). Input and output buffers for each layer are switched between subsequent kernel calls. We also alternate RELU and sigmoid activation functions between subsequent iterations. For statistical analysis, the entire sequence of operations is periodically repeated. Input size matrices for both input and output buffers are $(k * block) \times (k * block)$, with $k \in [1, 2, 4, 8]$ and $block = 16$ and the total sequence length of kernel launches is given by the number of kernel launches for a single iteration (three) multiplied by the length of the sequence: $3 \times l, l \in [1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000]$.

Trivially, by increasing the length of the sequence of kernel launches, the CPU is forced to increase its interactions with the GPU driver. By adjusting the data size, it is possible to change the ratio between CPU and GPU compute latencies. In our experiments, the range in which we vary the kernel iterations is compliant with the layer count of known neural networks, as deployed in real world applications: YOLO [26] is composed of 26 convolutional layers and 2 fully connected layers, whereas tiny-YOLO has 9 layers in total. Google's MobileNet [18] also features 28 layers, whereas larger networks can feature up to hundreds of layers (e.g. ImageNet [22], R-FCN [11] and latest YOLO versions).

For Vulkan, compute shader implementations are derived from CUDA device kernel code, that is then converted to GLSL compute shader with a line-by-line exact translation. Operating this translation is easy, as only the name of the built-in variables used for thread/block indexing differs between the two languages.

The Volta GPU architecture features application specific circuitry (i.e. *tensor cores*) for further acceleration of matrix-multiply-accumulate operations. However, we are not aware of any Vulkan/GLSL support for tensor core operations. To allow a fair comparison, also our CUDA kernel implementation uses only regular CUDA cores.

All the experiments have been performed with the highest supported frequency and voltage for the whole SoC with no dynamic frequency and voltage scaling (i.e. `nvpmodel -m 0 && jetson_clocks.sh`). Processes are pinned to one CPU core and set to FIFO99 priority. Display servers/compositors or any other GPU application other than our benchmarks are not permitted to execute. Code for our experiments can be found at `https://git.hipert.unimore.it/rcavicchioli/cpu_gpu_submission`.

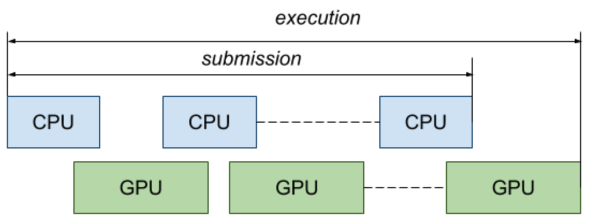


Figure 7 Simplified visualization of the submission and execution metrics evaluated in our experiments. Depending on the submission model, both the CPU and GPU tasks might be represented as a variable number of small blocks.

7 Results

We assume that context set up, memory allocations and kernel compiling procedures are operations performed only once, during the whole system initialization phase. Therefore, we did not characterize these phases, as they do not impact the runtime latencies of recurring real-time workloads. For CUDA graphs and Vulkan, we considered graph creation, command buffer recording and pipeline creation as part of the initialization phase. CUDA graph is created by capturing the same memory and kernel operations enqueued in the baseline stream.

For this experimental setting, we measured two response times: submission completion time (*submission*) and total execution time (*execution*). The submission time is the time spent on the CPU submitting thread between the asynchronous submission of a task to the time in which the control is given back to the CPU submitting thread. In the baseline scenario, this translates in measuring the time taken for enqueuing all the necessary operations in the CUDA stream. For CDP, this latency is only observed for submitting the parent kernel launch, whereas for CUDA graphs we only measure the latency of the *cudaGraphLaunch* call. Similarly, for Vulkan we only measure the submission of the pre-recorded command buffer (*VkQueueSubmit* function call, equivalent to *submitWork* in our VkComp wrapper). The total execution time is the time taken for completing the submitted job. As shown in Figure 7, this latter measure includes both the CPU submission work and the GPU sequence of copy and compute kernel operations.

The results are shown in Figure 8 and 9 (pp. 15, 16) and are quite surprising (please note the logarithmic scale of the plots). In the leftmost column, where the submission time is shown, a drastic reduction can be noticed for Vulkan compared to all the CUDA alternatives. With Vulkan, the worst-case time needed to regain control of the CPU submitting thread can be *orders of magnitude smaller* than in the CUDA-based approaches. More specifically, comparing Vulkan with CDP (the best performing among the CUDA methodologies), the improvement ranges from 4× to 11×.

Even more interestingly from a predictability perspective, while increasing the sequence length causes significant variations in submission times for CUDA baseline and graphs, such variations in Vulkan are almost negligible. Figure 10a shows the measured jitter for different sequence lengths, i.e. the difference between recorded maximum and minimum submission times.

When comparing the three CUDA methodologies, we see a consistent behaviour throughout all the tested configurations and data block sizes. As expected, the CUDA baseline performance are lower then the other two methodologies when considering both average and worst cases.

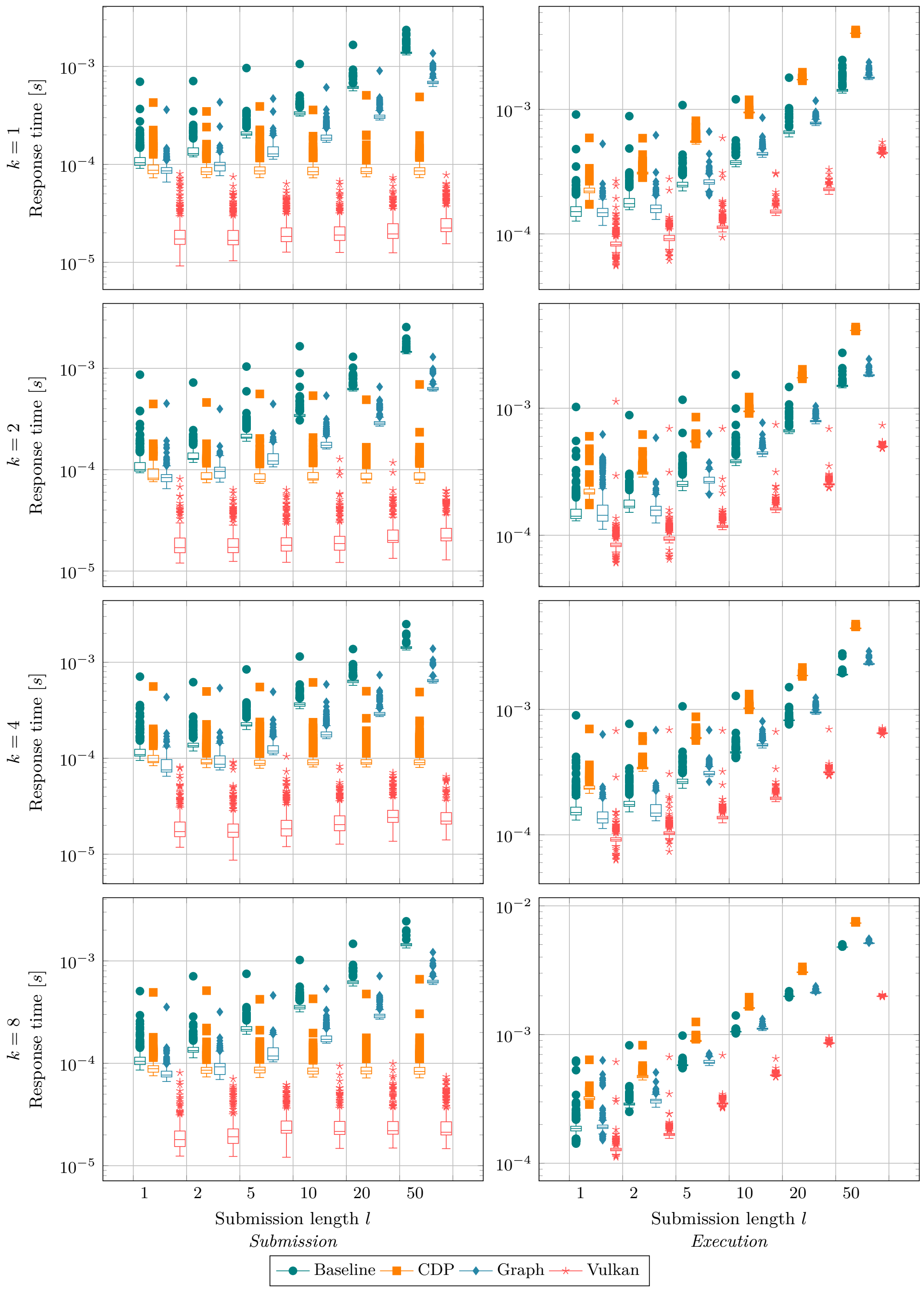


Figure 8 Results of response time for submission (left) and execution (right) for $l \in [1, 2, 5, 10, 20, 50]$, $block = 16$ and $k \in [1, 2, 4, 8]$. Y-scale is logarithmic and different between left and right column.

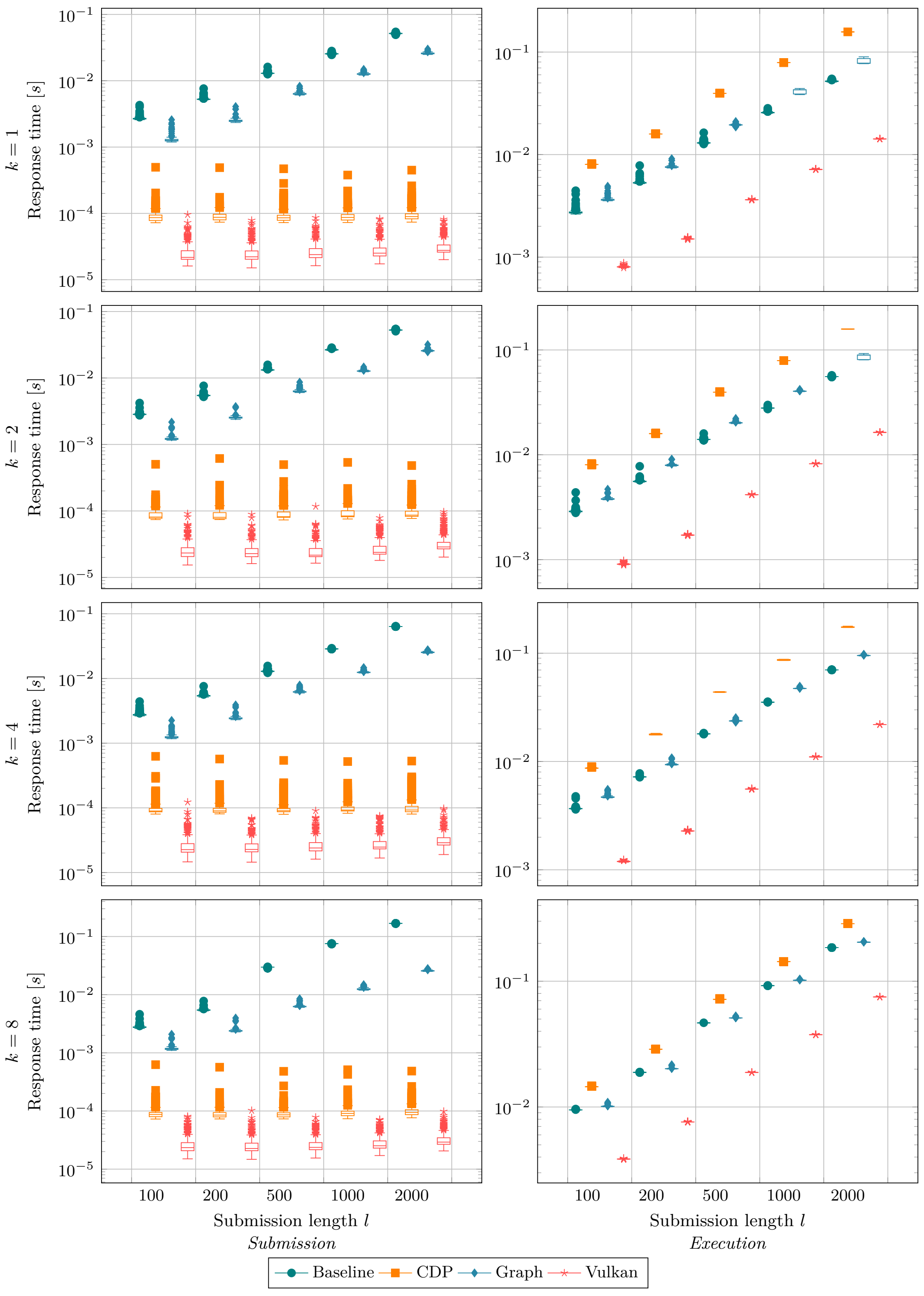
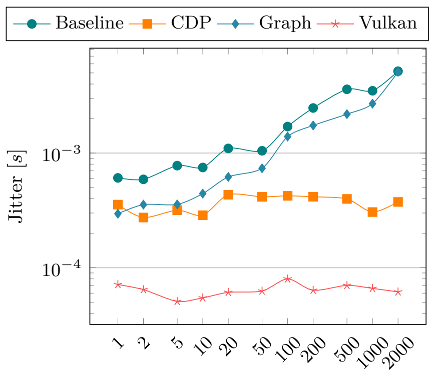
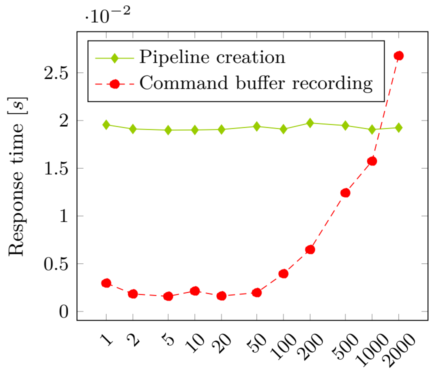


Figure 9 Results of response time for submission (left) and execution (right) for $l \in [100, 200, 500, 1000, 2000]$, $block = 16$ and $k \in [1, 2, 4, 8]$. Y-scale is logarithmic and different between left and right column.



(a) Difference between maximum and minimum submission times for each model, $k = 1$.



(b) Job-creation overheads for Vulkan.

Figure 10 Submission jitter and Vulkan creation latencies. X-Axis scale is logarithmic.

We now discuss the right side of Figures 8 and 9 detailing the total response times. The Vulkan approach still outperforms the other three methodologies. Increasing l, the measured response times increase in all the tested methodologies, as expected. However, Vulkan performance improvement over CUDA becomes more evident and stabilizes with larger l values (same slope observable in Figure 9), starting from $2\times$ for $l = 1$ up to $\sim 6\times$ for $l \geq 10$. CDP always performs worse then the other two CUDA based approaches and the deterioration of CDP performance gets more noticeable when increasing k and l. CUDA graphs execution times are comparable to the baseline implementation.

7.1 Discussion

The Vulkan design philosophy is responsible for such surprisingly short submission times. More specifically, by forcing the application developer to explicitly instantiate a pre-recorded version of the command pipeline and by avoiding any runtime validation checks, it manages to drastically reduce the interactions between the CPU and the GPU. On the contrary, by the time a CUDA runtime function is called to when it is actually enqueued in the command buffer, an overhead is added in terms of validation and error checking: such operations act as added overhead on the CPU side as they operate in an undisclosed and uncontrolled fashion.

An experiment with just four lines of code can serve as an example to reveal validation mechanisms at API level: let us create a CUDA runtime application that creates a host and a device pointer (respectively a pointer to CPU-visible memory region, and a GPU-only visible pointer). We allocate memory for both and we use pinned (*cudaMallocHost*) memory for the host pointer. We then set up a DMA transfer indicating the host pointer as the source, the device pointer as destination, and we wrongfully flag this to be a device to host memory copy. This program compiles, runs and – according to *nvprof*, the NVIDIA CUDA profiler – executes a host to device to memory copy, hence indicating that pointers were validated with respect to their address spaces, setting the actual DMA transfer differently than originally specified. The experiment suggests that calls were checked, validated and (if necessary) corrected. It is reasonable to assume that heavier driver mechanisms are also in place for kernel invocations.

Recall that kernels are identical in both the CUDA and Vulkan version. The performance improvement shown by Vulkan is higher for submission intensive workloads, where CPU times are not negligible with respect to GPU compute times. In these cases, minimizing submission

Table 1 Tracing result summary by tracing NVIDIA GPU kernel module: number of submissions, average duration, number of additional calls required to increment l by 1, memory management calls.

Submission model	*Number of submit calls*	*Average time [s]*	Δ	*Number of MM operations*
Baseline	611	$2.66776 \cdot 10^{-6}$	3	1,484
CDP	610	$1.92295 \cdot 10^{-6}$	0	1,484
Graph	635	$2.15276 \cdot 10^{-6}$	27	1,484
Vulkan	24	$2.05 \cdot 10^{-5}$	0	1,110

times allows drastically decreasing the total execution time. If the examined workload is characterized by few kernel calls and reasonably sized data buffers, total execution times between Vulkan and CUDA would not present such significant differences. This is partially visible in our results for smaller l and larger k.

A way to avoid paying the price of driver hidden mechanisms is to relieve the CPU from heavy submission work. The CDP methodology moves the responsibility of future kernel calls to threads that run on the GPU. From our results, we discovered that this is actually a very effective approach for minimizing CPU interactions for submission, at least for reasonable sequence lengths. However, the device-side overhead for nested kernel invocations causes significant performance deterioration with respect to GPU response time. This effect was already observed in older GPU architectures [6]. Likely, the CDP model cannot provide benefits to the kind of GPU workloads used in our tests: a sequence of kernel invocations, with little variations to their launch configuration, does not resemble a tree-like structure of nested kernel invocations. We however argue that the tree-like kernel invocations suitable to CDP are not as common as simpler sequential kernel invocations, at least for typical embedded workloads.

CUDA graphs performed below our expectations for both submission and execution metrics. We assumed that this recently-introduced CUDA feature would make it behave similarly to the Vulkan API – the idea of capturing stream operations strongly resembles Vulkan's command buffer recording. Despite launching a CUDA graph involves only calling a single CUDA runtime function, the hidden driver overhead is comparable to the one observed in the baseline methodology. Since our measurements only considered user space runtime functions, the performance of CUDA graph motivated us to further examine kernel driver overheads.

7.2 Tracing the kernel driver

Although the NVIDIA GPU runtime is proprietary and closed source, its kernel driver module is open source and freely available at the NVIDIA official website. In order to understand the submission mechanisms below the user space application that we implemented for our previous tests, we can thus analyze kernel driver operations with all the tested methodologies. Basic GPU kernel driver tracing allows us to obtain initial and final timestamps of the GPU command submission into the actual command push buffer, which is a memory region written by the CPU and read by the GPU [7, 8]. By enabling the debug interface of the *gk20a* module, we discover that command push buffer writes are delimited by *gk20a_channel_submit_gpfifo* and *gk20a_channel_submitted_gpfifo* function traces. We also traced driver functions related to memory management *gk20a_mm_** to understand their impact in the kernel submission process. We recorded those traces using the *trace-cmd-record* utility, reporting our findings in Table 1.

The second column of Table 1 shows the number of submit operations when $l = 1$ and $k = 1$, accounting for both context initialization and actual work submission. The average time for these submission calls has been calculated by subtracting the timestamps of *gk20a_channel_submitted_gpfifo* and its respective *gk20a_channel_submit_gpfifo* function trace. The Δ column is simply the difference of submit calls when $l = 2$, and represents the number of additional submit calls required by any submission after the first one. The last column is the counter of memory management operations.

In terms of sheer number, submit calls show no significant difference among the CUDA methods. Regarding CUDA graphs, despite the fact that launching a graph only involves a single user space runtime function, the number of submit calls is even larger than the baseline. This explains the reason why CUDA graphs response times are comparable to the baseline. Vulkan is characterized by a 25$\times$ reduction in submit calls compared to CUDA. However, Vulkan submit duration is a magnitude larger than all the other approaches. The Δ column shows the results we expected for baseline, CDP and Vulkan: by increasing l by a single unit, the baseline presents 3 additional submit calls (1 per additional kernel invocation), whereas CPD does not present any additional calls, as nested invocations are operated inside the GPU. Vulkan driver activity continues to follow the user space application model: increasing l leads to a larger recorded command buffer, but this does not cause an increased number of submit calls as every additional invocation is prerecorded in the same Vulkan command buffer/queue. We are unable to explain the $9 \times \Delta$ for the CUDA graph value compared to the baseline, as if we assume that driver interactions for CUDA graphs would resemble the baseline, having the same Δ as the baseline.

The captured traces for memory operations are identical in all the CUDA methodologies and do not increase with l, with Vulkan presenting 25% less memory management operations compared to CUDA. For what we were able to infer from a mostly closed source system, it is safe to assume that Vulkan is able to cache and pre-record GPU commands to then minimize CPU and GPU interactions (both in user and kernel space) when offloading periodic workloads. This implies spending a significant amount of time for initialization procedures (i.e. pipeline creation and command buffer recording) to then utilize the CPU for the bare minimum for submitting an already well-formatted sequence of operations. For the sake of completeness, we profiled the time taken by the CPU for creating pipelines and recording a command buffer of variable size. These metrics are reported in Figure 10b, where we characterized the time spent on the CPU during the initialization phase of a Vulkan application.

Recall that pipeline creation manages the binding of each distinct compute program with its input/output buffer layouts: these do not change when increasing l. On the contrary, the command buffer should record all the kernel invocations in advance. As a consequence, larger invocation sequences lead to larger command buffers, with a proportional increase in recording time. Pipeline creation is the most expensive operation (tens of ms). However, if the number of invocations reaches unrealistically large values (i.e. thousands of invocation in a single batch), recording times tend to dominate over pipeline creation.

Our findings show that no matter which submission model is selected for CUDA, Vulkan bare metal approach manages to minimize and better distribute the CPU overhead, especially for pathological workloads in which the CPU offloading times tend to dominate over the actual GPU compute and DMA transfer times.

8 Conclusion and future work

In this work, we aimed at characterizing and modeling CPU-to-GPU submission latencies for real-time systems executing on heterogeneous embedded platforms. We compared recently released CUDA submission models and the novel open standard Vulkan API for GPU accelerated workloads. In an extensive set of experiments, we considered typical workloads, consisting of inferencing on a neural network of parameterized topology, to profile GPU command submission times and total execution times. The results show that CPU offloading latencies can act as a bottleneck for performance, negatively impacting predictability and jitter, and making the schedulability analysis significantly more complex due to the large number of CPU-GPU interactions.

Considering CUDA approaches, recently introduced CUDA submission models can slightly improve performance (both on submission and execution times) compared to the commonly utilized baseline approach. However, considering a deeper neural network and buffer data size, the performance penalties during the actual kernel computations reduce or invalidate the benefits of a reduced CPU activity gained for submission operations, especially for the CDP approach.

On the other hand, the Vulkan API was able to minimize and better distribute the CPU overhead in all the tested configurations. This led to significant improvements, i.e. up to 11$\times$ faster submissions and from 2$\times$ to 6$\times$ faster GPU operations, with almost negligible jitter. Moreover, the significant reduction of CPU-GPU interactions significantly simplifies the topology of the DAG to consider for deriving an end-to-end response-time analysis, allowing a tighter characterization of the system schedulability. For these reasons, we argue that the Vulkan API should be seriously considered by real-time systems designers for dispatching periodic workloads upon predictable heterogeneous SoCs.

Despite the closed nature of the NVIDIA runtime, we traced the relevant GPU driver kernel module calls to explain the performance gap between CUDA and Vulkan. In this latter, driver interactions between application and GPU driver are kept to the bare minimum, hence providing the surprising results we discussed. Moreover, it is worth noticing that Vulkan is a cross platform API, with no hardware vendor or operating system restrictions and specified as an open standard, whose importance has been recently stressed [33]. Limitations for the Vulkan approach, when compared to CUDA, are to be found in the substantially higher implementation complexity for the developer, in the notable time needed to pre-record commands/pipelines, and in the lack of an ecosystem of utility libraries (e.g. *cuDNN* for Deep Neural Network operations, *cuBLAS* for linear algebra, etc.)

As a future work, we aim to include OpenCL in this comparison to experiment on different hardware platforms. We are also in the process of investigating different use cases, in which multiple concurrent and parallel kernels overlap in time, described by DAGs with parametrized breadth. This analysis is important when considering recurrent neural network such as R-FCN [11], which includes loops among different layers. This aspect differs from the feed-forward networks that inspired the use case in our experiments. Finally, we plan to benchmark the energy consumption for all the tested approaches. We speculate that the reduced CPU-GPU interaction might bring substantial benefits to the power consumption, a crucial aspect for the considered platforms.

References

1 Mohammed Alandoli, Mahmoud Al-Ayyoub, Mohammad Al-Smadi, Yaser Jararweh, and Elhadj Benkhelifa. Using Dynamic Parallelism to Speed Up Clustering-Based Community Detection in Social Networks. In *Future Internet of Things and Cloud Workshops (FiCloudW), IEEE International Conference on*, pages 240–245. IEEE, 2016.

2 Waqar Ali and Heechul Yun. Work-in-progress: Protecting real-time GPU applications on integrated CPU-GPU SoC platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE*, pages 141–144. IEEE, 2017.

3 Tanya Amert, Nathan Otterness, Ming Yang, James H Anderson, and F Donelson Smith. GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 104–115. IEEE, 2017.

4 Sanjoy Baruah, Marko Bertogna, and Giorgio Buttazzo. *Multiprocessor Scheduling for Real-Time Systems*. Springer, 2015.

5 Jens Breitbart. Static GPU threads and an improved scan algorithm. In *European Conference on Parallel Processing*, pages 373–380. Springer, 2010.

6 Nicola Capodieci and Paolo Burgio. Efficient implementation of Genetic Algorithms on GP-GPU with scheduled persistent CUDA threads. In *Parallel Architectures, Algorithms and Programming (PAAP), 2015 Seventh International Symposium on*, pages 6–12. IEEE, 2015.

7 Nicola Capodieci, Roberto Cavicchioli, and Marko Bertogna. Work-in-Progress: NVIDIA GPU Scheduling Details in Virtualized Environments. In *2018 International Conference on Embedded Software (EMSOFT)*, pages 1–3. IEEE, 2018.

8 Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Aingara Paramakuru. Deadline-based Scheduling for GPU with Preemption Support. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 119–130. IEEE, 2018.

9 Roberto Cavicchioli, Nicola Capodieci, and Marko Bertogna. Memory Interference Characterization between CPU cores and integrated GPUs in Mixed-Criticality Platforms. In *22nd IEEE International Conference on Emerging Technologies And Factory Automation (ETFA)*, 2017.

10 Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. EffiSha: A software framework for enabling effficient preemptive scheduling of GPU. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 3–16. ACM, 2017.

11 Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-fcn: Object detection via region-based fully convolutional networks. In *Advances in neural information processing systems*, pages 379–387, 2016.

12 Glenn A Elliott and James H Anderson. Real-world constraints of GPUs in real-time systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2011 IEEE 17th International Conference on*, volume 2, pages 48–54. IEEE, 2011.

13 Glenn A Elliott and James H Anderson. Robust real-time multiprocessor interrupt handling motivated by GPUs. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 267–276. IEEE, 2012.

14 Glenn A Elliott, Bryan C Ward, and James H Anderson. GPUSync: A framework for real-time GPU management. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pages 33–44. IEEE, 2013.

15 Kshitij Gupta, Jeff A Stuart, and John D Owens. A study of persistent threads style GPU programming for GPGPU workloads. In *Innovative Parallel Computing (InPar), 2012*, pages 1–14. IEEE, 2012.

16 Islam Harb and Wu-Chun Feng. Characterizing Performance and Power towards Efficient Synchronization of GPU Kernels. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2016 IEEE 24th International Symposium on*, pages 451–456. IEEE, 2016.

17 Cheol-Ho Hong, Ivor Spence, and Dimitrios S Nikolopoulos. GPU virtualization and scheduling methods: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 50(3):35, 2017.

18 Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint*, 2017. `arXiv:1704.04861`.

19 Khronos Group Khronos. Khronos SPIR-V Registry. *Khronos Group*, 2016. URL: `https://www.khronos.org/registry/spir-v/#spec`.

20 Khronos Group Khronos. The OpenGL Shading Language Language Version: 4.50. *Khronos Group*, 2016. URL: `https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.50.pdf`.

21 Khronos Group Khronos. Vulkan 1.0.98 - A Specification. *Khronos Group*, 2019. URL: `https://www.khronos.org/registry/vulkan/specs/1.0/html/vkspec.html`.

22 Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

23 B Neelima, Bharath Shamsundar, Anjjan Narayan, Rithesh Prabhu, and Crystal Gomes. Kepler GPU accelerated recursive sorting using dynamic parallelism. *Concurrency and Computation: Practice and Experience*, 29(4):e3865, 2017.

24 CUDA Nvidia. Programming Guide Version 10.0. *Nvidia Corporation*, 2018. URL: `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

25 Ignacio Sañudo Olmedo, Nicola Capodieci, and Roberto Cavicchioli. A Perspective on Safety and Real-Time Issues for GPU Accelerated ADAS. In *IECON 2018-44th Annual Conference of the IEEE Industrial Electronics Society*, pages 4071–4077. IEEE, 2018.

26 Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

27 Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. *arXiv preprint*, 2016. `arXiv:1612.08242`.

28 Davesh Shingari, Akhil Arunkumar, and Carole-Jean Wu. Characterization and throttling-based mitigation of memory interference for heterogeneous smartphones. In *2015 IEEE International Symposium on Workload Characterization (IISWC)*, pages 22–33. IEEE, 2015.

29 Joseph A Shiraef. An exploratory study of high performance graphics application programming interfaces. Master's thesis, University of Tennessee at Chattanooga, 2016.

30 Jan-Philipp Stauffert, Florian Niebling, and Marc Erich Latoschik. Towards comparable evaluation methods and measures for timing behavior of virtual reality systems. In *Proceedings of the 22nd ACM Conference on Virtual Reality Software and Technology*, pages 47–50. ACM, 2016.

31 Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. FLEP: Enabling Flexible and Efficient Preemption on GPUs. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017.

32 Ming Yang, Tanya Amert, Kecheng Yang, Nathan Otterness, James H Anderson, F Donelson Smith, and Shige Wang. Making OpenVX Really "Real Time". In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 80–93. IEEE, 2018.

33 Ming Yang, Nathan Otterness, Tanya Amert, Joshua Bakita, James H Anderson, and F Donelson Smith. Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems. In *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

34 Peter Zhang, Eric Holk, John Matty, Samantha Misurda, Marcin Zalewski, Jonathan Chu, Scott McMillan, and Andrew Lumsdaine. Dynamic parallelism for simple and efficient GPU graph algorithms. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, page 11. ACM, 2015.

Generating and Exploiting Deep Learning Variants to Increase Heterogeneous Resource Utilization in the NVIDIA Xavier

Roger Pujol
Universitat Politecnica de Catalunya (UPC), Spain
Barcelona Supercomputing Center (BSC), Spain

Hamid Tabani
Barcelona Supercomputing Center (BSC), Spain

Leonidas Kosmidis
Barcelona Supercomputing Center (BSC), Spain

Enrico Mezzetti
Barcelona Supercomputing Center (BSC), Spain

Jaume Abella
Barcelona Supercomputing Center (BSC), Spain

Francisco J. Cazorla
Barcelona Supercomputing Center (BSC), Spain

Abstract

Deep learning-based solutions and, in particular, deep neural networks (DNNs) are at the heart of several functionalities in critical-real time embedded systems (CRTES) from vision-based perception (object detection and tracking) systems to trajectory planning. As a result, several DNN instances simultaneously run at any time on the same computing platform. However, while modern GPUs offer a variety of computing elements (e.g. CPUs, GPUs, and specific accelerators) in which those DNN tasks can be executed depending on their computational requirements and temporal constraints, current DNNs are mainly programmed to exploit one of them, namely, regular cores in the GPU. This creates resource imbalance and under-utilization of GPU resources when executing several DNN instances, causing an increase in DNN tasks' execution time requirements. In this paper, (a) we develop different variants (implementations) of well-known DNN libraries used in the Apollo Autonomous Driving (AD) software for each of the computing elements of the latest NVIDIA Xavier SoC. Each variant can be configured to balance resource requirements and performance: the regular CPU core implementation that can run on 2, 4, and 6 cores; the GPU regular and Tensor core variants that can run in 4 or 8 GPU's Streaming Multiprocessors (SM); and 1 or 2 NVIDIA's Deep Learning Accelerators (NVDLA); (b) we show that each particular variant/configuration offers a different resource utilization/performance point; finally, (c) we show how those heterogeneous computing elements can be exploited by a static scheduler to sustain the execution of multiple and diverse DNN variants on the same platform.

2012 ACM Subject Classification Computer systems organization → Neural networks; Computer systems organization → System on a chip; Computing methodologies → Graphics processors

Keywords and phrases Deep Neural Network (DNN), GPU, Heterogenous Resources

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.23

Funding This work has been partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under grant TIN2015-65316-P, the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 772773), and the HiPEAC Network of Excellence. MINECO partially supported Jaume Abella under Ramon y Cajal postdoctoral fellowship (RYC-2013-14717), Enrico Mezzetti under Juan de la Cierva-Incorporación postdoctoral fellowship (IJCI-2016-27396), and Leonidas Kosmidis under Juan de la Cierva-Formación postdoctoral fellowship (FJCI-2017-34095).

1 Introduction

Motivated by the higher accuracy achieved by deep learning (DL) solutions over traditional algorithms based on shallow learning, the use of DL in the mainstream computing domain has rapidly widespread. This covers a variety of areas ranging from pattern recognition to natural language processing. Critical real-time embedded systems (CRTES) are not an exception to this trend, with DL-based algorithms used in areas like robotics and autonomous driving (AD). In fact, DL has emerged as the reference algorithmic solution for the realization of several functionalities in AD such as computer vision (e.g., object detection and tracking), path planning, driver-monitoring systems, and voice-based command and control [11].

The other side of the coin is that DL increased accuracy comes at the cost of a substantial increase in the required computational demands. At software level, highly optimized frameworks, tools, and low-level libraries are deployed to improve the hardware utilization significantly, and also to facilitate the software development process [1, 26, 13, 27]. At hardware level, high-performance hardware is used to satisfy the massive computation needs of DL workloads [5, 2, 4], with GPUs at the forefront of those solutions and being extensively evaluated by OEMs and TIER companies in the automotive domain [3]. Despite these efforts, autonomous driving – the target domain of our work – still challenges the computational capabilities of existing solutions. Just for advanced driver-assistance systems (ADAS), which arguably require much lower performance than AD, ARM projects an 100x increase in computation needs from 2016 to 2024 [17]. Capturing these demands requires a computation capacity of tens of tera operations per second (TOPS), which can theoretically be achieved by having a variety of specialized computing elements (accelerators) in the GPU platforms and automotive system-on-chips (SoC), e.g., Tensor cores and deep learning accelerators.

Problem statement. Since DL is used in a variety of modules for different AD functionalities, several DL instances will be running simultaneously on the underlying SoC. For instance, while the object detector module analyzes the current frame, the tracking module processes the objects recognized in previous frames and matches them with the objects in the current frame. At the same time, the planning module calculates the best path trajectory. To make things worse, (i) each module can require several DNN instances to implement the required functionality, and (ii) the module can be instantiated several times, once per each input sensor, e.g. camera, LiDAR, and radar. However, while modern GPUs offer a powerful heterogeneous platform with several type of (accelerating) computing elements (CE), current DL libraries are implemented to mostly exploit one of them, which at the time of writing this paper are the regular cores in a GPU. Regardless of the specific CE, the fact that just one CE is used, heavily under-exploits modern heterogeneous SoC computation capacity. Our view is that the ability to run DL-based *variants*, each using different CEs, would improve timing and throughput, and would pay off the extra effort required to implement those different variants. As a matter of fact, recently, NVIDIA integrated powerful deep learning accelerators (NVDLA) designed and specialized for DL workloads, and Tensor cores for DL inference, which are capable of different data type operations, from `int8` to `fp16` and `fp32`, and provide massive and flexible computation capacity.

Contributions. In this paper, with focus on the Xavier SoC and the Apollo AD framework [9], we make the following main contributions:

1. DNN usage in Apollo. We perform an analysis of the number of DNN instances that can be active during the execution of Apollo. We show that at least seven instances can be active at the same time and each instance comes with different computation needs and different time constraints. Also, based on observed indicators, we conclude that the number of DNN instances is expected to increase in future AD systems.

2. We implement distinct variants of different DL libraries so that each DL application can be executed on different CEs in the NVIDIA's Xavier SoC: CPU, GPU regular cores, GPU Tensor cores, and NVDLA. Our variants are programmed such that they can be executed under different thread-level parallelism (TLP) degrees. This allows more flexibility when exploiting the existing CEs.
3. We make an in-depth analysis of the implications of running the different variants of the DL libraries on the NVIDIA's Xavier SoC and show that each implementation/TLP-setup offers a different design point in terms of used resources and performance.
4. We model a multicore cyclic executive scheduler as a linear programming (LP) problem to assess the increase in guaranteed performance enabled by heterogeneous resources. We show how the variable execution requirements exhibited by tasks on the different heterogeneous computing elements can be exploited to increase the number of advanced neural network based functionalities on the same SoC, with clear advantages in terms of reduction in procurement costs and reliability concerns.

The rest of this paper is structured as follows. Section 2 introduces DNNs and Apollo. Section 3 analyzes the DNNs used in Apollo and the projection in the use of DNN in CRTES. Section 4 presents the main details of our target platform, the NVIDIA's Xavier SoC. Section 5 details the different implementations we developed for different DL libraries and their resource usage and performance in the Xavier SoC. Section 6 shows how scheduling can benefit from this TLP-configurable implementations to increase system load or adapt DL execution time requirements to its allocated time budget. Section 7 presents the most relevant related works, and, finally, Section 8 summarizes the main conclusions of this work.

2 Deep Neural Networks and their use in AD

2.1 Introduction to DNN

Deep Neural Networks (DNNs) [35] provide high accuracy solutions in several domains including computer vision for functions such as image classification and object detection. Nowadays, DNNs are widely used in a variety of areas and CRTES are not an exception to this. Recurrent neural networks (RNNs) are another class of artificial neural networks with internal state that are very successful for history-based workloads such as speech recognition, path planning and machine translation. RNNs are used as the state-of-the-art approach for path planning in industrial autonomous driving systems.

2.2 Apollo Autonomous Driving Software

We study *Apollo* [9], arguably the most sophisticated open-source autonomous driving framework available and already deployed on a variety of prototype vehicles (including autonomous trucks). Apollo supports state-of-the-art hardware such as latest LIDARs and cameras from Velodyne and other vendors, as well as GPU acceleration. Apollo comprises 8 main modules and several sub-modules, as shown in Figure 1. These software modules operate in a software-pipelined fashion and work in general at frame level.

M_0 *Speech recognizer* processes the voice-based commands from the driver/passengers and transmit them to the control unit.

M_1 *Perception* identifies the surrounding area around the autonomous car.

$M_1.d$ The *detection* submodule is in charge of detecting obstacles and objects from different sensors.

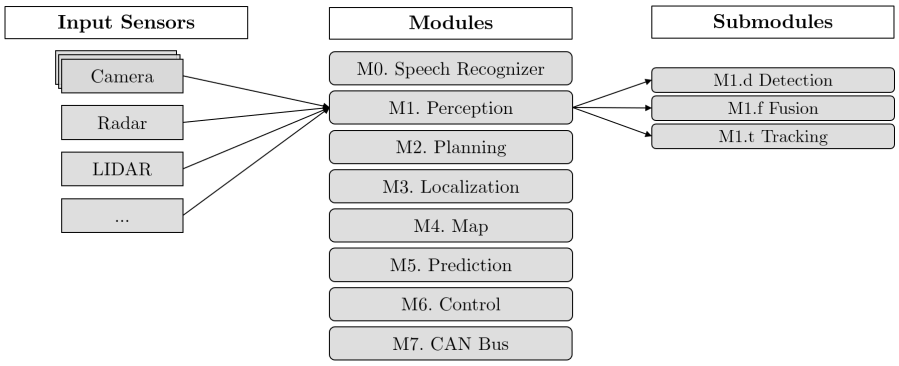


Figure 1 Modules, sub-modules and input sensors of Apollo.

- $M_1.f$ *fusion* takes the results of all detected objects from different sensors and combines them by a sensor fusion algorithm.
- $M_1.t$ *tracker* follows the detected objects and matches them with the objects detected in previous frames.

M_2 The *Planning* plans the spatio-temporal trajectory for the vehicle to take.

M_3 *Localization* leverages information received from different input sensors to estimate the precise position of the vehicle.

M_4 The *Map* provides ad-hoc structured information regarding the roads.

M_5 *Prediction* anticipates the future motion trajectories of perceived obstacles/objects.

M_6 *Control* generates control commands such as accelerating/braking and steering.

M_7 *CAN Bus* passes all the control commands to the vehicle hardware and provides information back to the autonomous system.

In this paper, we used Apollo default input data sets which are real data from sensors of an AD car collected and provided by the Apollo team. In addition, we used similar neural network architectures that Apollo employs in its different stages.

3 Analysis of the DL elements in Apollo

In the literature, we can find a wide range of DNNs and other DL algorithms. In this paper, we focus on those normally used for DL-based solutions in AD systems. In this line, Table 1 shows different types of (state-of-the-art) neural networks widely used in key domains for AD functionalities. We can observe that three modules (M_0, M_1, and M_5) and 2 sub-modules ($M1.d$ and $M1.t$) use neural networks, DNNs and RNNs in particular. Table 1 shows:

- The perception module, M_1, relies on different DNNs for detecting ($M1.d$) obstacles and objects from different sensors. The results of all detected objects are fused by a sensor fusion algorithm ($M1.f$) that does not use DNNs. As a last step, an object tracker ($M1.t$) deploys a DNN to track and follow detected objects.
- The prediction module, M_5, uses RNNs to build a model to predict the target lane that the vehicle should take. One RNN model is for lane sequences and another RNN model for the associated object states. The concatenation of these two RNNs is fed into another neural network to estimate the probability for each lane sequence. Interestingly, the modules using neural networks, *Perception* and *Prediction*, are the most compute-intensive modules: they consume more than 70% of the time Apollo uses to process each frame.

Table 1 Neural networks used in different modules of the Apollo autonomous driving system.

	Deep Learning Software	**Description**
M0. Speech Recognition	Voice Command and Control	A DNN-based accurate speech recognition application to process speech commands
M1. Perception	Camera Object Detection	A DNN-based algorithm to identify objects and traffic signals from camera sensors
	LiDAR object Detection	A DNN-based algorithm to identify objects from LiDAR sensors
	Object Tracker	A DNN-based algorithm to track identified objects in consecutive frames
M5. Prediction	Lane sequences (RNN1)	A RNN for lane sequence-based prediction
	Obstacle status (RNN2)	A RNN for obstacle status
	A RNN using the output produced by RNN1 and RNN2	A RNN to compute the probability of each lane sequence based on RNN1 and RNN2

- Some AD systems suggest to deploy AI-assistant applications to be implemented inside the cabin, which are all based on neural networks [11]. Such applications are proposed for driver-monitoring, and command and control using gestures and voice. and RNNs.

  Table 2 summarizes the modules using DNNs and RNNs.

Table 2 Modules of Apollo using DNNs (⊛) and RNNs (⊚).

	M1 (Perception)			**Other Modules**						
Input	M1.d	M1.f	M1.t	M0	M2	M3	M4	M5	M6	M7
Camera	⊛									
LiDAR	⊛		⊛	⊛				⊚ ⊚ ⊚		
Radar										

3.1 Real Execution Trace

Figure 2 shows a trace collected from an actual execution of Apollo when all DNN and RNN instances in the different modules/sub-modules use the regular GPU cores in the Jetson AGX Xavier. Each rectangle shows the span of execution of each DNN/RNN instance, i.e. since it starts running (i.e. it is executable) until its execution finishes.

As Figure 2 shows, at a given time t_i, several instances of different neural networks are executed concurrently. Also, each particular DNN/RNN has diverse computational requirements with more than 12× variability among them: the voice command runs for 4.5ms, whereas the different RNNs in the prediction module range from 48.61ms to 61.06ms; finally in the perception module DNN instances span goes from 38.96ms to 50.18ms. It is also the case that temporal constraints vary up to 10x across DNN/RNN instances. This comes from the fact that the rate at which frames arrive across different input sensors can vary from 10ms for Radars to 100ms for LiDARs.

3.2 DNN instances

Current trends show that the number of concurrent neural network instances can easily reach dozens:

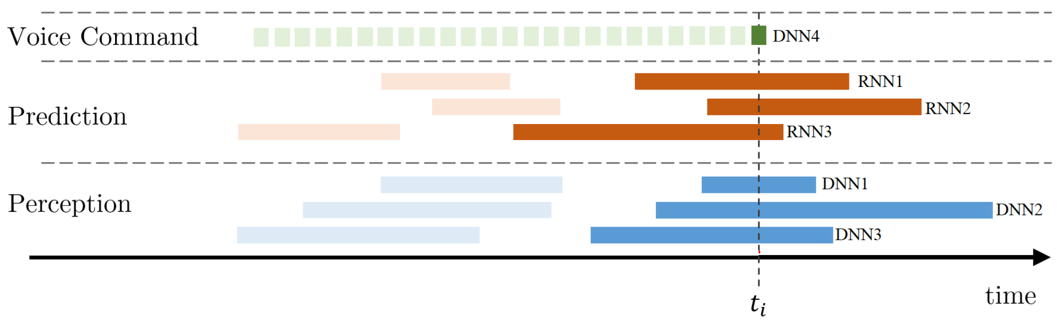


Figure 2 Concurrent execution of different modules or instances of a module in Apollo, an industrial autonomous driving software.

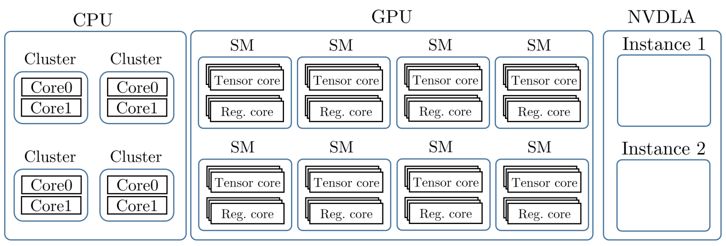
- *More input sensors.* Moving toward fully autonomous driving (Level 5 [8]) will naturally require to increase the number of sensors to cover the car's surrounding more accurately. Today, some of the AD systems, which are still far from a Level 5 system, use more than 8 cameras and radars (e.g., Telsa [28] uses 8 cameras and 12 ultrasonic sensors, and NVIDIA autopilot [11] uses 8 high-resolution cameras, 8 radars and optionally up to 3 LiDARs). Therefore, more DNN-based workloads will have to be processed, increasing the computation demand significantly.
- *More sophisticated algorithms.* Perception submodules tend to use more sophisticated DNNs, with larger number of layers and, therefore, higher computational needs for further improvements in the accuracy of object and obstacle detection, specially in conditions with reduced visibility such as fog, night, rain, and snow. The *Prediction* module already uses 3 different neural networks either to achieve higher accuracy or to cover more complex scenarios. Indeed, this type of modules normally use sophisticated neural network architectures [44, 45].
- *More functionalities.* Besides the main functions of an AD system, extra features are introduced to improve the quality and safety of driving: from gesture detection and speech-based command and control up to driver-monitoring to predict take-over readiness [30].

This trend towards exploiting multiple neural networks running in parallel and the increasing number and type of accelerators we witness in modern GPUs motivate our idea of assessing the benefits of DNN/RNN variants in modern GPUs.

4 Main Computing Elements in The Jetson AGX Xavier

NVIDIA has recently introduced the Xavier SoC as the cornerstone of its automotive platforms. Xavier delivers over 30 TOPS for DL applications while consuming less than 30 Watts. Xavier comprises four main computing elements (CEs) capable of processing deep learning workloads: traditional CPU cores, GPU cores, GPU Tensor cores, and the NVDLAs. The Xavier SoC also integrates several other accelerators such as vision accelerator, video encoder, etc. However, these accelerators cannot be used for DNN/RNN inference, due to their limited programmability. Therefore, in this paper, we focus only on CPU, GPU (regular and Tensor) cores, and the NVDLA.

1. CPU cores. The CPU complex (CCPLEX) comprises eight homogeneous carmel ARMv8.2 processors. Each core has its private instruction and data caches. In each cluster of two cores (4 clusters in total), an L2 cache is shared between both cores. An L3 cache is shared between all CPU cores.



CPU	GPU	NVDLA
1 cluster	4 SMs	1 instance
2 clusters	8 SMs	2 instances
3 clusters		

Figure 3 CEs in the Xavier SoC and granularity at which we exploit them.

2. GPU regular and Tensor cores. The Volta GPU microarchitecture comprises 512 regular cores (CUDA cores in NVIDIA terminology) and 64 Tensor cores. The GPU is structured in 8 Streaming Multiprocessors (SMs) each containing 64 regular and 8 Tensor cores.
   Tensor cores [12] accelerate large matrix operations, which are at the heart of many AI functions. While each regular core can perform up to one single precision multiply-accumulate operation per 1 GPU clock, each Tensor core can perform one matrix multiply-accumulate operation per 1 GPU clock. The Tensor core can multiply two `fp16` 4×4 matrices and adds the multiplication product `fp32` matrix to the accumulator, which is also a `fp32` 4×4 matrix.
   In each SM, threads can use either the regular cores or the Tensor cores. Hence, at most, 512 regular or 64 Tensor cores can be used in parallel.
3. NVDLA provides a flexible, robust inference acceleration solution. Xavier SoC has two NVDLAs which can be configured to run deep learning workloads. To the best of our knowledge, this is the very first work that considers NVDLAs in the real-time domain.

Overall, the NVIDIA Xavier SoC offers four different CEs that we will use to illustrate the benefits of our proposal. In order to reduce the exploration space we reduce the granularity at which we explore each CE, see the table on the right in Figure 3.

- At the CCPLEX level, we restrict our approach to core clusters. Also, since all DNN/RNN instances using the GPU or the NVDLA are initiated from the CPU, we reserve 2 CPUs for them. Overall, at the CPU level, a DNN/RNN instance can use 2, 4, or 6 of the remaining cores.
- At the GPU level, we setup a minimum granularity of 4 SMs. Hence, a DNN/RNN task can use either 4/8 SMs to exploit 256/512 GPU regular or 32/64 tensor cores, respectively.
- Each task can use one or two NVDLA accelerators.

5 Diverse DNN Implementations

This work started with a massive effort to port Apollo to the Jetson AGX Xavier. In fact, to our knowledge, this is the first work showing results of Apollo industrial framework on NVIDIA GPUs. To that end, we change the implementation in the baseline source code which is based on x86 and GPU. Depending on the target CE, we need to use appropriate libraries and re-implement Apollo modules. Deep learning workloads, in general, are implemented layer by layer by defining specific functions for each layer. Then, depending on the layer and on the highly-optimized low-level target library (e.g., cuBLAS) input data needs to be transformed to match the proper format expected by the low-level library function.

The current version of Apollo exploits only regular cores in the GPU for inferencing DNNs. The most computationally-intensive part of inference, such as convolution or fully connected layers, are usually reduced to GEneral Matrix Multiplication (GEMM), which are implemented with cuBLAS [1]. It is worth mentioning that, to our knowledge, the version of Apollo that we studied in this paper does not use TensorRT.

5.1 Specialized per-CE libraries

Table 3 presents the optimized libraries that we have used to implement our software. As it can be seen, for each specific CE, we used different libraries. In addition, we modified the baseline code in order to run the optimized code. Recently, as part of the introduction of Tensor cores, NVIDIA provided some low-level libraries to support their use. NVDLA, which can be accessed through TensorRT [6], is a platform for high-performance deep learning inference. TensorRT offers a deep learning inference optimizer and runtime that can deliver low latency and high-throughput for DL inference applications.

Table 3 Optimized libraries used to implement the Apollo software for each particular CE.

CE	Optimized Libraries
CPU	We used openMP [29] to implement all the functions to run on the CPU cores. Our implementation allows fixing the maximum number of cores that can be used.
GPU Regular Cores	The baseline implementation targets regular cores to run the kernels.
GPU Tensor Cores	We used specific libraries and adapted our code to exploit the Tensor cores. Some of our target deep neural networks consist of 100+ layers. The implementations of all the layers had to be modified.
NVDLA	We adapted each neural network configuration to be compatible with TensorRT [6], except the RNNs as they are not supported by NVDLA [7]. We use the TensorRT framework to launch applications on the NVDLAs.

5.2 Implementation for different CEs

We illustrate the required effort to modify all the functions in the source code to run the entire workload on a specific CE, by focusing on a small function performing a matrix multiplication (GEMM) operation without transposing the operand matrices. It is worth noting that each of the functions that implement the different layers of the neural networks are functionally different and therefore, each of them requires different modifications.

In this example, the matrix multiplication function builds on the following formulation, in which A, B, and C are matrices and α and β are floating-point coefficients.

$$C = \alpha A \times B + \beta C \tag{1}$$

5.2.1 CPU implementation

Figure 4 shows the CPU version of the matrix multiplication operation presented in Equation 1. As input parameters the function takes $ALPHA$ and $BETA$ as shown in Equation 1; M, N, and K that are the dimensions of the matrices; and lda, ldb, and ldc are leading dimensions

[1] For completeness, we have performed several experiments comparing the same DNN operations using cuDNN and cuBLAS. Our results show that cuBLAS achieves very competitive results w.r.t. cuDNN. However, note that main idea of the paper, i.e. having diverse DNN implementations, does not depend on the particular library used.

of matrices A, B, and C respectively. In other words, lda, ldb, and ldc determine the forward move in memory when is reached the end of a row (in row-major order) or column (in column major). In fact, these parameters define strides that provide plenty of flexibility to work with smaller tile sizes inside a larger matrix.

In the first loop, lines 4-8, the βC operation is executed according to Equation 1. In lines 10-17, the main loops are implemented to perform the matrix operations. The openMP pragma at line 9 will automatically parallelize the outer loop so that independent loop iterations can be executed in parallel.

```
 1 void OpenMPgemmNN(int M, int N, int K, float ALPHA, float const *A, int lda, float
        const *B, int ldb, float BETA, float *C, int ldc)
 2 {
 3     int i, j,k;
 4     for(i = 0; i < M; ++i){
 5         for(j = 0; j < N; ++j){
 6             C[i*ldc + j] *= BETA;
 7         }
 8     }
 9     #pragma omp parallel for
10     for(i = 0; i < M; ++i){
11         for(k = 0; k < K; ++k){
12             register float A_PART = ALPHA*A[i*lda+k];
13             for(j = 0; j < N; ++j){
14                 C[i*ldc+j] += A_PART*B[k*ldb+j];
15             }
16         }
17     }
18 }
```

Figure 4 CPU implementation of the reference matrix multiplication (gemm) operation.

5.2.2 GPU regular core implementation

Figure 5 shows the implementation for the GPU regular cores, with the function requiring the same parameters as for CPU version. Also note that in this example, we assume that the matrices are already in the device's memory space.

```
void GRCSgemmNN(int M, int N, int K, float ALPHA, float const *A,
int lda, float const *B, int ldb, float BETA, float *C, int ldc)
{
    static int init[16] = {0};              // Vector for initialized handles
    static cublasHandle_t handle[16];       // Vector of actual handles
    int i;
    cudaGetDevice(&i);                      // Get current device
    if(!init[i]) {                          // If not initialized
        cublasCreate(&handle[i]);           // Creates the handle
        init[i] = 1;
    }
    cudaError_t status = cublasSgemm(handle[i],
            CUBLAS_OP_N, CUBLAS_OP_N,       // Select the non-transpose matrices
            N, M, K,                        // Sizes of the matrices
            &ALPHA,
            B, ldb,                         // B and it's leading size
            A, lda,                         // A and it's leading size
            &BETA,
            C, ldc);                        // C and it's leading size
    if (status != cudaSuccess)              // Check if there is any error
        printf("CUDA Error: %s\n",cudaGetErrorString(status));
}
```

Figure 5 GPU implementation for regular cores.

First, we get the device ID and we check whether we have initialized a cuBLAS handle for it. If it is not the case we create a new one. Once we obtain the handle, we call `cublasSgemm` but with the matrices in reversed order. This is because C/C++ assumes a row major layout whereas CUDA assumes column major layout, which means that CUDA is reading the matrices in a transposed manner. Then, since everything is transposed, we can simply reverse the operators:

$$A \times B = C \iff B' \times A' = C'$$

Finally, we check whether cuBLAS triggered any error during the GEMM.

5.2.3 GPU Tensor core implementation

Reprogramming the GPU code to be run on the Tensor cores only requires to change the math mode to `CUBLAS_TENSOR_OP_MATH`. Nonetheless, this implementation builds on some preconditions to run on the Tensor cores: K, lda, ldb and ldc have to be multiple of 8 and N has to be multiple of 4. Figure 6 shows the implementation for the GPU Tensor cores.

```
void GTCSgemmNN(int M, int N, int K, float ALPHA, float const *A, int lda, float
    const *B, int ldb, float BETA, float *C, int ldc)
{
    static int init[16] = {0};                // Vector for initialized handles
    static cublasHandle_t handle[16];         // Vector of actual handles
    int i;
    cudaGetDevice(&i);                        // Get current device
    if(!init[i]) {                            // If not initialized
        cublasCreate(&handle[i]);             // Creates the handle
        init[i] = 1;
    }
    cublasSetMathMode(handle[i], CUBLAS_TENSOR_OP_MATH);     // Set math mode to
    enable Tensor cores
    cudaError_t status = cublasSgemm(handle[i],
            CUBLAS_OP_N, CUBLAS_OP_N,         // Select the non-transpose matrices
            N, M, K,                          // Sizes of the matrices
            &ALPHA,
            B, ldb,                           // B and it's leading size
            A, lda,                           // A and it's leading size
            &BETA,
            C, ldc);                          // C and it's leading size
    if (status != cudaSuccess)                // Check if there is any error
        printf("CUDA Error: %s\n",cudaGetErrorString(status));
}
```

Figure 6 GPU implementation for Tensor cores.

5.2.4 NVDLA

The steps we have followed to run the neural network workload on the NVDLAs, are shown in Figure 7. As a first step, the DNN configuration needs to be in the proper format, *prototxt* that is compatible with TensorRT. To that end, we developed a script that goes layer by layer in the configuration file of the neural network and changes its format to *prototxt*. It is worth mentioning that some layers in the original format are translated into several layers in *prototxt*. For instance, a *Convolutional* layer that has *Batch Normalization* and an activation of type *Leaky* is divided into four different layers: a regular convolution, a *Batch Normalization*, a scale, and a *ReLU* (Rectified Linear Unit) with negative slope.

Following this step, we obtain a functional configuration file in the proper format. However, in most cases, some layers are not supported yet by TensorRT. At the time of writing this paper, several types of layers, especially for RNNs, are not implemented, and therefore, cannot

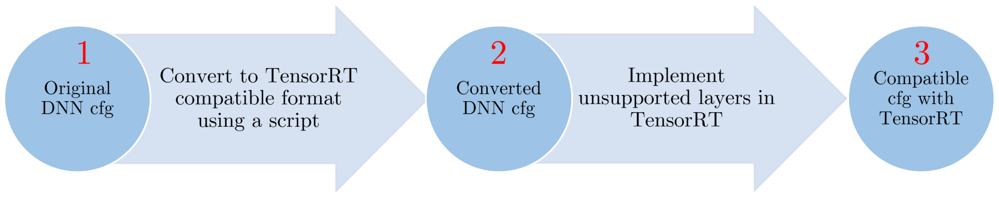

1
Original DNN cfg
Convert to TensorRT compatible format using a script
2
Converted DNN cfg
Implement unsupported layers in TensorRT
3
Compatible cfg with TensorRT

Figure 7 The steps required to specify neural network layers in order to be run on the NVDLAs.

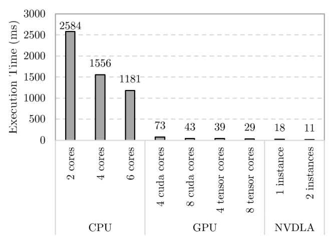
be executed on the NVDLA. To overcome this limitation, we adapted some of the available layers using equivalent and currently supported methods. The conflicting layers in our neural networks are *Upsample* layer and *Leaky ReLU* layer. To solve this issue, we implemented the layers according to TensorRT specification. After this modifications, configuration files are in the correct format required by TensorRT. Using the generated configurations, TensorRT can parse and build the model of neural network and run it in the NVDLAs. Note that TensorRT can allow unimplemented layers to fallback and run on the GPU cores. In our experiments for this paper, we avoid any fallback by adapting all the unimplemented layers using other equivalent layers (this process does not include RNN layers which, as stated before, are not supported) and the entire neural network is running on the NVDLA.

5.3 Timing Analysis Results

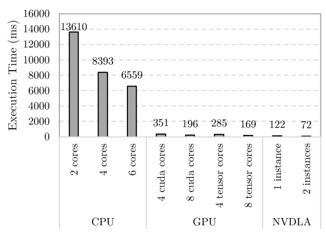
We analyse the results obtained for each DNN variant under different CE and TLP levels: CPU cores (2 cores, 4 cores, and 6 cores), GPU regular cores (4 SMs, 8 SMs), GPU Tensor cores (4 SMs, 8 SMs), and NVDLAs (1 or 2 NVDLAs). Note that we always reserve 2 cores of the CPU for managing the operating system tasks and the tasks that are running in the GPU or NVDLA, since they are also triggered by their corresponding CPU processes.

Figure 8 shows the timing results of different neural network instances of Apollo running on each CE. Timing characterization has been performed with other DNN/RNN instances run in parallel. While we did not run specific experiments to hit the worst-case timing interference among computing elements, we assume the obtained results also factor in contention effects. We can derive the following conclusions.

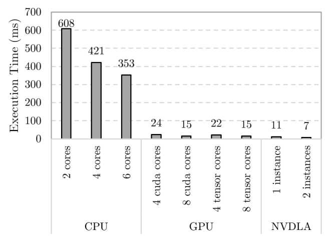
- For the camera object detector ($M1.d$), Figure 8 (a) shows that using more CPU cores significantly improves the performance. Regarding the timings on GPU CEs, as expected for this workload, Tensor cores provide better performance in comparison to the regular cores, with 8 SMs providing significantly higher performance in comparison to 4 SMs. Also, the NVDLA accelerates this NN achieving the best performance results.
- Figure 8 (b) shows the results for ($M1.d$) under another configuration for the object detector with the same neural network architecture, but with a higher camera resolution. As the results show, we have the same trends as in Figure 8 (a), however, due to the increase in the workload size, execution times increase.
- Figure 8 (c) shows the results for the LiDAR object detector, $M_1.d$. Similarly to the previous results, GPU CEs provide better performance than CPU cores, though this time Tensor cores do not result in significant improvements over GPU regular cores. NVDLA again provides the best results. As the workload is smaller than for camera detector, the times are proportionally reduced.
- Figure 8 (d) shows the timing results for object tracker, $M_1.t$. For this specific workload, GPU regular cores provide higher performance than the Tensor cores. After a detailed analysis and designing some experiments, we find out that Tensor cores achieve worse



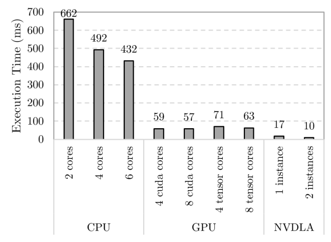
(a) M1.d normal camera resolution



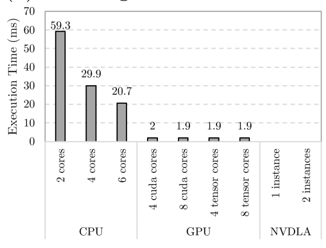
(b) M1.d high camera resolution



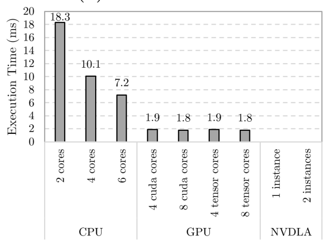
(c) M1.d LIDAR



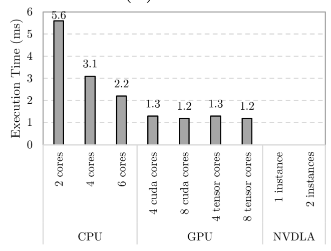
(d) M1.t



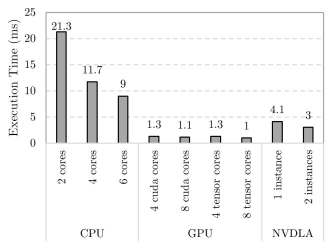
(e) M5 (RNN1)



(f) M5 (RNN2)



(g) M5 (RNN3)



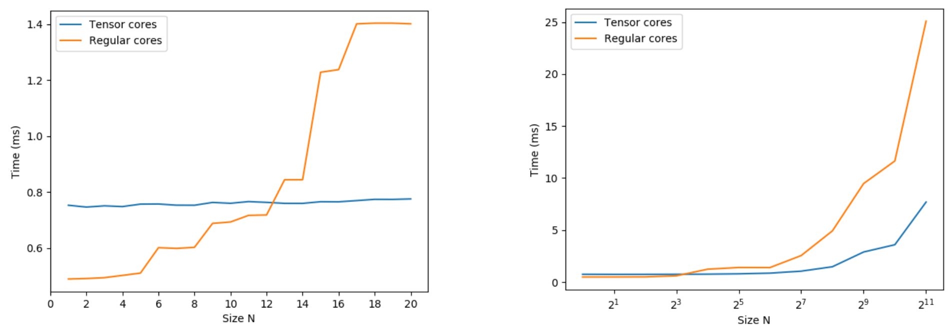
(h) M0 (Speech and comm. control)

Figure 8 Timing results of different Apollo neural network instances on different CEs.

performance than regular cores whenever we run a GEMM of $A_{M\times K}B_{K\times N}$, in which N has a very small value. More specifically as Figure 9(a) shows for $N \leq 12$ Tensor cores exhibit worse performance than regular cores. This particular case directly affects the Object Tracker (M1.t) since all the GEMMs that are performed by this DNN have $N = 1$. However, by increasing the value of N, (in this particular experiment, for $N > 12$) Tensor cores provide considerably better performance, see Figure 9 (b).

- Figures 8 (e), (f), and (g) show the timing results for the three recurrent neural networks in the prediction module. Due to the nature of the RNNs, these workloads cannot highly benefit from increased parallelization. Also GPUs provide up to an order of magnitude better performance in comparison with CPUs. However, this is too far from highly utilizing the GPU resources. In terms of the NVDLA, since several key layers of the RNN networks are not implemented in the TensorRT, we are unable to run these workloads on the NVDLA [10].
- Finally, Figure 8 (h) shows the speech recognizer module ($M0$) which uses a DNN as discussed in previous sections. This DNN network improves performance by one order of magnitude with GPU cores. Instead the NVDLA, while better than the CPU cores, performs significantly worse than the GPU cores.

Overall we can see that, (i) for some DNNs the NVDLA variant and the GPUrc (regular cores) and GPUtc (Tensor cores) variants offer comparable performance. (ii) Some times the GPUrc variant provides better results than GPUtc. (iii) It is also the case that in some cases the performance of CPU is relatively close ($\approx 2x$) of that obtained with the GPUtc



(a) N ranging from 1 to 20. **(b)** N ranging from 1 to 2^{11}.

Figure 9 Time spent in a GEMM ($A_{M \times K} B_{K \times N}$) where $M = K = 1024$.

and GPUrc variants. (iv) Across the different neural networks, we see that the CPU time requirements for some of them (e.g. (e), (f), (g), (h)) is comparable to that required by others in the GPU and NVDLA (e.g. (a), (b), (c), (d)). This makes it worth exploiting all CEs.

5.4 Other Considerations

TLP controllability. Fully exploiting variants requires exercising control on TLP as provided by NVIDIA's MPS (Multiprocessing Service), which allows multiple kernels from different processes to be executed concurrently in the GPU, while limiting their resource usage i.e., how many SMs each kernel will be using. The use of MPS has been shown to provide positive results in real-time systems [42], which paves the way for its ubiquitous adoption in all GPUs. Furthermore, MPS only requires driver updates. As this feature is not present in the Xavier SoC, we emulate its effect in our experiments by executing the GPU tasks in isolation and using the Xavier's capability to enable only a certain number of SMs in the GPU.

Contention effects on timing behavior is a widely studied topic in the real-time community mainly for CPUs, with few techniques proposed for GPUs [25] to reduce contention bounds. Contention bounding techniques, e.g. [38, 31] produce a factor Δ_{cont} to be added on top of the in-isolation timing estimates. In the scope of this paper we assume that the observed execution time factor in relevant contention effects.

Accuracy. Different implementations may use different standards for floating-point (FP) number representation (e.g. 16-bit or 32-bit representations), different FP operations or, at least, different FP operation orders. Due to rounding effects, this may lead to slightly different numerical results, whose impact on the system-level functionality needs to be assessed. However, functional results (i.e. objects detected, driving decisions, etc.) match since those tiny numerical variations have no impact in the semantics of the framework. For instance, whether the probability of recognizing an object varies by $\pm 0.1\%$ makes no practical difference in general (e.g. 90.7% vs 90.8%). Hence, despite the different implementations across CEs, the results of all implementations match functionally.

Multi-CE variants. In our current implementation, each neural network instance exploits a single CE. As future work, we consider adding a multi-CE capability, so that a single instance can exploit several CEs, e.g. 4 cores, 1 SM, and 1 NVDLA. While this offers more flexibility, our current single-CE per neural network instance approach already shows significant improvements over the baseline in which all instances use the same CE.

6 Exploiting Diversity to Increase Schedulability

With platforms supporting “diverse” computing elements and TLP degrees, the timing behavior of an application is inherently dependent on the deployed configuration. Applications will exhibit different execution time bounds depending on the actual CE they are mapped to. The overall mapping strategy is thus fundamental to determine the schedulability of a given set of applications as a whole. The identification of optimal mapping strategy, is not a specific requirement for heterogeneous platforms [19], but is a well-studied problem at the basis of several scheduling approaches for homogeneous systems, from partitioned to cyclic-executive scheduling approaches (e.g., [36, 23]). Computing an optimal partitioning is NP-hard in the general case: depending on the complexity of the problem instance, provided solutions range from exact optimization frameworks to heuristic-based approaches.

In this paper, we are interested in assessing the benefits, in terms of system schedulability, that can be enjoyed with execution platforms supporting diverse CE/TLP configurations. As a common characteristic, the different DNN instances realizing the functionalities of the AD framework can be modeled as *recurrent applications* that are periodically executed according to a given frame rate. The frame rate depends on the frequency at which inputs need to be elaborated. Static scheduling or cyclic-executive approaches are particularly suitable for this kind of systems: despite their known limitations in term of flexibility and scalability, they are relatively easy to implement and provably predictable, even on multicores. For this reasons, cyclic-executive is still widely adopted in the critical embedded real-time system domains, and is at the basis of standard frameworks (e.g., AUTOSAR [18], ARINC [16]) in critical embedded real-time system domains.

A static schedule results in the repeated execution of a sequence of intervals or frames. Tasks associated to a frame must execute and complete within that frame (i.e., performance guarantees are enforced at each frame boundary). A sequence of frames is then periodically repeated as part of a major frame, corresponding to the hyperperiod. The recurrent behavior of the diverse DNN instances (and the relative independence between them) is naturally modeled with a static schedule. Constructing a schedule for a cyclic executive consists in finding a task-to-core mapping that allows all tasks to complete within their frame (or, reciprocally, that the cumulative utilization of all tasks in a frame does not exceed 1). While there exist specific rules to define appropriate frame number and size, the schedulability of a cyclic executive systems reduces to showing that all computations have completed within the frame. A common approach to construct a valid static schedules consists in formulating the scheduling problem as a *linear programming* (LP) model.

In the scope of our evaluation, we model the problem of scheduling an heterogeneous workload of several diverse DNN instances as a cyclic executive system. We exploit a LP-based representation of the problem to assess the increase in schedulability that can arise when multiple CE/TLP configurations are supported. Without loss of generality, we assume in this paper that all DNN/RNN variants share a common time frame: the ILP formulation allows intercepting those deployment scenarios where it is impossible to schedule all the DNN variant within a frame. In the following we first discuss our assumptions in terms of schedule constraints and LP formulation, and then we present the experimental set-up.

6.1 Task Model and Linear Programming Model

We consider a periodic task system $\mathcal{T}$ and we model DNN variants as a set of n independent periodic tasks $\tau_1, \ldots, \tau_n \in \mathcal{T}$ that have to be statically scheduled on a multiprocessor platform, comprising a set of heterogeneous m cores. We assume an implicit-deadline periodic task model where each task τ_i is characterized by a period p_i and a relative deadline d_i (in this work we assume implicit-deadline tasks, thus $d_i = p_i$).

Given the heterogeneous nature of the platform, a task cannot be associated to a single-valued computational requirement. Moreover, it is not even sufficient to model the variation in the time as a function of the specific core the task is executing on. As an example, the GPU in the Xavier SoC include 8 Streaming Multiprocessors, which can be used as regular (CUDA) cores or can be configured to exploit also the Tensor cores, and an application (e.g., a DNN instance) may be executed on a variable number of SMs. Therefore, each task may exhibit different time bounds depending on the computational element (and mode) it is executed, and the TLP degree it is granted. We capture this dimensions as a set of CE/TLP configurations $\mathcal{CE} := \{ce_1, \ldots, ce_k\}$ so that each task is associated a set of time bound $\mathcal{C} = \{c_{i,1}, c_{i,2}, \ldots c_{i,k}\}$ with $c_{i,j}$ denoting the time of task τ_i under configuration $ce_j \in \mathcal{CE}$.

In line with our assessment objective, we are not interested in modeling a full static schedule over a full major frame. We limit our scope to the problem of finding a feasible schedule (if it exists) at the smallest time interval (frame) at which timing constraints are enforced. Given a set of tasks $\mathcal{T}' \subseteq \mathcal{T}$ to be statically scheduled on a set of CE/TLP configurations $\mathcal{CE}$ within a frame f, a static schedule for $\mathcal{T}'$ in f under configuration $ce_j \in \mathcal{CE}$ is valid only if the cumulative task utilization does not exceed f.

We modeled the cyclic executive scheduling problem on multiple CE/TLP configurations as an LP problem. LP-based approaches have been exploited for deriving static schedules in both homogeneous [23] and heterogeneous [19] multiprocessor systems. While the considered optimization problem is NP-hard [37], LP approaches have been shown to be effective in most cases; heuristic-based methods have been proposed to overcome scalability concerns.

An LP model comprises a set of decision variables (possibly constrained to assume only integer values), a set of linear constraints, and an objective function. Constraints and objective function are expressed as (linear) inequalities over the decision variables. The cyclic executive schedule can be modeled as an instance of a 0/1 optimization, as the sought solution will model whether or not a task is mapped to a given computational element. Intuitively, the objective function aims at minimizing the total utilization and failing to find a solution to the LP problem means that the task set is not schedulable under any feasible configuration. Other criteria can be specified in the form of weights to guide mapping decisions. It is worth noting that, in our particular case, we are not interested in finding an optimal solution, but only in proving or disproving the task set schedulability.

To instantiate the task model to the Xavier SoC, and consistently with the investigation conducted in the paper, we consider a sub-set of all the supported CE/TLP configurations $\mathcal{CE}_{Xavier} = \{\mathtt{CPU}, \mathtt{GPU}^{\mathtt{RC}}, \mathtt{GPU}^{\mathtt{RC-comb}}, \mathtt{GPU}^{\mathtt{TC}}, \mathtt{GPU}^{\mathtt{TC-comb}}, \mathtt{GPU}^{\mathtt{RC+TC}}, \mathtt{NVDLA}, \mathtt{NVDLA}^{\mathtt{comb}}\}$. Here `comb` configurations for the NVDLA and GPU cores hints at the possibility of being constrained to always use the multiple instances of the CE as a block. The set of tasks' timing bounds per configuration is given in input to the ILP as a static bi-dimensional matrix $U[\tau_i \in \mathcal{T}][ce_j \in \mathcal{CE}]$ holding the timing budget of task τ_i when deployed to node ce_j. The main decision variable consists in a bi-dimensional boolean matrix $B[\tau_i \in \mathcal{T}][ce_j \in \mathcal{CE}]$ representing whether τ_i is deployed to ce_j. Accordingly, the objective function would consist in minimizing the cumulative utilization, $\sum_{\tau_i \in \mathcal{T},\ ce_j \in \mathcal{CE}} U[\tau_i][ce_j] * B[\tau_i][ce_j]$. A set of LP constraints has been defined to guarantee a task can only be mapped to one CE/TLP (we assume tasks cannot be deployed to multiple CEs) and to enforce the maximum utilization on a CE/TLP configuration not to exceed 100% [23]. Constraints also handle the inter-correlations between CE/TLP, such as the fact that while two applications can be mapped to `NVDLA` at the same time, $\mathtt{NVDLA}^{\mathtt{comb}}$ is a configuration that implies exclusive use of the `NVDLA`.

6.2 Experimental setup

In our experiments we assess the impact that supporting different CE and thread-level parallelism (TLP) may have on the schedulability of several DNN instances on the same platform. We build on the information we derived from the Apollo software to perform a scenario-based evaluation. We used the timing profile of the applications analyzed in Section 5 to derive a predefined set of DNN (or RNN) applications (DNN_{1-5}, RNN_{1-3}), with varying computational and timing requirements under the different CE/TLPs configuration (where the set of CE/TLP configurations matches the one considered in Section 5). Each application is represented as a recurrent task with a worst-case execution time distribution, in the range $[U_{max}, U_{min}]$ milliseconds, which depends on the concrete CE/TLP configuration. The time interval has been derived by applying a $\pm15\%$ inflation factor to the values observed on the Xavier SoC reported in Figure 8. As observed in Section 5.3 those reference values also factor in contention effects and we assume the timing requirements are not changing depending on the deployment configuration of corunning applications.

Table 4 Utilization distributions for the DNN/RNN types and CE/TLP.

		CPU			GPU				NVDLA	
					GPU^{RC}		GPU^{TC}			
		2	4	6	4	8	4	8	1	2
DNN_1	U_{max}	×	×	×	83.95	49.45	44.85	33.35	20.70	12.65
	U_{min}	×	×	×	62.05	36.55	33.15	24.65	15.30	9.35
DNN_3	U_{max}	×	×	×	27.60	17.25	25.30	17.25	12.65	8.05
	U_{min}	×	×	×	20.40	12.75	18.70	12.75	9.35	5.95
DNN_4	U_{max}	×	×	×	67.85	65.55	81.65	72.45	19.55	11.50
	U_{min}	×	×	×	50.15	48.45	60.35	53.55	14.45	8.50
RNN_1	U_{max}	68.19	34.38	23.80	2.30	2.19	2.19	2.19	-	-
	U_{min}	50.41	25.42	17.60	1.70	1.62	1.62	1.62	-	-
RNN_2	U_{max}	21.05	11.62	8.28	2.19	2.07	2.19	2.07	-	-
	U_{min}	15.56	8.59	6.12	1.62	1.53	1.62	1.53	-	-
RNN_3	U_{max}	6.44	3.57	2.53	1.50	1.38	1.50	1.38	-	-
	U_{max}	4.76	2.64	1.87	1.11	1.02	1.11	1.02	-	-
DNN_5	U_{max}	24.50	13.46	10.35	1.50	1.27	1.50	1.15	4.72	3.45
	U_{max}	18.11	9.95	7.65	1.11	0.94	1.11	0.85	3.49	2.55

For each CE/TLP configuration, we generated 16,000 synthetic task sets under different overall utilization thresholds (with a mechanism similar to UUnifast [22]). Task set were generated by randomly selecting several instances of the diverse DNN/RNN types. The utilization of each DNN (RNN) is drawn from the intervals reported in Table 4 above (values are in milliseconds), which is in turn built on the timing characterization results in Section 5.3. DNN_2, the object detector version working with high resolution images in Figure 8, was not included in the evaluation as it is corresponds to a high-resolution variant of object detection application that is clearly over-demanding for the target platform. We use instead DNN_1, the object detector working with standard resolution images. Still on Table 4, it is also worth noting that applications (DNN1, DNN3, DNN4) could not be scheduled on CPU cores (utilization larger than 100%), and RNNs execution is not supported on NVDLA). This is supported in the ILP model by forcing $B[\tau_i][ce_j] = 0$ for specific combinations.

As commented above, focusing on a single scheduling frame is sufficient to fulfill our evaluation objective. We therefore assumed all applications to fit in the same frame, with a reference size of 100ms.

6.3 Schedulability results

We used our LP formulation to assess the schedulability of the task sets under specific CE/TLP configurations. All DNNs (RNNs) are required to run concurrently on the same system, as observed in the case of Apollo. A task set is considered to be infeasible if the LP problem admits no solution. We consider different CE/TLP setting, ranging from single-CE configurations (CPU, $\texttt{GPU}^{\texttt{RC}}$, $\texttt{GPU}^{\texttt{TC}}$, and NVDLA only), to mixture configuration, up to the most flexible setting where all CE/TLP configurations are supported. The experiments aim at confirming that being able to configure and exploit different computing elements with different task-level parallelism is a fundamental enabler for successfully deploying multiple DNN variants on the same system. We asses how support for different CE/TLP can be leveraged to sustain the schedulability of systems that would have been not schedulable otherwise. Also, when a system admits multiple feasible schedules, the ILP could be also instructed to identify, among the existing feasible CE/TLP configuration, the one satisfying a predefined criterion, such as maximizing performance.

In order to analyze the benefits of our neural network variant proposal, we use, as a baseline reference, single-CE setups, where only one CE is exploited. We create several scenarios in which an increasing subset of all CEs are used (CPU, $\texttt{GPU}^{\texttt{RC}}$, $\texttt{GPU}^{\texttt{TC}}$, NVDLA). In each scenario, the utilization thresholds considered for the experiments are computed on the reference utilization of the CE providing the highest performance.

The scenarios we addressed are the following:

- nvdla+gpu_rc+gpu_tc+cpu: takes $\texttt{NVDLA}^{\texttt{comb}}$ as reference highest-performance CE, and considers the CE/TLP configurations CPU, $\texttt{GPU}^{\texttt{TC}}$, $\texttt{GPU}^{\texttt{RC+TC}}$, $\texttt{NVDLA}^{\texttt{comb}}$, NVDLA;
- gpu_tc+gpu_rc+cpu: takes $\texttt{GPU}^{\texttt{TC}}$ as reference highest-performance CE, and considers the CE/TLP configurations CPU, $\texttt{GPU}^{\texttt{TC}}$, $\texttt{GPU}^{\texttt{RC+TC}}$;
- gpu_rc+cpu: only uses CPU and $\texttt{GPU}^{\texttt{RC}}$, with the latter being the highest-performing CE.

This approach lets us assess our variants approach under different scenarios with increasing number of supported CEs, each with its specific performance characteristics. Additionally, we assess the flexibility of considering all the units of the highest-performing CE as a single element with their combined performance ($\texttt{NVDLA}^{\texttt{comb}}$) versus providing the scheduler the flexibility to allocate NN instances to independent CE units (NVDLA).

As explained in Section 6.2, a large set of workloads with different NN instances has been generate for each scenario, using the cumulative utilization relative to the highest-performing CE as a threshold. In all scenarios we considered such threshold to vary in 100% to 400% utilization over the scheduling interval.

NVDLA. Figure 10 shows the ratio of feasible task sets under the considered utilization thresholds (relative to NVDLA) and CE/TLP configurations in the nvdla+gpu_rc+gpu_tc+cpu scenario. Under 100% NVDLA utilization, the NVDLA alone can always schedule the task set: both in the $\texttt{NVDLA}^{\texttt{comb}}$ and NVDLA setups we observed a 100% success ratio. This is obviously the case for $\texttt{NVDLA}^{\texttt{comb}}$, as it is the scenario used to compute the utilization threshold. But it also normally holds for two separate instances of NVDLA as the combined use of the $\texttt{NVDLA}^{\texttt{comb}}$ does not necessarily exploit full parallelism. Clearly, with increasing utilization, $\texttt{NVDLA}^{\texttt{comb}}$ cannot schedule any workload. NVDLA instead still exhibits a high success ratio at 120% utilization, that only falls rapidly at 140% and becomes zero after 160%. This is explained by the fact that the utilization is relative to the *combined* use of NVDLA which is not providing exactly double performance when compared to a single NVDLA instance.

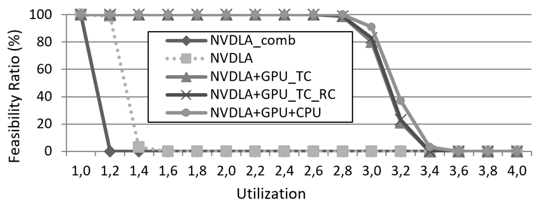


Figure 10 Percentage of schedulable workload when the NVDLA is the highest-performance CE.

Table 5 Average DNN/RNNs per workload in `nvdla+gpu_rc+gpu_tc+cpu` scenarios.

			NVDLA	NVDLA	NVDLA	
	NVDLA^{comb}	NVDLA	GPU^{TC}	GPU^{TC+RC}	GPU^{TC+RC}	CPU
1.0	12.16	12.16	12.16	12.16	12.16	
1.2	×	14.66	14.66	14.66	14.66	
1.4	×	16.50	17.23	17.23	17.23	
1.6	×	×	19.83	19.83	19.83	
1.8	×	×	22.46	22.46	22.46	
2.0	×	×	24.93	24.93	24.93	
2.2	×	×	27.56	27.56	27.56	
2.4	×	×	30.13	30.13	30.13	
2.6	×	×	32.63	32.63	32.63	
2.8	×	×	35.29	35.29	35.24	
3.0	×	×	38.46	38.31	38.13	
3.2	×	×	43.30	43.22	42.72	
3.4	×	×	×	×	49.50	

Analyzing the benefits of our variants approach, we can see that enabling the use of other CEs allows to sustain the execution of all NN instances (100% success ratio) for loads up to 2.8, significantly beyond what is observed with NVDLAs only. In between 2.8 and 3.4, the flexibility of CE/TLP deployment is exploited at most, allowing to successfully schedule some task sets. The average numbers of NN-base functionalities successfully scheduled under the considered workloads and CE/TLP configurations are reported in Table 5. Within the feasibility region, all scenarios behaves quite similarly as the average task set population grows as long as the computational load increases. Still within the feasibility region, the average number of instances does not increase when adding more CEs. The only minimal variation happens at 140% utilization, where enabling the GPU allows for one additional NN-based functionality to be successfully deployed in the average case. When the NVDLAs are saturated, the GPU elements alone are capable of providing up to 340% utilization (NVDLA-defined) and changing the GPU configuration (enabling regular CUDA cores) or introducing the CPUs is slightly affecting both schedulability and number of allocated DNNs.

GPU Tensor cores. The ratio of feasible task sets under the `gtc+grc+cpu` scenario is reported in Figure 11. The considered utilization thresholds are relative to the use of 8 `GPU`^{TC} as a block. Similarly to the NVDLA, the more flexible configuration, where GPU cores are used as two separate clusters, guarantees an improved schedulability ratio.

The improvement in terms of schedulability is even larger than in the NVDLA case, as the flexible use of the GPU allows to schedule almost 80% of the task sets even under a 150% workload. When other CEs are enabled, as suggested by our approach, the schedulability ratio further improves and reaches 85% at 160% utilization. It is interesting to note the performance improvement obtained by moving from using only `GPU`^{TC} as two independent clusters to using potentially both `GPU`^{TC} and `GPU`^{RC}. In fact, one would expect regular cores not to bring any improvement over the Tensor cores scenario, being the Tensor a more advanced accelerator

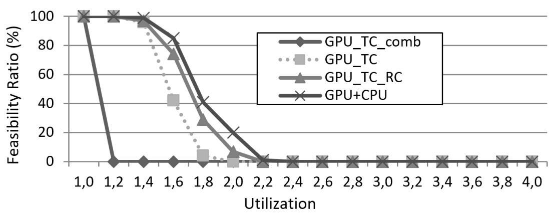


Figure 11 Percentage of schedulable workload when $\texttt{GPU}^{\texttt{TC}}$ is the highest-performance CE.

Table 6 Average DNN/RNNs per workload in `gpu_tc+gpu_rc+cpu` scenarios.

	$\texttt{GPU}^{\texttt{TC-comb}}$	$\texttt{GPU}^{\texttt{TC}}$	$\texttt{GPU}^{\texttt{TC+RC}}$	$\texttt{GPU}^{\texttt{TC+RC}}$ CPU
1.0	10.52	10.52	10.52	10.52
1.2	×	11.27	11.27	11.27
1.4	×	11.55	11.58	11.60
1.6	×	11.41	11.72	12.45
1.8	×	11.33	11.90	12.47
2.0	×	×	13.00	14.10
2.2	×	×	×	9.00

than regular GPU cores. However, while being more advanced, Tensor cores are also more specialized and their use can be counter-productive for generic applications, as can be also observed in Table 4. Exploiting the CPU, instead, allows a comparatively smaller increase in computational power, as expected. The average numbers of NN-based functionalities successfully scheduled under the considered workloads and CE/TLP configurations (see Table 6) show substantially similar values for all configurations. The configuration using Tensor cores in clusters of four shows slightly different values than those observed when enabling the regular cores and the CPU. Similarly to the NVDLA scenario, a flexible use of the $\texttt{GPU}^{\texttt{TC}}$ alone allows sustaining up to 200% utilization. Again, introducing the CPUs is not affecting schedulability and is not allowing a larger number of DNN instances.

GPU Regular cores. As final step in our incremental evaluation, we assess the benefits of our approach in an CE/TLP configuration where only the CPU and the GPU regular cores are made available. In this case, the benefit of the flexible approach $\texttt{GPU}^{\texttt{RC}}$ over the combined $\texttt{GPU}^{\texttt{RC-comb}}$ is remarkable. Conversely, the benefit offered by enabling additional CEs is less consistent, when compared to the flexible use of the reference CE. The reason is that regular GPU cores do not seem to be able to support a good degree of parallelism for NN-based functionalities, as confirmed by relatively close performance between using 4 or 8 GPU cores in Table 4. Enabling the use of CPUs only makes a negligible difference in the success ratio, which is explained by the relatively small increase in computational power provided by the CPU cores. The average number of NN-based functionalities scheduled under these configurations (see Table 7), confirms the trend observed for the NVDLA and Tensor cores scenarios. The number of scheduled instances increases with the utilization. Only few DNN instances are added in the average after enabling the use of CPUs.

7 Related Works

DL techniques are increasingly used in critical domains for they deliver substantially more precise functional results compared to other approaches. GPUs are being considered for the execution of DL software because of their capability of performing massively-parallel

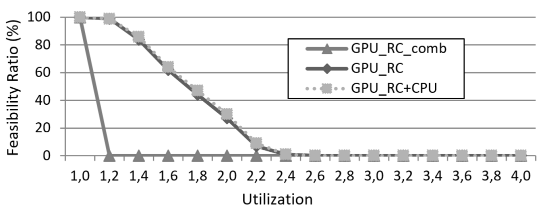


Figure 12 Percentage of schedulable workload when $\texttt{GPU}^{\texttt{RC}}$ is the highest-performance CE.

Table 7 Average DNN/RNNs per workload in `gpu_rc+cpu` scenarios.

	$\texttt{GPU}^{\texttt{RC-comb}}$	$\texttt{GPU}^{\texttt{RC}}$	$\texttt{GPU}^{\texttt{RC}}$ $\texttt{CPU}$
1.0	9.59	9.59	9.59
1.2	×	10.54	10.54
1.4	×	11.35	11.45
1.6	×	11.96	12.08
1.8	×	12.49	12.65
2.0	×	12.88	13.39
2.2	×	14.01	14.71
2.4	×	18.00	18.35

general-purpose computations and efficiency supporting DL libraries [21]. However, the use of GPUs in CRTES, such as vehicles, brings plenty of challenges for safety [14, 40] and timing. The latter, which can be categorized into three groups, have been addressed by different works: (i) some works focus on the implications of GPUs in the real-time properties of the system, (ii) others aim at improving utilization and efficiency of the existing DL and machine vision software, and (iii) other works propose low-level modifications to support DL such as scheduling algorithms or hardware support.

Research on the real-time properties of GPUs has been conducted for almost a decade. Initial works focused on scheduling proposals for the special timing behavior of GPUs, which is based on interrupts [32], and deal with their non-preemptive nature, which requires task synchronization [34]. Multiple CPU-GPU allocation strategies have been considered in [33], where the authors evaluate different partitioning and clustering schemes to enable sharing multiple instances of the same GPU across multiple cores. In our work we deal with a heterogeneous set of accelerators, and GPU regular/Tensor cores. We consider a set of diverse parallel tasks that can be scheduled under varying TLP through multiple CPUs, GPU SMs, as well as on other specialized accelerators; we study how their execution requirements varies depending on the computing element they are scheduled on. More recent works have focused on exposing undocumented or mis-documented features of NVIDIA GPUs and their benchmarking [15, 42, 39]. Moreover, [42] is the first real-time paper evaluating NVIDIA's MPS system, which allows multiple processes to execute kernels concurrently in the GPU, containing their SM usage, which is an essential feature for our work. Similarly to our work, [41] considers fine-grained vision-related schedulable entities that can be executed on CPU or GPU, but it does not consider several accelerators beyond the GPU's SMs.

Authors in [43] apply sensor fusion and propose a supervised scheduling algorithm for multiple DNN layers, considering each one as a separate dynamically schedulable entity on a GPU. Similarly to our work, the proposed approach focuses on multiple DNN instances. The focus, however, is limited to a single computational element and does not include the use of multiple elements and thread-level parallelism configurations. Bateni et al. [20] proposed

ApNet, an approximation-aware real-time neural network, to guarantee that DNN workloads meet their deadlines by using an efficient approximation. Despite their proposal can incur some accuracy loss, it can guarantee the timing predictability. Our work is orthogonal to the ApNet and applying both approaches can further improve resource utilization and performance. In another work, Bateni et al. [21] proposed *Predjoule*, which is a timing predictable energy optimization framework. Predjoule targets DNN workloads and guarantees the latency and energy efficiency of such workloads. We believe that this work can be extended to support various hardware resources and, in combination with our work, could provide better improvements in latency and energy consumption.

Capodieci et al. [24] presented a real-time scheduler for GPU activities on SoC systems such as NVIDIA Jetson TX2. They implement and test Earliest Deadline First (EDF) for GPU tasks, which is enhanced with a Constant Bandwith Server (CBS) based timing isolation mechanism. On the contrary, our work allows the co-scheduling of different computing resources such as CPU cores, GPU cores and Tensor cores, and DL accelerators.

Overall, to the best of our knowledge, this is the first work to study the performance variability of diverse DNN/RNN variants with different computing elements and TLP setups in the Xavier SoC. Also, we exploit a LP model of a heterogeneous static scheduler to assess the capability of the platform to sustain the execution of multiple DNN/RNN instances.

8 Conclusions

As the number of DNN/RNN instances running in parallel continues to increase in future AD systems, so does the ability to exploit the heterogeneous computing elements in modern computing SoCs. In this paper, in support to the first claim, we have analyzed the neural networks concurrently running in the Apollo AD and the current projections in their number. To sustain the latter claim, instead, we have created distinct variants of the different neural-network libraries used in Apollo. Our results show high diversity in the performance obtained by each variant in each of the computing elements of the Jetson AGX Xavier. This diversity provides an opportunity for exploiting the scheduling strategy to simultaneously deploy multiple NN-based instances on the same platform. We used an LP formulation for a multicore cyclic executive scheduler to demonstrate the performance increase potentially enabled by different heterogeneous computing elements, and to show how this allows deploying multiple advanced NN-based functionalities on the same SoC.

References

1 Implementation of BLAS (Basic Linear Algebra Subprograms) on top of the NVIDIA CUDA runtime. URL: `http://docs.nvidia.com/cuda/cublas/`.

2 Intel® GO™ Automated Driving Solution Product Brief. URL: `https://www.intel.es/content/dam/www/public/us/en/documents/platform-briefs/go-automated-accelerated-product-brief.pdf`.

3 NVIDIA DRIVE PX. Scalable supercomputer for autonomous driving. URL: `http://www.nvidia.com/object/drive-px.html`.

4 QUALCOMM Snapdragon 820 Automotive Processor. URL: `https://www.qualcomm.com/products/snapdragon/processors/820-automotive`.

5 RENESAS R-Car H3. URL: `https://www.renesas.com/en-us/solutions/automotive/products/rcar-h3.html`.

6 TensorRT: A platform for high-performance deep learning inference. URL: `https://developer.nvidia.com/tensorrt`.

7 TensorRT Support Matrix. URL: https://docs.nvidia.com/deeplearning/sdk/tensorrt-support-matrix/index.html.

8 AUTOMATED DRIVING, Levels of driving automation are deined in new SAE International standard J3016., 2014. URL: https://www.sae.org/standards/content/j3016_201609/.

9 APOLLO, an open autonomous driving platform., 2018. URL: http://apollo.auto/.

10 Deep Learning SDK Documentation, 2018. URL: https://docs.nvidia.com/deeplearning/sdk/tensorrt-archived/tensorrt-504/tensorrt-support-matrix/index.html.

11 Self-driving Safety Report, 2018. URL: https://www.nvidia.com/en-us/self-driving-cars/safety-report/.

12 Tensor Core, The Next Generation of Deep Learning., 2018. URL: https://www.nvidia.com/en-us/data-center/tensorcore/.

13 Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL: https://www.tensorflow.org/.

14 Sergi Alcaide, Leonidas Kosmidis, Hamid Tabani, Carles Hernandez, Jaume Abella, and Francisco J Cazorla. Safety-Related Challenges and Opportunities for GPUs in the Automotive Domain. *IEEE Micro*, 38(6):46–55, 2018.

15 Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed. In *IEEE Real-Time Systems Symposium (RTSS)*, 2017.

16 ARINC. *Specification 651: Design Guide for Integrated Modular Avionics*. Aeronautical Radio, Inc, 1997.

17 ARM. ARM Expects Vehicle Compute Performance to Increase 100x in Next Decade, 2015. URL: https://www.arm.com/about/newsroom/arm-expects-vehicle-compute-performance-to-increase-100x-in-next-decade.php.

18 AUTOSAR. *Specification of RTE Software - AUTOSAR CP Release 4.3.1*, 2017.

19 Sanjoy K. Baruah, Vincenzo Bonifaci, Renato Bruni, and Alberto Marchetti-Spaccamela. ILP models for the allocation of recurrent workloads upon heterogeneous multiprocessors. *Journal of Scheduling*, pages 1–15, 2018.

20 Soroush Bateni and Cong Liu. ApNet: Approximation-Aware Real-Time Neural Network. In *IEEE Real-Time Systems Symposium (RTSS)*, 2018.

21 Soroush Bateni, Husheng Zhou, Yuankun Zhu, and Cong Liu. PredJoule: A Timing-Predictable Energy Optimization Framework for Deep Neural Networks. In *IEEE Real-Time Systems Symposium (RTSS)*, 2018.

22 Enrico Bini and Giorgio C. Buttazzo. Measuring the Performance of Schedulability Tests. *Real-Time Systems*, 30(1):129–154, 2005.

23 Alan Burns, C Deutschbein, Thomas David Fleming, and S Baruah. Multi-core Cyclic Executives for Safety-Critical Systems. *Dependable Software Engineering Theories, Tools and Application*, 172:94–109, 2017.

24 Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Aingara Paramakuru. Deadline-based Scheduling for GPU with Preemption Support. In *IEEE Real-Time Systems Symposium (RTSS)*, 2018.

25 Roberto Cavicchioli, Nicola Capodieci, and Marko Bertogna. Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms. In *IEEE Emerging Technologies and Factory Automation (ETFA)*, 2017.

26 Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint*, 2014. `arXiv:1410.0759`.

27 François Chollet. Keras, 2015. URL: `https://github.com/fchollet/keras`.

28 Tesla Corp. Tesla Autopilot, 2018. URL: `https://www.tesla.com/autopilot`.

29 Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering (CiSE)*, 5(1):46–55, 1998.

30 Nachiket Deo and Mohan M Trivedi. Looking at the Driver/Rider in Autonomous Vehicles to Predict Take-Over Readiness. *arXiv preprint*, 2018. `arXiv:1811.06047`.

31 Enrique Díaz, Enrico Mezzetti, Leonidas Kosmidis, Jaume Abella, and Francisco J. Cazorla. Modelling multicore contention on the AURIX^{TM} TC27x. In *ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018.

32 Glenn A. Elliott and James H. Anderson. Robust Real-Time Multiprocessor Interrupt Handling Motivated by GPUs. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

33 Glenn A. Elliott and James H. Anderson. Exploring the Multitude of Real-Time Multi-GPU Configurations. In *IEEE Real-Time Systems Symposium (RTSS)*, 2014.

34 Glenn A. Elliott, Bryan C. Ward, and James H. Anderson. GPUSync: A Framework for Real-Time GPU Management. In *IEEE Real-Time Systems Symposium (RTSS)*, 2013.

35 Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *IEEE Computer Vision and Pattern Recognition (CVPR)*, 2014.

36 Joël Goossens, Pascal Richard, Markus Lindström, Irina Iulia Lupu, and Frédéric Ridouard. Job Partitioning Strategies for Multiprocessor Scheduling of Real-time Periodic Tasks with Restricted Migrations. In *ACM Real-Time and Network Systems (RTNS)*, 2012.

37 Richard Karp. Reducibility Among Combinatorial Problems. *Complexity of Computer Computations*, 40:85–103, 1972.

38 Jan Nowotsch, Michael Paulitsch, Daniel Bühler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

39 Nathan Otterness, Ming Yang, Sarah Rust, Eunbyung Park, James H. Anderson, F. Donelson Smith, Alex Berg, and Shige Wang. An Evaluation of the NVIDIA TX1 for Supporting Real-Time Computer-Vision Workloads. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

40 Hamid Tabani, Leonidas Kosmidis, Jaume Abella, Guillem Bernat, and Francisco J Cazorla. Assessing the Adherence of Industrial Autonomous Driving Software to ISO-26262 Guidelines for Software. In *ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2019.

41 Ming Yang, Tanya Amert, Kecheng Yang, Nathan Otterness, James H. Anderson, F. Donelson Smith, and Shige Wang. Making OpenVX Really "Real Time". In *IEEE Real-Time Systems Symposium (RTSS)*, 2018.

42 Ming Yang, Nathan Otterness, Tanya Amert, Joshua Bakita, James H. Anderson, and F. Donelson Smith. Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2018.

43 Husheng Zhou, Soroush Bateni, and Cong Liu. S^{3}DNN: Supervised Streaming and Scheduling for GPU-Accelerated Real-Time DNN Workloads. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018.

44 Alex Zyner, Stewart Worrall, and Eduardo Nebot. A Recurrent Neural Network Solution for Predicting Driver Intention at Unsignalized Intersections. *IEEE Robotics and Automation Letters (RA-L)*, 3(3):1759–1764, 2018.

45 Alex Zyner, Stewart Worrall, and Eduardo Nebot. Naturalistic Driver Intention and Path Prediction Using Recurrent Neural Networks. *arXiv preprint*, 2018. `arXiv:1807.09995`.

A Bandwidth Reservation Mechanism for AXI-Based Hardware Accelerators on FPGAs

Marco Pagani
Scuola Superiore Sant'Anna, Pisa, Italy
Université de Lille, CNRS, Centrale Lille, UMR 9189, CRIStAL, Lille, France
marco.pagani@santannapisa.it

Enrico Rossi
Scuola Superiore Sant'Anna, Pisa, Italy
enrico.rossi@santannapisa.it

Alessandro Biondi
Scuola Superiore Sant'Anna, Pisa, Italy
alessandro.biondi@santannapisa.it

Mauro Marinoni
Scuola Superiore Sant'Anna, Pisa, Italy
mauro.marinoni@santannapisa.it

Giuseppe Lipari
Université de Lille, CNRS, Centrale Lille, UMR 9189, CRIStAL, Lille, France
giuseppe.lipari@univ-lille.fr

Giorgio Buttazzo
Scuola Superiore Sant'Anna, Pisa, Italy
giorgio.buttazzo@santannapisa.it

Abstract

Hardware platforms for real-time embedded systems are evolving towards heterogeneous architectures comprising different types of processing cores and dedicated hardware accelerators, which can be implemented on silicon or dynamically deployed on FPGA fabric. Such accelerators typically access a shared memory to exchange a significant amount of data with other processing elements. Existing COTS solutions focus on maximizing the overall throughput of the system, rather than guaranteeing the timing constraints of individual hardware accelerators. This paper presents the AXI budgeting unit (ABU), a hardware-based solution to implement a bandwidth reservation mechanism on top of the AMBA AXI standard infrastructure for hardware accelerators deployed on FPGAs. An accurate and tractable model, as well as the corresponding analysis, are also proposed to bound the response time of hardware accelerators in the presence of ABUs, in order to verify whether they can complete before their deadlines. Finally, a set of experiments are reported to evaluate the proposed approach on a state-of-the-art platform, namely the Zynq-7020 by Xilinx. The resource consumption of the ABU has been quantified to be less than 1% of the total FPGA resources of the Zynq-7020.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → System on a chip; Hardware → Reconfigurable logic and FPGAs

Keywords and phrases AXI Bus, Bandwidth Reservation, Hardware Acceleration, FPGA

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.24

1 Introduction

Current computer architectures are evolving towards heterogeneous platforms consisting of different processing elements including general-purpose processing cores, graphics processing cores with general-purpose capabilities, and dedicated hardware accelerators [13]. Moreover, some popular modern SoCs platforms, like Altera's Stratix 10 SX [20] and Xilinx's Zynq

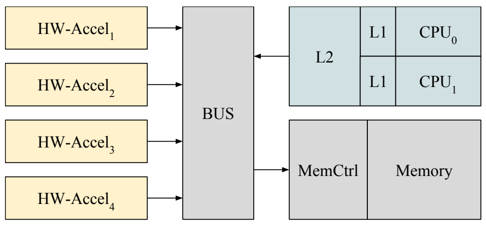


Figure 1 Block diagram of a custom system deployed on a SoC-FPGA platform.

UltraScale+ [39], include a reconfigurable FPGA fabric tightly coupled with general purpose processing elements. This feature vastly extends the capability of these platforms to allow offloading intensive computational activities from the general-purpose processing elements to custom hardware accelerators deployed on the FPGA fabric.

With respect to other types of hardware acceleration, like GPU co-processing, FPGA acceleration allows for precise control of the logic design, resulting in a very predictable behavior of the accelerators and allowing for an accurate estimation of the worst-case execution time [14, 28]. Such characteristics have made FPGA-based acceleration attractive in several safety-critical domains for signal processing and many other computationally-intensive dataflow applications [19, 21]. To name a relevant application of FPGA-based acceleration, these features enable the efficient execution of machine learning algorithms and convolutional neural networks [38] on embedded devices for safety-critical applications, as robotics and automotive.

Hardware accelerators are typically *memory-intensive*, high-performance units capable of autonomously retrieving data from the system memory using direct memory access (DMA) or bus mastering techniques. Each hardware accelerator is implemented using a subset of the FPGA's logic resources that are reserved only to that specific accelerator. Therefore the execution units of accelerators are completely independent from each other and can operate in parallel. For this reason, the execution time of a hardware accelerator depends only on the input data and the available bus and memory bandwidth. Clearly, in the context of a system comprising multiple hardware accelerators, like the one shown in Figure 1, bus/memory contention becomes the dominant factor in determining the response time of the accelerators. If the effects of such a contention are not taken into account, the system execution becomes unpredictable and hardware accelerators may introduce interferences that can jeopardize the entire system.

This scenario is worsened by the fact that often it is not possible for a designer to control the actual bus demand rate of each accelerator deployed on the system. For instance, if the accelerator is available in the form of a closed IP, it may be impossible to tune the actual rate at which bus transactions are issued. Another aspect to consider is the increasing relevance that high-level synthesis (HLS) is gaining in the design of hardware accelerators for FPGAs [26, 11]. While these tools allow for a significant speedup of the hardware design process, they lack the precise control over the design that a register-transfer level (RTL) implementation can achieve. This effectively reduces the possibility for the designer to precisely tune the rate of bus transactions. Finally, hardware accelerators can be plagued by design issues and bugs that may lead to execution overruns or illegal memory accesses.

To mitigate these issues, some hardware vendors typically integrate traditional priority-based arbitration in their interconnect implementations. More recent FPGA platforms also include (limited) mechanisms for QoS-aware arbitration [40]. However, the closed source nature of these implementations, often paired with an opaque description of the internals, makes it difficult to model such closed IPs and derive formal properties. In fact, the limited flexibility of those mechanisms and the lack of a proper reservation policy make them unsuited for safety-critical environments.

These challenges could be tackled by a methodology that enforces a more predictable environment, allowing for a controlled integration of first and third-party accelerators. As modern operating systems provide isolation and supervision mechanisms for software processes, it is worth providing supervision and reservation mechanisms also for the hardware activities performed by accelerators. This would enhance system predictability and enable the FPGA acceleration paradigm to be effectively used in safety-critical applications.

1.1 Contributions

This paper makes the following contributions.

- First, it proposes the *AXI Budgeting Unit* (ABU), which is a custom hardware component realized in programmable logic that provides bus bandwidth reservation for hardware accelerators deployed on FPGAs. An ABU *shields* a hardware accelerator from possible misbehaviors of other accelerators (in terms of exceeding bus data transfers) by *predictably* enforcing a given bus bandwidth. The ABU is not a bus arbiter but a *traffic shaper* component to be placed between hardware accelerators and a standard AMBA AXI bus infrastructure. ABUs can seamlessly be integrated into any FPGA design on top of the proprietary AXI Interconnect provided by vendors. This approach reduces the development costs and enhances portability and compatibility with any future releases of AXI-compliant IPs. ABUs have been implemented and tested upon state-of-the-art FPGA-based system-on-chips. The resource consumption of an ABU has also been quantified in less than 1% of the total FPGA resources on a Zynq-7020 platform by Xilinx.
- Second, after presenting a model for hardware accelerators based on the characteristics of realistic implementations (from Xilinx IP libraries and OpenCV), the paper proposes an analysis to bound the response times of hardware accelerators. The analysis is performed in the bus bandwidth domain and results to be tractable, as well as accurate to study FPGA-based hardware accelerators.
- Third, the paper reports a set of experimental results conducted on the Zynq-7020 aimed at demonstrating **(i)** the effectiveness of the reservation mechanism implemented by ABUs, even in the presence of misbehaving hardware accelerators, and **(ii)** the validity of the proposed analysis.

The rest of the paper is organized as follows. Section 2 presents the system model and the essential background. Section 3 presents the ABUs. Section 4 illustrates the problem of analyzing hardware tasks in the bandwidth domain and highlights crucial analysis issues. Section 5 shows how ABUs can be leveraged to analyze the system. Section 6 reports on the experimental evaluation. Section 7 reviews the related work and finally Section 8 states our conclusions.

2 System model and Background

This work focuses on FPGA-based system-on-chips and considers an AXI system composed of an interconnect, a set $\Gamma = \{\tau_1, \ldots, \tau_n\}$ of hardware accelerators, and a shared sink module (e.g., a memory controller). The hardware accelerators are implemented as AXI memory-mapped master modules capable of autonomously accessing data in a shared memory, which is reachable through the sink. Each accelerator performs a specific computational activity, therefore, from now on, they will be referred to as *hardware tasks* (HW-tasks). All HW-tasks are connected to an Interconnect block, which in turn is connected to the sink module.

The next subsections introduce a model for the Interconnect together with some essential background related to the AXI bus, a model of the HW-tasks, and a model of the sink module. It is important to note that *most of the assumptions reported in this section are only adopted for the purpose of analyzing the system* (Section 4), while the system-level mechanism proposed in this paper (Section 3) – i.e., the ABU – is independent of most of the adopted modeling strategies.

2.1 AXI Interconnect

The central element of an AXI-based system is the AXI Interconnect, which acts like a “switch” connecting one or more AXI master devices to one or more slave devices. The Interconnect performs crucial activities such as protocol conversions and the arbitration of memory transactions. In this work, the Interconnect is assumed to be configured in a N-to-1 mode, i.e., it connects $N \geq 1$ masters to a single slave device such as a memory controller. Under this setting, the Interconnect is in charge of arbitrating the transactions issued by the master modules.

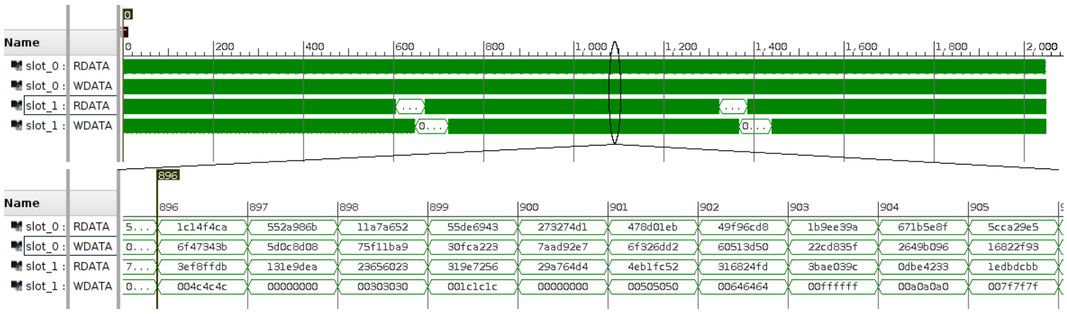
2.1.1 Arbitration policy

The AXI specification [5] does not mandate any specific arbitration protocol for the Interconnect. Some implementations of the Interconnect, such as the Xilinx standard Interconnect IP [41], provide two arbitration modes: (i) fixed-priority scheduling, in which the user configures static priorities for the slave ports, and (ii) a fair allocation using round robin. In recent releases of the Vivado suite, Xilinx provides the new SmartConnect IP [45] (meant to replace the current Interconnect IP in new designs) in which the fixed-priority arbitration has been dropped retaining the round-robin arbitration only. Hence, to match realistic modern designs, this work only focuses on round-robin arbitration. In addition, it is assumed that the Interconnect **(i)** implements *ideal round-robin scheduling* with reclaiming, i.e., the unused bandwidth is fairly re-distributed by the contenders that demand more than the fair bandwidth share, and **(ii)** does not introduce any overhead. Note that the actual implementation of the round-robin policy is typically not known, e.g., as it is the case of the Xilinx IPs, which are closed-source and lack of a proper detailed documentation concerning arbitration policies. As a result, a more accurate modeling of the arbitration may be difficult to obtain and may introduce inconsistencies among different versions of the IPs. Nevertheless, the experimental results carried out in this work surprisingly revealed a *marginal* deviation of behavior of the Xilinx Interconnects with respect to the ideal case (see Section 6).

2.1.2 AXI Links

An AXI link provides a bidirectional connection between a master and a slave interface. Each AXI link comprises five independent transaction channels: two channels (read address and read data) for read transactions, and three channels (write address, write data, and write response) for write transactions. Each channel implements a two-way handshake mechanism by using a pair of VALID and READY signals. The producer generates the VALID signal to indicate when the address or data are available. The consumer generates the READY signal to indicate that it can accept the information. The actual transfer occurs only when both the VALID and READY signals are asserted. In this paper, to distinguish between READY and VALID signals of read and write transactions, the letters R and W are appended before their names (e.g., RREADY and WREADY).

Read and write channels of a link can operate independently one from each other, i.e., each HW-task may perform read and write transactions concurrently. However, the AXI specification [5] does not mandate how the Interconnect should manage such a level of concurrency among channel groups. In this work, it is assumed that the Interconnect arbitrates read and write channel groups independently, thus permitting concurrent read and write transactions from master modules. For instance, note that both the standard and smart Interconnect IPs provided by Xilinx can operate in this mode [41], [45].



Name		896	897	898	899	900	901	902	903	904	905
slot_0 : RDATA	5...	1c14f4ca	552a986b	11a7a652	55de6943	273274d1	478d01eb	49f96cd8	1b9ee39a	671b5e8f	5cca29e5
slot_0 : WDATA	0...	6f47343b	5d0c8d08	75f11ba9	30fca223	7aad92e7	6f326dd2	60513d50	22cd835f	2649b096	16822f93
slot_1 : RDATA	7...	3ef8ffdb	131e9dea	23656023	319e7256	29a764d4	4eb1fc52	316824fd	3bae039c	0dbe4233	1edbdcbb
slot_1 : WDATA	0...	004c4c4c	00000000	00303030	001c1c1c	00000000	00505050	00646464	00ffffff	00a0a0a0	007f7f7f

Figure 2 Screenshots of bus signals for read and write memory transactions of two HW-tasks (FIR and SOBEL filters), taken from the Vivado tool by Xilinx. The HW-tasks are implemented with High Level Synthesis upon a Xilinx Zynq-7020 platform. The figure also reports a zoom of about 10 clock cycles.

2.2 HW-tasks

All HW-tasks are periodically activated, and thus generate a potentially-infinite sequence of execution instances (also referred to as *jobs*). Each HW-task operates like a DMA module, generating an equal number of read and write transactions. The transactions issued by each HW-task are assumed to be uniformly distributed during its execution and hence issued at a fixed rate. Please observe that, despite this modeling strategy may seem coarse, many real-world hardware accelerators that perform data-parallel operations (e.g., video, image, and signal processing on raw data) *present regular memory access patterns that can be modeled with a uniform demand.* As a representative example, Figure 2 reports the bus signals for memory transactions of two state-of-the-art HW-tasks, namely a FIR filter (slot0 in the figure) and a Sobel filter from the OpenCV library (slot1 in the figure). The trace at the top of the figure reports the execution of the 0.76% and the 0.6% of a job of the two HW-tasks, respectively. The HW-tasks have been implemented with high-level synthesis (HLS) upon a Xilinx Zynq-7020 platform. As it can be noted from the figure, the FIR filter exhibits a uniform pattern of transactions (one 32-bit word per clock cycle); the same holds for the Sobel filter, with the exception of a few clock cycles every about 600 clock cycles (the stop is attributed to the end of the processing of a row of the input image). Across all its execution, the amount of clock cycles in which the Sobel filter does not issue bus transactions corresponds to less than the 10%. Nevertheless, please observe that for the purpose of analysis the Sobel filter can still be pessimistically modeled by assuming that bus transactions are issued even in the last 600 clock cycles: further details on this strategy are discussed in Section 6.3.

Formally, each HW-task τ_i is characterized by the following three parameters: **(i)** a *demand rate* D_i, which represents the rate of memory transactions (both reads and writes), **(ii)** the maximum number N_i of memory transactions issued by each job, and **(iii)** its period T_i. Due to the presence of separate channels for reads and writes, the demand rate of each HW-task is bounded by two transactions per clock cycle. Demand rates are typically expressed as number of transactions per clock cycle; when needed, a word size (such as 32-bit) may also be used in place of the number of transactions. It is very important to note that HW-tasks have very *different characteristics with respect to classical software tasks*. Indeed, HW-tasks have an intrinsic parallel execution and are usually implemented such that they can perform computations while issuing memory transactions (i.e., computations and memory accesses are overlapped in time). For instance, this fact can also be observed from Figure 2, as the hardware accelerators issue memory transactions at (almost) every clock cycle. For this reason, computations times are not modeled and HW-tasks are assumed to be completed when they complete all their N_i memory transactions.

2.3 Sink module

The sink module models an endpoint block like a memory controller or a downstream AXI Interconnect (e.g., in the presence of multiple Interconnects that are connected in a hierarchical manner). Formally, the sink module is modeled with a *supply bandwidth* S that denotes the total rate of transactions it can accept, i.e., the maximum ratio of read and write transactions served per clock cycle.

It is worth mentioning that the size, in bytes, of a single transaction may vary even on the same system depending on how the AXI logic has been implemented on each module. Actually, the AXI standard allows connecting multiple hardware modules with different transaction word sizes, or even protocol version; the Interconnect is then responsible to convert the format of transactions. For instance, the High-Performance ports included in the Zynq platforms by Xilinx to access DDR memories dispose of a supply rate of two double-word (64-bit) transactions per clock cycle, while the default configuration of AXI master ports for hardware accelerators uses single-word transactions. In this paper, when it is necessary to avoid possible inconsistencies, demand and supply rates are always expressed by using the smallest word in the system.

3 AXI Budgeting Unit

This work proposes an infrastructure that comprises a set $\mathcal{A} = \{A_1, \ldots, A_n\}$ of ABU modules controlled by a central unit named *ABU controller*. Each ABU module is conceived to be placed between a hardware accelerator and the remainder of the bus infrastructure. A sample setup is shown in Figure 3. The purpose of each ABU module is to *supervise* the bus traffic generated by the corresponding hardware accelerator providing both temporal and spatial isolation. Specifically, the objectives of ABUs are:

- implementing a memory bandwidth reservation mechanism by **(i)** keeping track of the number of bus transactions issued by HW-tasks, and **(ii)** enforcing a maximum *budget* of transaction within periodic time windows; and
- as a side feature, implementing a memory protection mechanism that restricts the address space accessible by HW-tasks to a set of configurable regions.

The ABU controller serves as a central control point that allows programming the ABU modules by means of memory-mapped registers exposed through a single AXI slave interface. In its typical usage, such memory-mapped registers are controlled by the CPU (e.g., by

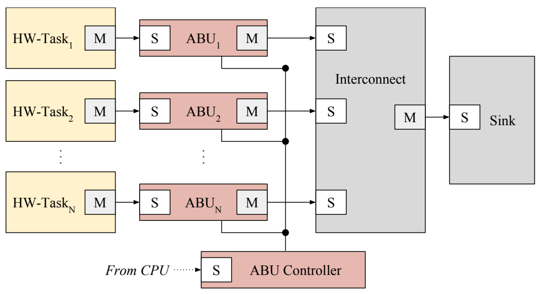


Figure 3 Illustration of an AXI system with hardware accelerators protected by ABUs. The boxes labeled with M and S denote master and slave AXI ports, respectively.

a driver at the level of the operating system or a hypervisor). The ABU modules are in turn connected to the ABU controller through a custom bus, which is used to transfer configuration parameters and signals. As it is illustrated in Figure 3, each ABU module also exports one AXI master and one AXI slave interface. The AXI slave interface serves as the access point for the hardware accelerator, while the AXI master port is meant to be connected to the remainder of the bus. These components are implemented in VHDL using a RTL behavioral description and deployed onto the FPGA fabric.

Working principle. According to the AXI standard, the master modules are the ones in charge of initiating bus transactions. Consequently, the HW-tasks drive the system by concurrently performing requests for bus transactions to the Interconnect, which in turn selects which pending transactions need to be propagated to the sink. The main idea behind the budgeting mechanism of ABUs is to act as a *proxy* between HW-tasks and the Interconnect by monitoring and altering the AXI signals. An example of a ABU in action is shown in Figure 4 for the case of a HW-task that performs a set of write transactions. The figure reports the state of the AXI signals that are relevant for the considered examples, namely WVALID in output from the HW-task and the ABU (first and second rows, respectively), WREADY in output from the Sink and the ABU (third and fourth rows, respectively), and WDATA to show the data traffic on the bus (last row). The evolution of the ABU budget over time is also reported at the top of the figure. As it can be observed from the figure, when the ABU budget ends at time t_1, write transactions are blocked despite the HW-task is ready to transmit data (WVALID in output from the HW-task is up) and the Sink is ready to receive it (WREADY in output from the Sink is up). This is accomplished by masking signals WVALID and WREADY forcing their logic state to zero, as it is illustrated in the second and fourth rows in the figure within time interval $[t_1, t_2]$. Note that, when no budget exhaustion occurs, the ABU has a transparent behavior mirroring all signals (see time window $[t_2, t_3]$ in the figure).

Budgeting mechanism. For each ABU A_i, the proposed solution allows configuring **(i)** a maximum budget B_i of number of transactions, and **(ii)** a period P_i with which the budget is replenished. Each ABU also keeps track of a variable parameter denoted as *instantaneous budget* b_i. At the system startup, $b_i = B_i, \forall i = 1, \ldots, n$. Then, as a HW-task performs bus transactions, the instantaneous budget is decremented until it reaches zero (budget depletion). As long as its instantaneous budget is zero, an ABU forbids bus transactions by acting on (R/W)VALID and (R/W)READY data and address signals. The instantaneous budget is recharged in a periodic and synchronous manner, i.e., if the system startup corresponds to

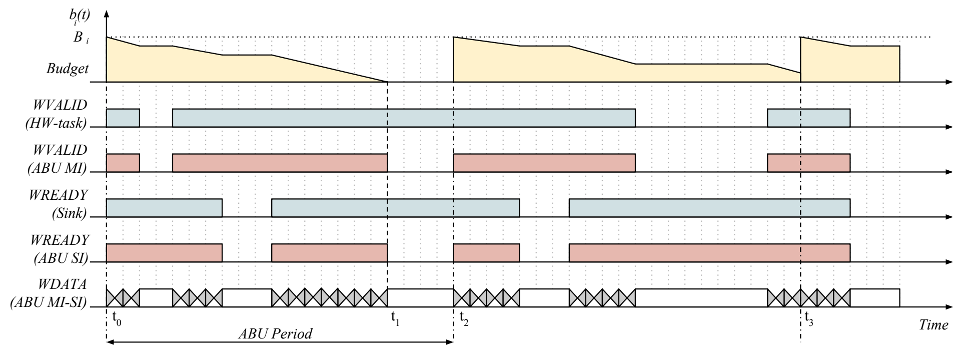


Figure 4 Example of ABU in action: impact on the AXI bus signals.

time $t = 0$, the instantaneous budget of A_i is set to $b_i = B_i$ at every time $t = kT_i,\ k \in \mathbb{N}$. From the perspective of memory bandwidth, note that each ABU enforces a transaction rate B_i/P_i for the corresponding HW-task *independently of the behavior of the latter.*

Memory protection. The ABU controller allows configuring X memory address regions for each ABU A_i to which the corresponding HW-task is allowed to access. Each of such regions $r_{i,j}$ (with $j = 1, \ldots, X$) is identified with a base memory address and a size, which are configurable by means of memory-mapped registers offered by the controller. Whenever a HW-task τ_i performs an access outside one of the regions $r_{1,1}, \ldots, r_{i,X}$, the corresponding ABU A_i blocks all memory transactions of τ_i, as it would be disconnected from the bus; consequently, the ABU controller raises an interrupt signal. The HW-task that triggered the fault can be identified by reading a status register of the controller. The normal operation of the ABU can be restored by acting on another control register offered by the controller. This feature is particularly useful in the context of virtualized systems, where a hypervisor running on the CPU of the system-on-chip can configure the memory regions and react to illegal memory accesses.

ABU internals. The internal architecture of an ABU module is illustrated in Figure 5. The communication channels on the AXI link between the master and the slave interfaces are routed through a *decoupler* block that can stop the master from issuing transactions. The decoupler works by acting on the *ready* and *valid* signals to temporarily suspend the handshake procedure. The budgeting mechanism is implemented by means of a transaction counter that keeps track of each read/write transaction and, when the budget is exhausted, sends a signal to activate the decoupler block. The ABU controller provides a pair of registers for configuring the budget and the period of each ABU. Such registers are accessible as memory-mapped via the AXI slave interface of the controller. The memory protection function is implemented by comparing the values on the read and write address channels with the range of addresses specified for each region $r_{i,j}$.

Note that the core logic of ABUs is implemented with lightweight mechanisms (counters, comparators, and switches) and hence no extra clock cycles are needed to traverse ABUs. Therefore, ABUs *do not introduce delays*: the cost of using them is only attributed to the additional FPGA resources required to be deployed. The resource utilization of one ABU and the ABU controller when implemented upon a Xilinx Zynq-7020 platform is reported in Table 1. The table also reports the percentage of resources occupied by the two modules with respect to the total amount resources available on the Zynq-7020. As it can be noted from the table, ABUs have a very marginal impact on resource consumption.

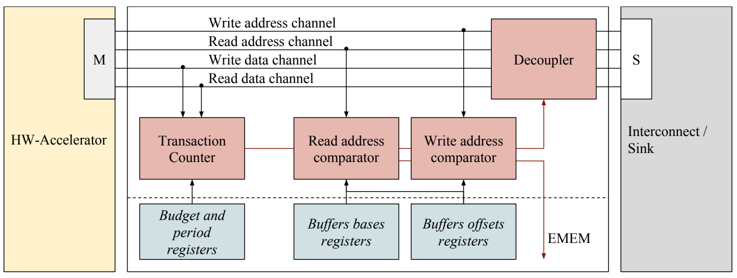


Figure 5 Internal functional block diagram of an ABU.

Table 1 Resource utilization for an ABU unit and the ABU controller on a Zynq-7020 platform.

Resource type	One ABU	ABUs Controller
LUT	436/53200 (0.82%)	279/53200 (0.56%)
FF	379/106400 (0.36%)	529/106400 (0.50%)
DSP	0/140 (0%)	0/140 (0%)
BRAM	0/220 (0%)	0/220 (0%)

4 Bandwidth-driven response-time analysis

This section studies the effect of bandwidth contention on HW-tasks under the considered modeling strategy, and presents a methodology to guarantee the system predictability using the ABU. Differently from most proposals in the literature, *the analysis proposed in this paper does not aim at accounting for possible interleaves of bus transactions over time* (e.g., like the analysis of classical periodic real-time tasks), but *aims at studying the contention incurred by HW-tasks in the bandwidth domain*, i.e., considering the actual rates at which the transactions make progress in the presence of other interfering tasks. **Why this approach?** As mentioned in Section 2.2, real-world hardware accelerators typically perform uniformly-distributed bus transactions at a constant rate, and, in particular, they even issue transactions at every clock cycle (see Figure 2). These characteristics make possible to treat HW-tasks as *fluid* computational activities that make progress at a given rate (e.g., similarly to fair multiprocessor scheduling [4]), and hence allow studying the system in bandwidth domain.

To better illustrate this peculiarity of the problem studied in this work, a simple example is firstly reported to show the effect of the contention introduced by round-robin arbitration (Sec. 4.1) in the bandwidth domain. Then, an observation concerning the critical instant for a set of HW-tasks is presented together with an illustrative example (Sec. 4.2). Finally, a strategy to enhance the system predictability by making HW-tasks prone for worst-case response-time analysis is presented (Sec. 5).

4.1 Illustrative example

To illustrate the effect of bandwidth contention incurred by HW-tasks subject to round-robin arbitration, consider a system composed of **(i)** a sink module providing a supply of $S = 6$, **(ii)** an interconnect directly connected to the sink module, and **(iii)** three HW-tasks, namely τ_1, τ_2, and τ_3, directly connected to the interconnect. The HW-tasks have the same demand $D_1 = D_2 = D_3 = S/2 = 3$ corresponding to half of the supply. The first HW-task (τ_1) needs to perform $N_1 = 6$ transactions and has a period of $T_1 = 9$ time units. The second HW-task (τ_2) performs $N_2 = 24$ transactions within a period of $T_2 = 11$ time units. Finally, the third HW-task (τ_3) performs $N_3 = 30$ transactions within a period of $T_3 = 15$ time units.

To avoid possible misunderstanding, please bear in mind that HW-tasks are statically allocated onto the FPGA area and hence do not contend the logical resources of the FPGA. For this reason, HW-tasks operate in a *parallel* fashion using their own (private) logic resources and can incur in contention only when issuing bus transactions.

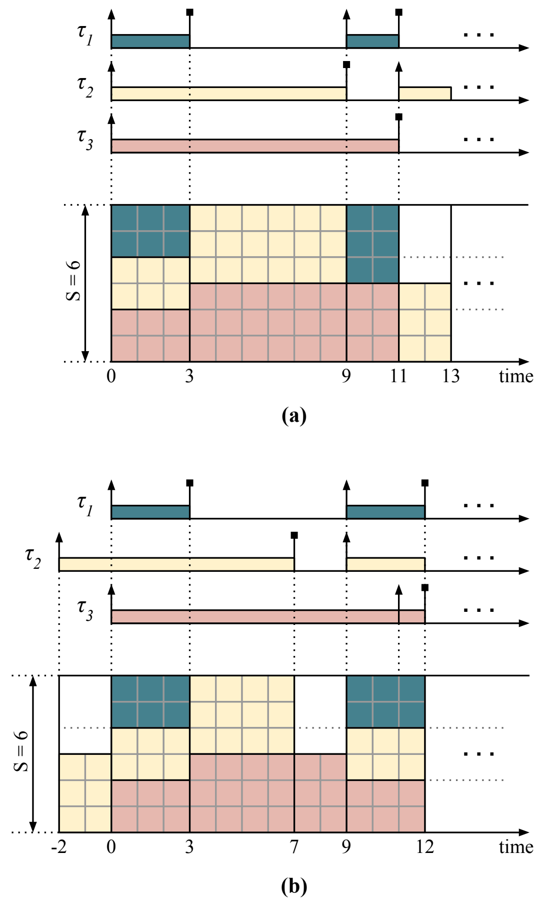


Figure 6 Examples of HW-task scheduling in the bandwidth domain with **(a)** synchronous release and **(b)** without synchronous release. In **(b)**, HW-task τ_3 experiences a longer response time with respect to the schedule in **(a)**.

Consider the case in which all HW-tasks are synchronously released at the same instant $t = 0$. Figure 6(a) illustrates the resulting schedule of the three HW-tasks by showing the intervals of time in which they are operating (on the top of the figure) and the repartition of the bandwidth over time (on the bottom of the figure). Each square unit of the bandwidth supply in the figure represents a transaction unit. At time $t = 0$, since the total bandwidth demanded by all HW-tasks $D_1 + D_2 + D_3 = 9$ exceeds the available bandwidth supply $S = 6$, the Interconnect limits the bandwidth of the three HW-tasks to a fair share of $S/3 = 2$. This bandwidth allocation continues up to $t = N_1/(S/3) = 3$, when τ_1 finishes its execution. Once τ_1 completes, τ_2 and τ_3 can proceed at their full rate of $S/2 = 3$ without suffering any contention. At time $t = 9$, τ_2 completes but a new periodic instance of τ_1 is also released. Again, both τ_1 and τ_3 can progress at their full rate without contention. At time $t = 11$, τ_1 and τ_3 complete at the same time and a new instance of τ_2 is activated. The latter can then proceed to operate while no other HW-task is active. Since τ_2 demands a bandwidth of $D_2 = 3$, half of the supply is left unused up to the next activation of τ_3 (which will occur at time $t = 15$).

4.2 Analysis issues

As it can be noted from Figure 6(a), HW-tasks are "slowed down" only when the total bandwidth demanded by active HW-tasks exceeds the supply (as it happens in $[0, 3)$ in the figure), i.e., when they make progress at a rate that is lower than their demand. Clearly, this phenomenon affects the worst-case response times of the HW-tasks.

Unfortunately, a bandwidth-driven response-time analysis cannot be accomplished by leveraging classical techniques for periodic real-time tasks. In particular, when studying the problem, we identified a set of issues (in some way similar to those identified in the analysis of multiprocessor real-time systems under global scheduling [16]) that prevent to analyze the system by looking at a single scheduling scenario.

To provide a taste of the identified issues, this section demonstrates that the classical critical instant theorem for periodic real-time tasks under uniprocessor scheduling does not hold for the problem studied in this work. Indeed, the longest response time of a HW-task *may not occur when it is synchronously released together with all other HW-tasks.*

To this end, consider the same system setup used for the previous example (Sec. 4.1). This time, assume that τ_2 is released before τ_1 and τ_3 at time $t = -2$, as shown in Figure 6(b). In this way, the first job of τ_2 can issue six transactions without suffering contention before τ_1 and τ_3 are activated at time $t = 0$. Hence the first job of τ_2 completes early (time $t = 7$) with respect to the case of synchronous release, leaving half of the bandwidth supply unused in time interval $[7, 9)$. Since τ_2 has been released earlier, also its next instance will be released earlier at time $t = 9$. The second job of τ_2 interferes with both τ_1 and τ_3 causing τ_3 to finish at time $t = 12$, i.e., one unit of time later than in the case of synchronous release. Hence τ_3 misses its deadline at time $t = 11$.

Proving a correct critical instant for the general case resulted a challenging problem that is still open for the authors. Nevertheless, as it is shown in the following section, ABUs can be extremely useful to make the system far more prone to analysis, hence increasing its predictability.

5 Response-time analysis with ABUs

Besides ABUs implement resource reservation, hence protecting the system from misbehaving HW-tasks, they can also be leveraged at the stage of analysis to help bounding the response times of the HW-tasks. Indeed, under the assumption that the ABU periods are orders of magnitude smaller then the periods of the HW-tasks, i.e., $P_i \ll \min_{i=1,\ldots,n} \{T_i\}$, ABUs can act as *bandwidth regulators* limiting the maximum demand rate of HW-tasks.

Differently to software-based reservation techniques, for which a short reservation period determines a high overhead, the assumption on ABUs' periods is practical because ABUs are realized in hardware and hence do not introduce relevant issues when adopted with short reservation periods. Specifically, as mentioned in Section 3, ABUs are built with counters and signal switches that do not introduce delays and do not represent bottlenecks for the logic circuits deployed onto the FPGA such that the operating frequency of the latter has to be limited.

Under this setting, each ABU offers to the corresponding HW-task a virtual, *dedicated* supply of bus bandwidth B_i/P_i, which is independent of the behavior of the other HW-tasks as long as the ABU budgets are guaranteed. Therefore, the problem of analyzing a set of HW-tasks protected by ABUs can be decomposed into two independent steps:

1. guaranteeing that a set of ABUs can provide the corresponding bandwidths in the worst case, i.e., their entire budgets can be safely provided in every period; and
2. guaranteeing that the bandwidth provided by each ABU is sufficient for the corresponding HW-task to meet its deadline.

These steps are addressed in the following two sub-sections, respectively.

5.1 Analyzing ABUs

As long as the sum of the bandwidths provided by a set of ABUs does not exceed the total supply S, i.e., $\sum_{i=1}^{n} B_i/P_i \leq S$, no contention can occur; therefore, it is guaranteed that their budgets can be provided within every periodic instance. However, in the general case, this condition may not hold, and hence the analysis of ABUs must account for contention exactly as discussed in the example of Section 4.1.

Nevertheless, differently from a direct analysis of HW-tasks, two observations can be leveraged to make the analysis of ABUs tractable. First, as mentioned in Section 3, ABUs are synchronously activated at the system startup. Second, due to the assumption on the ABUs' periods ($P_i \ll \min_{i=1,\ldots,n} \{T_i\}$), there is no particular advantage in assigning heterogeneous periods to ABUs, and hence to act as fluid bandwidth regulators they can be all configured with the same period P. How to configure a suitable value for the period P is discussed in the experimental evaluation reported in Section 6.

Under this setting, it is then sufficient to study the case of synchronously released ABUs by analyzing a single problem window of length P that contains a single periodic instance of each ABU. In other words, it is enough to verify that all ABUs can provide their budget before time $t = P$ assuming that they are all released at time $t = 0$.

When contention occurs, it is not straightforward to compute how the available bandwidth supply is distributed between a set of active HW-tasks. In fact, considering n arbitrary HW-tasks and a supply S, they can be classified in **(i)** those that demand less (or the same) bandwidth than the fair share S/n, and **(ii)** those that demand more bandwidth than S/n, with the result that the spare bandwidth left by HW-tasks of type (i) is fairly re-distributed between the HW-tasks of type (ii). Algorithm 1 is presented to account for this phenomenon and computes the actual share of bandwidth of a supply S for each HW-task in a set $\mathcal{C}_{HW}$ of contending HW-tasks.

Algorithm 1: Computing bandwidth shares.

```
   Input: A set of HW-tasks: $\mathcal{C}_{HW} = \{\tau_1, \ldots, \tau_m\}$
   Input: Sink supply: $S$
   Output: A set of bandwidth shares: $\overline{\mathcal{D}} = \{\overline{D}_1, \ldots, \overline{D}_m\}$
1  begin
2      $S_{rem} \leftarrow S$
3      $M \leftarrow |\mathcal{C}_{HW}|$
4      for $\tau_i^{hw} \in \mathcal{C}_{HW}$ by increasing $D_i$ do
5          $\overline{D}_i \leftarrow \min(D_i,\, S_{rem}/M)$
6          $S_{rem} \leftarrow S_{rem} - \overline{D}_i$
7          $M \leftarrow M - 1$
8      end
9      return $\overline{\mathcal{D}}$
10 end
```

The correctness of the algorithm is stated by the following lemma.

▶ **Lemma 1.** *Given a sink with supply S and a set of HW-tasks $\mathcal{C}_{HW}$ that contend for the supply, Algorithm 1 computes the correct share of bandwidth $\overline{D}_i$ assigned to each HW-task $\tau_i \in \mathcal{C}_{HW}$ under a fair arbitration.*

Proof. The proof is by induction on the iterative steps of the algorithm. *Base case (first iteration, $M = |\mathcal{C}_{HW}|$):* Let τ_i be the HW-task considered at the first iteration. If $D_i \geq S/M$, then by line 5 τ_i is assigned a bandwidth share $\overline{D}_i = S/M$, which is correct, as it corresponds to the fair share. Since the set of HW-tasks is explored in order of increasing D_i (see line 4), then all the following iterations will consider HW-tasks with $D_i \geq S/M$ and, for the same reason, will be assigned a bandwidth equal to the fair share. Otherwise, if $D_i < S/M$, then the HW-task will be assigned a bandwidth share equal to the required demand $\overline{D}_i = D_i$. Note that this cannot affect the bandwidth assignment of the other HW-tasks as D_i is lower than the fair share S/M. *Inductive case ($M < |\mathcal{C}_{HW}|$):* Suppose that the algorithm assigned a correct bandwidth to the first $|\mathcal{C}_{HW}| - M + 1$ HW-tasks and that it remains to distribute a supply bandwidth S_{rem} to $M < |\mathcal{C}_{HW}|$ HW-tasks. Let τ_i be the HW-task considered at the current iteration. Similarly to the base case, if $D_i \geq S_{rem}/M$, then by line 5 τ_i is assigned a bandwidth share $\overline{D}_i = S_{rem}/M$, which is correct, as it corresponds to the fair share with respect to the remaining $M - 1$ HW-tasks. Again, since the set of HW-tasks is explored in order of increasing D_i, the same will hold for all the following iterations. Otherwise, if $D_i < S_{rem}/M$, then the HW-task will be assigned a bandwidth share equal to the required demand $\overline{D}_i = D_i$, which again cannot affect a fair distribution for the following $M - 1$ HW-tasks. Hence the lemma follows. ◀

Leveraging Algorithm 1, it is finally possible to build a schedulability test that verifies whether a set of ABUs can provide their budget within their period P. This is accomplished by Algorithm 2, which unrolls the execution of a set of HW-tasks protected by ABUs within an analysis window $[0, P]$.

The algorithm inputs the set of HW-tasks $\mathcal{T}_{HW}$ and the corresponding set of ABUs $\mathcal{A}$ (the i-th ABU is connected to the i-th HW-task), and returns a boolean predicate that indicates whether the ABUs are schedulable or not. The algorithm keeps track of the analysis time t (initialized to $t = 0$) and the instantaneous budget b_i available for each ABU A_i, which is initialized to B_i (full budget). At the system startup ($t = 0$), all ABUs have available budget and hence all HW-tasks are considered active, i.e., they can generate transactions. Consequently, at line 4, the set of active HW-tasks, denoted with $\mathcal{C}_{HW}$, is initialized to $\mathcal{T}_{HW}$. Then, the procedure enters a loop at line 5. At each iteration, the algorithm computes the distribution of the supply S among the active HW-tasks by means of Algorithm 1, so obtaining the share of bandwidth $\overline{D}_i$ for each HW-task $\tau_i \in \mathcal{C}_{HW}$. Subsequently, it computes the amount of time Δ needed by at least one ABU A_i to provide all the available budget b_i, which is given by $\Delta = \min(b_i \,/\, \overline{D}_i)$. If a HW-task is not able to complete within the period P, then the system is deemed unschedulable and the algorithm terminates (lines 8-9). Otherwise, the algorithm proceeds by updating the budget of each ABU accounting for a lower-bound on the transactions performed in an interval of length Δ (line 12). Also, if the budget of an ABU is depleted ($b_i = 0$), then the corresponding HW-task is prevented to issue transactions and hence is removed from the set of active HW-tasks $\mathcal{C}_{HW}$ (line 14). Finally, the algorithm advances the time t by Δ and continues to iterate until the set $\mathcal{C}_{HW}$ is empty. If the algorithm completes without never detecting a deadline miss at lines 8-9, then the system is deemed schedulable.

Algorithm 2: Analysis of ABUs.

```
Input: A set of HW-tasks: $\mathcal{T}_{HW} = \{\tau_0, \dots, \tau_n\}$
Input: A set of ABUs: $\mathcal{A} = \{A_0, \dots, A_n\}$
Output: Result of the schedulability test (true/false)
begin
    $t \leftarrow 0$
    $b_i \leftarrow B_i \;\; \forall i = 1, \dots, n$
    $\mathcal{C}_{HW} \leftarrow \mathcal{T}_{HW}$
    while $\mathcal{C}_{HW} \neq \emptyset$ do
        $\overline{\mathbf{D}} \leftarrow$ Algorithm 1$(\mathcal{C}_{HW}, S)$
        $\Delta \leftarrow \min_{\tau_i \in \mathcal{C}_{HW}} (b_i \,/\, \overline{D}_i)$
        if $\Delta + t \geq P$ then
            return false
        end
        for $\tau_i \in \mathcal{C}_{HW}$ do
            $b_i \leftarrow b_i - \lfloor \overline{D}_i \cdot \Delta \rfloor$
            if $b_i = 0$ then
                $\mathcal{C}_{HW} \leftarrow \mathcal{C}_{HW} \setminus \{\tau_i\}$
            end
        end
        $t \leftarrow t + \Delta$
    end
    return true
end
```

Finally, the following lemma states that the analysis of Algorithm 2 is sustainable, i.e., increasing the ABU budgets can only worsen the schedulability of a set of ABUs (and, vice versa, a set of schedulable ABUs remains schedulable if the budgets are decreased).

▶ **Lemma 2.** *The schedulabiliy test provided by Algorithm 2 is sustainable with respect to budgets B_i.*

Proof. Suppose that a set of ABUs is not schedulable according to Algorithm 2. Hence, there exists a certain time t at which the condition at line 8 holds. Consider an arbitrary ABU A_i (associated to task τ_i) and let $[0, t')$ be the interval of time in which τ_i is in set $\mathcal{C}_{HW}$ during $[0, t)$, i.e., $t' \leq t$. There are two cases: **(i)** τ_i is still in set $\mathcal{C}_{HW}$ at time t (i.e., $t' = t$), **(ii)** τ_i left set $\mathcal{C}_{HW}$ before time t (i.e., at time $t' < t$).

Case (i): In $[0, t)$, τ_i always contributed to the bandwidth distribution by means of Algorithm 1. Hence, if the budget B_i is increased, the bandwidth shares $\overline{D}_i$ assigned during $[0, t)$ are the same and therefore the schedulability result cannot change. If $\Delta = b_i/\overline{D}_i$ (i.e., at time t, τ_i is the task detected to miss its deadline), then, by increasing the budget B_i, Δ can only increase and hence the condition at line 8 would hold too.

Case (ii): Similarly to the previous case, τ_i always contributed to the bandwidth distribution in $[0, t')$ and hence, if B_i is increased, the execution of Algorithm 2 cannot change up to time t'. If the budget B_i is increased to $B_i + \epsilon$, at time t' it can be either that the value of Δ remains the same, or that it increases too by ϵ. Consequently, τ_i will remain for more time into set $\mathcal{C}_{HW}$, contributing to the bandwidth distribution also after time t', or still leaves set $\mathcal{C}_{HW}$ at time t'. In both these cases the schedulability result cannot change.

Hence the lemma follows. ◀

5.2 Assigning ABU budgets

As ABUs act as bandwidth regulators for HW-tasks, they enforce a specific rate at which transactions are issued. Specifically, a HW-task τ_i protected by ABU A_i issues transactions at rate B_i/P as long as the ABU is guaranteed to be schedulable according to the analysis presented in the previous section. Therefore, to guarantee that τ_i is capable of performing N_i transactions within its implicit deadline T_i, it is sufficient that the following inequality is satisfied: $\frac{N_i}{B_i/P} \leq T_i$. By rewriting the latter equation, it is possible to derive a constraint on the ABU budgets to ensure the schedulability of a set $\mathcal{T}_{HW}$ of HW-tasks, i.e.,

$$\forall \tau_i \in \mathcal{T}_{HW}, \quad B_i \geq \frac{N_i \cdot P}{T_i}. \tag{1}$$

Note that the same constraint can be generalized to the case of constrained deadlines by simply replacing T_i with the relative deadline of the HW-task.

▶ **Lemma 3.** *If a set of HW-tasks $\mathcal{T}_{HW} = \{\tau_0, \ldots, \tau_n\}$ respectively protected by a set of ABUs $\mathcal{A} = \{A_0, \ldots, A_n\}$ is not schedulable (according to Algorithm 2) by setting the ABU budgets as $B_i = \frac{N_i \cdot P}{T_i}$, then it is not schedulable with any other budget assignment.*

Proof. Given the constraint of Equation (1), $B_i = \frac{N_i \cdot P}{T_i}$ is the *minimum* budget for each ABU A_i such that the schedulability of τ_i can be guaranteed. Hence, feasible budget configurations can include only budget values larger than $\frac{N_i \cdot P}{T_i}$. By Lemma 2, if a set of ABUs is not schedulable by assigning such minimum budgets, then it is also not schedulable by assigning larger budgets. Hence the lemma follows. ◀

6 Experimental evaluation

To assess the effectiveness of the ABUs on a real hardware system, an experimental evaluation has been conducted on the Zynq-7020 SoC platform by Xilinx. The Zynq-7020 belongs to the Zynq-7000 SoCs family, which comprises a collection of SoCs mainly differing for the size and class of the FPGA fabric. Almost all SoCs of the Zynq-7000 family include a dual-core ARM Cortex-A9 processor with a set of integrated peripherals (PS subsystem) tightly coupled with a 7-series FPGA fabric (PL subsystem) that can be used to extend the system with custom hardware modules. The experimental evaluation is structured in two parts: the first part aims at evaluating the effectiveness of the reservation mechanism enforced by the ABUs using DMA-like HW-tasks; the second part evaluates the ABUs with a case study application that comprises a finite impulse response (FIR) HW-task for signal processing and a Sobel HW-task for image processing from OpenCV.

All HW-tasks used in this evaluation have been designed with the Vivado high-level synthesis (HLS) tool by Xilinx. The choice of utilizing HLS comes from the steadily increasing relevance that high-level synthesis is assuming in the design of hardware accelerators. For instance, a HLS tool can also be used to synthesize a HW-task implementing a custom compute unit for executing an OpenCL kernel. The hardware-level interface of the HW-tasks used in this evaluation consists of **(i)** two AXI4 master interfaces for accessing the system memory; **(ii)** an AXI4-lite slave control interface, to expose a set of memory-mapped registers through which the software can control the HW-task; and **(iii)** an interrupt signal to notify the processor when the computation of the HW-task is completed.

Each HW-task is controlled by a periodic software task running on top of the FreeRTOS kernel, which in turn runs upon one the Cortex-A processors of the Zynq-7020. The software task relies on a device driver for managing the HW-task, feeding the addresses of the source and destination memory buffers as arguments. The driver controls the HW-tasks through the set of control registers exported via the AXI4-lite slave interface. Each job of each software task starts the corresponding HW-task and then self-suspends waiting for the HW-task to complete the execution. When the HW-task has completed, it sends an interrupt signal, which is caught by the interrupt service routine included in the driver. The service routine, in turn, wakes up the software task, which can then complete its job. This evaluation is focused on the timing properties of HW-tasks only.

Evaluation of the reservation mechanism. The first part of the experimental evaluation aims at validating the effectiveness of the reservation mechanism when one or more HW-tasks deviate from the nominal behavior by demanding a higher transaction rate and issuing more transactions than expected. Note that, from the perspective of bus contention, the bus transactions issued by HW-tasks are the only relevant aspect. Therefore, this evaluation employs a set of DMA-like HW-tasks, which allows for an almost-arbitrary control of the bus transactions that are generated. Nevertheless, also note that several hardware accelerators for FPGAs, including those of the Xilinx's IPs library such as FFT [43], FIR filter [44], and Convolution Encoder [42], require the support of a DMA for accessing the system memory.

Variants of HW-tasks. To simulate the effect of a misbehaving HW-task, three variants of the same DMA-like HW-task have been designed. Each variant differs by the amount of data N_i and the demand rate D_i. The parameters of these variants, referred to as modes, are summarized in Table 2. The demand value in MB/s is calculated by considering that each bus transaction involves a 32-bit word and that the clock rate of the FPGA is set to 100 MHz. All the HW-tasks issue 16-word burst transactions. On the Zynq-7020, the maximum supply bandwidth S available to access the memory from the PS through a high performance (HP) port is four transactions per clock for each port, as they operate in 64-bit mode (the DRAM clock is set to 525 MHz).

Table 2 Configuration of HW-tasks. The demand D_i is expressed in both transactions per clock cycle and in megabytes per second.

HW-task mode	D_i		N_i	
	[tr/clk]	[MB/s]	[tr]	[MB]
1	2	763	524288	2
2	1	381	262144	1
3	2/3	254	131072	0.5

Description of the experimental setting. The system setup used for this evaluation comprises four DMA-like HW-tasks allocated on the Zynq's PL and connected to a single HP port through an AXI Interconnect. The Interconnect is set in performance mode to maximize the bandwidth available to the HP port. An ABU module is placed between each HW-task and the Interconnect. The baseline configuration includes two HW-tasks, τ_1 and τ_2, set in mode 1, a HW-task, τ_3, operating in mode 2, and the last HW-task τ_4 set in mode 3. This configuration represents the system operating in *nominal conditions*, i.e., when all the HW-tasks respect their nominal demand D_i and data length N_i values, and is referred to as *nom*. To study the effect of misbehaving HW-tasks, two additional variants of the baseline configuration have been defined. In the first misbehaving configuration, referred to as *misb-3*, τ_3 operates in mode 1 instead of mode 2. This configuration, represents the case in which a single HW-task exceeds its nominal values, demanding a higher transaction rate and length. In the second misbehaving configuration, named *misb-3-4*, τ_3 and τ_4, normally operating in mode 2 and mode 3 respectively, now operate in mode 1. This configuration aims at reproducing the scenario in which two HW-tasks exceed their nominal values.

6.1 Profiling HW-tasks

The first set of experiments has been carried out to characterize the system configurations without ABUs. To this end, a separate profiling experiment has been conducted for each configuration of the system: the base configuration *nom*, and two misbehaving configurations *misb-3* and *misb-3-4*. These experiments allow evaluating the impact of one or more misbehaving HW-tasks on the response times of the other HW-tasks when using the default round-robin arbitration policy of the Interconnect. For this set of experiments, τ_1 is activated every 10 ms, τ_2 every 15 ms, τ_3 every 25 ms, and τ_4 every 50 ms. Measurements on the hardware have been conducted with multiple runs by testing random activation offsets of the HW-tasks, for a total of about 30 minutes of execution (collecting data for hundreds of thousands of jobs). Figure 7 presents the results of these experiments by reporting the longest-observed response times on the real hardware as solid color bars. The results corresponding to the misbehaving HW-tasks are highlighted with different colors and patterns. Comparing the response times observed under nominal conditions (*nom*) with the response times obtained under misbehaving configurations, it is evident that even a single misbehaving HW-task (*misb-3*) could have a significant impact on the response time of the other HW-tasks. This effect becomes even more tangible when taking into account the configuration *misb-3-4* in which two HW-tasks misbehave. For instance, the response time of τ_1 in *misb-3-4* increases by more than 50% with respect to nominal conditions.

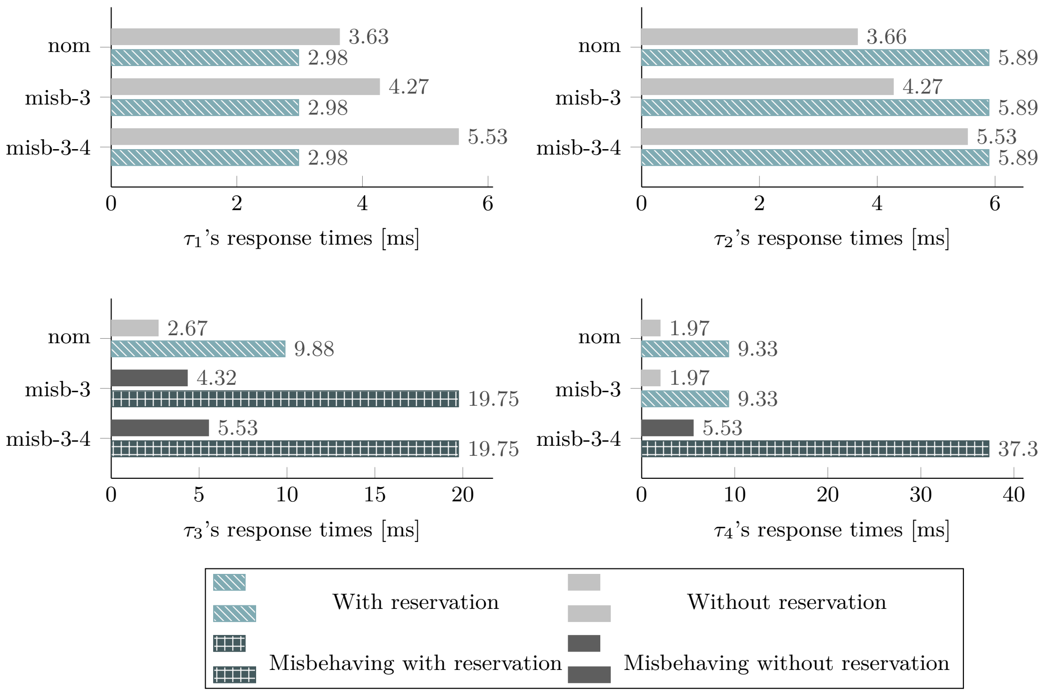


Figure 7 Response times of four HW-tasks without and with ABUs under multiple configurations.

6.2 Evaluating the reservation mechanism

The following set of experiments analyzes what happens when the ABUs are present. These experiments serve two purposes: first, to test the effectiveness of temporal isolation between HW-tasks; second, to confirm that the assumptions made in Sections 2 and 4 to model and analyze the system are realistic. To this end, the longest-observed response times on the hardware have been compared with the response-time bounds computed by the analysis of Section 4. The ABUs have been configured according to the minimum budgets provided by Lemma 3 under nominal conditions. The period of ABUs has been selected according to the following rationale. Since the ABUs count integer transactions, the period must be chosen as the smallest value that can can ensure that all the minimum budgets provided by Lemma 3 are integers. Furthermore, to avoid splitting transaction bursts, it is worth choosing a period such that the budget is a multiple of the burst size. Such a period can be easily obtained with a binary search. The resulting ABU configuration for this experimental setting is reported in Table 3. The table also reports the response times, both observed on the hardware and obtained by the analysis proposed in this work, under configuration *nom*.

As it can be noted from Figure 7, ABUs allow controlling the longest-observed response times (e.g., fixed to 2.98 for τ_1) independently of the behavior of the other HW-tasks; indeed, the response times are the same even in the misbehaving configurations *misb-3* and *misb-3-4*. Clearly, this improvement is achieved at the expenses of the misbehaving tasks (τ_3 and τ_4): in fact, their response times in misbehaving configurations is penalized.

Table 3 Configuration parameters for the ABUs and response times for the corresponding HW-tasks under the nominal configuration.

HW-Task	ABU		Response times [ms]	
	B_i [tr]	P [clk]	Longest observed	By analysis
τ_1	224	128	2.982	2.995
τ_2	112		5.893	5.991
τ_3	32		9.876	10.485
τ_4	16		9.328	10.485

6.3 A case study

The second part of the experimental evaluation considers a case-study application that comprises a FIR filter HW-task for signal processing, a Sobel HW-task for image processing, and two DMA-like HW-tasks operating in mode 1. The FIR filter implements a 12th order low-pass filter designed to process 16kHz audio samples with a cutoff frequency of 4 kHz. Internally, the FIR filter uses fixed-point representations to take advantage of the FPGA's DSP blocks. Each instance of the FIR filter processes 1 MB of samples. The Sobel filter processes 640x480 RGB images with 24-bit color depth, resulting in a size of 1200 KB. Table 4 summarizes the characteristics of these accelerators, which both issue 16-word burst transactions.

As visible from the trace shown in Figure 2, the access pattern generated by the Sobel filter HW-task is not strictly uniform due to a short pause occurring between two image lines. Such a signal analysis has been performed on the real hardware by instrumenting the design with an integrated logic analyzer (ILA) module. Clearly, the access pattern of the Sobel HW-task violates the uniform transaction hypothesis made in Section 2 to model the system. However, by performing the pessimistic assumption that the Sobel HW-task continues issuing

Table 4 Parameters of the Sobel and FIR hardware accelerators.

HW-task	D_i		N_i	
	[tr/clk]	[MB/s]	[tr]	[KB]
Sobel	1.9	725	614400	2400
FIR	2	763	524288	2048

transactions even during the brief pause between a line and the next, it is still possible to safely model it as a uniform access accelerator. Such a model can be used to assign the ABU budget and compute safe upper bounds on the response time of the Sobel HW-task. The case study application has been tested with a set of four experiments considering different HW-task periods and ABU budgets. Table 5 summarizes the parameters used for the experiments. The ABU period P is set to 128 clock cycles in all of the experiments. The results are reported in Figure 8, which compares the response times calculated using the response-time analysis presented in this paper, plotted as solid bars, with the longest-observed response times obtained on the real hardware, illustrated with striped bars. Measurements on the hardware have been performed as described in the previous section.

The experimental results show that the ABU is indeed effective even considering a case-study application comprising a realistic hardware workload suited for signal and image processing. The response times bounds obtained with the analysis are close to the longest-observed values with a maximum relative error of 3% in the case of HW-tasks with uniform demand. As expected, the maximum difference between the bound and the measurements (13%) occurs for the Sobel HW-task, since it has been pessimistically modeled by assuming a continuous bus access at its maximum rate.

Table 5 Configuration parameters for the case study (HW-task periods and ABU budgets).

Task	Experiment							
	1		2		3		4	
	T_i [ms]	B_i [tr]	T_i [ms]	B_i [tr]	T_i [ms]	B_i [tr]	T_i [ms]	B_i [tr]
FIR	6	176	6	160	8	96	6	160
Sobel	7	160	8	144	9	112	12	96
DMA-2	10	80	12	64	6	144	7	128
DMA-1	12	64	7	112	7	128	10	80

7 Related work

Resource reservation techniques have been introduced in the context of real-time systems for CPUs scheduling [31, 1, 8] and applied to share other computational resources like programmable GPUs [23, 22]. Essentially, the idea is assigning to each entity (e.g., task) a fraction of a shared resource under contention (e.g., processor) in order to provide temporal isolation. Similarly, this work adjusts the same approach to the contention of the AMBA AXI bus in the context of hardware-programmable SoC FPGA platforms.

Many research efforts have been dedicated to the problem of bus contention in real-time systems. Schliecker et al. [33] use an event-based model to estimate delays for communications and computation activities on a multicore SoC platform. Pellizzoni and Caccamo [29] analyzed the interaction between CPU and peripherals while contending a shared main memory within a theoretical framework and proposed a conceptual solution based on a hardware server to control the unpredictable behavior of COTS peripherals. Betti et al. [6] presented a

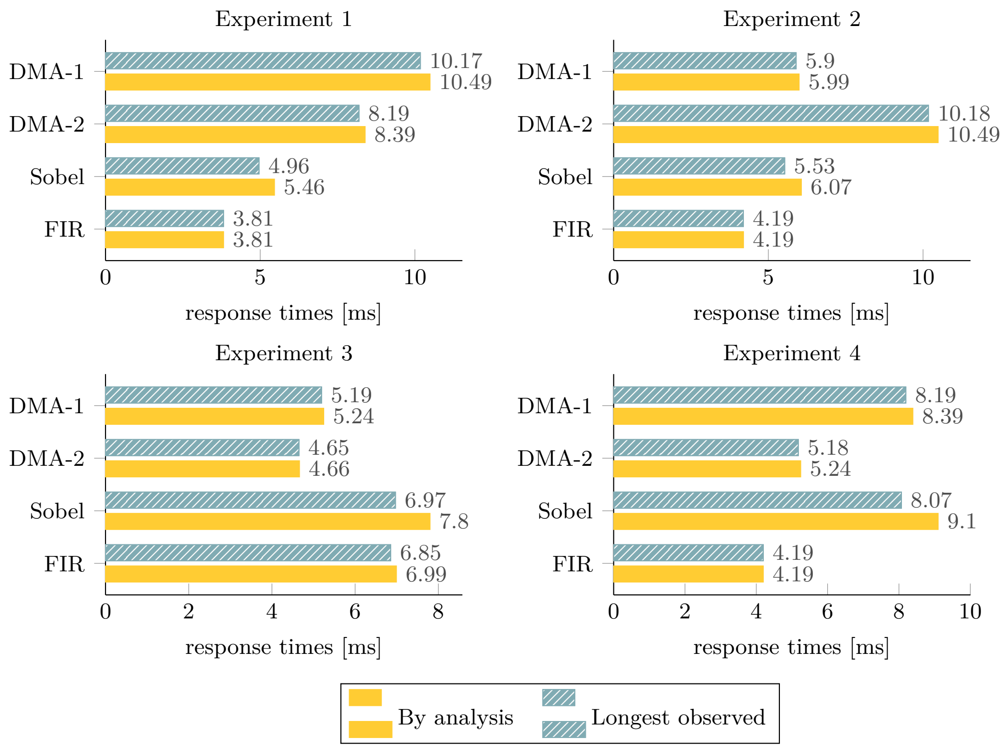


Figure 8 Response times for the case study.

framework for providing real-time guarantees in a COTS platform. Each peripheral within the platform is supervised by a "real-time bridge" controlled by a system-wide peripheral scheduler. Their framework has been developed and evaluated on PC platforms with PCI Express bus while our approach considers on-chip buses for integrated SoC-FPGA platforms.

In the context of memory contention on multicore platforms, Agrawal et al. [2] presented a technique to perform the analysis both WCETs and schedulability of real-time activities under dynamic memory scheduling. Yum et al. [47] proposed a memory bandwidth reservation mechanism named MemGuard. The system provides memory performance isolation employing a bandwidth regulator for each core. The bandwidth regulators enforce a budgeting mechanism and are implemented using performance counters. Our approach is somehow related to this work since both consider bandwidth regulation of bus master agents. However, while MemGuard considers inter-core interference on an Intel chip multiprocessor, our work considers bus interference generated by hardware accelerators on the AMBA AXI bus.

In the domain of packet switching networks, many efforts have been dedicated to the modeling and the analysis of traffic scheduling algorithms to provide quality of service (QoS) guarantees [15, 37]. Such methodologies have also been employed on SoCs platforms to develop and analyze arbiters for heavily-contented resources like the system memory [3, 17]. The ABU can be improved by leveraging the results of these works. Concerning the development of on-chip communication infrastructures for SoC platforms, transaction-based buses and packet-based networks on chip (NoC) remain the dominant approaches [32]. Typically, arbitration for on-chip interconnects is performed using Fixed Priority, Round Robin, and Time-Division Multiple Access (TDMA). Poletti et al. presented a performance analysis comparing different arbitration policies for SoCs platforms in [30]. A TDMA-based arbitration scheme with dynamic timeslot allocation is employed in [32, 10] to improve system predictability while providing good average-case performance. Lahiri et al. [24] proposed a statistical approach to arbitration using a ticket-based random selection which was further extended by other works [12, 25] to improve predictability. Steine et al. [36] proposed a TDMA budget based scheduler for data flow applications, which has been used by Staschulat et al. [35] for memory arbitration. However, while the latter work is explicitly targeted at embedded systems, it is still limited to dataflow applications. Bourgade [9] proposed a bus arbitration scheme for multicore platforms designed to ease the estimation of the tasks' worst-case execution times. Reconfigurable bus arbiters [46, 34] can be dynamically configured to change the arbitration policy depending on the application requirements. Likewise, several papers in the literature addressed the problem of designing predictable memory controllers for multi-core architectures. Guo et al. [18] presented a comparative analysis of predictable DRAM controllers.

8 Conclusions

This paper presented the ABU, a hardware-based reservation mechanism for the AMBA AXI bus aimed at isolating hardware accelerators implemented on FPGAs. After describing the internal architecture of the ABU, a response-time in the bandwidth domain has been presented to verify the schedulability of a set of hardware accelerators under real-time constraints. The proposed mechanism has been implemented and validated on the Xilinx Zynq-7020 platform to demonstrate its practical applicability. An substantial experimental evaluation confirmed the effectiveness of the proposed solution, showing that it can efficiently be implemented by consuming less than 1% of the total FPGA resources. As a future work, we plan to evaluate the possibility of including a reclaiming mechanism for the unused supply and extend the analysis to support for dynamic workloads by taking advantage of partial reconfiguration [7, 27].

References

1 Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Real-Time Systems Symposium, 1998. Proceedings. The 19th IEEE*, pages 4–13. IEEE, 1998.

2 Ankit Agrawal, Renato Mancuso, Rodolfo Pellizzoni, and Gerhard Fohler. Analysis of Dynamic Memory Bandwidth Regulation in Multi-core Real-Time Systems. In *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, December 2018.

3 Benny Akesson, Liesbeth Steffens, and Kees Goossens. Efficient service allocation in hardware using credit-controlled static-priority arbitration. In *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 59–68. IEEE, 2009.

4 James H. Anderson, Philip Holman, and Anand Srinivasan. Fair Scheduling of Real-Time Tasks on Multiprocessors. In *Handbook of Scheduling - Algorithms, Models, and Performance Analysis*. Chapman and Hall/CRC, 2004.

5 ARM. *AMBA AXI and ACE Protocol Specification*, 2011.

6 E. Betti, S. Bak, R. Pellizzoni, M. Caccamo, and L. Sha. Real-Time I/O Management System with COTS Peripherals. *IEEE Transactions on Computers*, 62(1):45–58, January 2013. doi:10.1109/TC.2011.202.

7 Alessandro Biondi, Alessio Balsini, Marco Pagani, Enrico Rossi, Mauro Marinoni, and Giorgio Buttazzo. A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs. In *Real-Time Systems Symposium (RTSS)*, pages 1–12, 2016.

8 Alessandro Biondi, Alessandra Melani, and Marko Bertogna. Hard constant bandwidth server: Comprehensive formulation and critical scenarios. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, pages 29–37. IEEE, 2014.

9 Roman Bourgade, Christine Rochange, and Pascal Sainrat. Predictable bus arbitration schemes for heterogeneous time-critical workloads running on multicore processors. In *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*, pages 1–4. IEEE, 2011.

10 Paolo Burgio, Martino Ruggiero, Francesco Esposito, Mauro Marinoni, Giorgio Buttazzo, and Luca Benini. Adaptive TDMA bus allocation and elastic scheduling: A unified approach for enhancing robustness in multi-core RT systems. In *Computer Design (ICCD), 2010 IEEE International Conference on*, pages 187–194. IEEE, 2010.

11 Andrew Canis, Jongsok Choi, Blair Fort, Ruolong Lian, Qijing Huang, Nazanin Calagar, Marcel Gort, Jia Jun Qin, Mark Aldham, Tomasz Czajkowski, et al. From software to accelerators with legup high-level synthesis. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, page 18. IEEE Press, 2013.

12 Chien-Hua Chen, Geeng-Wei Lee, Juinn-Dar Huang, and Jing-Yang Jou. A real-time and bandwidth guaranteed arbitration algorithm for SoC bus communication. In *Design Automation, 2006. Asia and South Pacific Conference on*, pages 6–pp. IEEE, 2006.

13 Eric S Chung, Peter A Milder, James C Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 225–236. IEEE Computer Society, 2010.

14 Ben Cope, Peter YK Cheung, Wayne Luk, and Lee Howes. Performance comparison of graphics processors to reconfigurable logic: A case study. *IEEE Transactions on computers*, 59(4):433–448, 2010.

15 Rene L Cruz et al. A calculus for network delay, part I: Network elements in isolation. *IEEE Transactions on information theory*, 37(1):114–131, 1991.

16 Robert I. Davis and Alan Burns. A Survey of Hard Real-time Scheduling for Multiprocessor Systems. *ACM Comput. Surv.*, 43(4), 2011.

17 Manil Dev Gomony, Jamie Garside, Benny Akesson, Neil Audsley, and Kees Goossens. A globally arbitrated memory tree for mixed-time-criticality systems. *IEEE Transactions on Computers*, 66(2):212–225, 2017.

18 Danlu Guo, Mohamed Hassan, Rodolfo Pellizzoni, and Hiren Patel. A comparative study of predictable dram controllers. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(2):53, 2018.

19 Dominik Honegger, Helen Oleynikova, and Marc Pollefeys. Real-time and low latency embedded computer vision hardware based on a combination of fpga and mobile cpu. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 4930–4935. IEEE, 2014.

20 Intel. *Stratix 10 GX/SX Device Overview*, October 2017.

21 Jan Moritz Joseph, Morten Mey, Kristian Ehlers, Christopher Blochwitz, Tobias Winker, and Thilo Pionteck. Design space exploration for a hardware-accelerated embedded real-time pose estimation using vivado HLS. In *ReConFigurable Computing and FPGAs (ReConFig), 2017 International Conference on*, pages 1–8. IEEE, 2017.

22 Shinpei Kato, Karthik Lakshmanan, Yutaka Ishikawa, and Ragunathan Rajkumar. Resource sharing in GPU-accelerated windowing systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 191–200. IEEE, 2011.

23 Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*, pages 17–30, 2011.

24 Kanishka Lahiri, Anand Raghunathan, and Ganesh Lakshminarayana. The LOTTERYBUS on-chip communication architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(6):596–608, 2006.

25 Bu-Ching Lin, Geeng-Wei Lee, Juinn-Dar Huang, and Jing-Yang Jou. A precise bandwidth control arbitration algorithm for hard real-time SoC buses. In *Proceedings of the 2007 Asia*

and South Pacific Design Automation Conference, pages 165–170. IEEE Computer Society, 2007.

26 Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, et al. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016.

27 Marco Pagani, Alessio Balsini, Alessandro Biondi, Mauro Marinoni, and Giorgio Buttazzo. A linux-based support for developing real-time applications on heterogeneous platforms with dynamic fpga reconfiguration. In *2017 30th IEEE International System-on-Chip Conference (SOCC)*, pages 96–101. IEEE, 2017.

28 Karl Pauwels, Matteo Tomasi, Javier Diaz Alonso, Eduardo Ros, and Marc M Van Hulle. A comparison of FPGA and GPU for real-time phase-based optical flow, stereo, and local image features. *IEEE Transactions on Computers*, 61(7):999–1012, 2012.

29 R. Pellizzoni and M. Caccamo. Impact of Peripheral-Processor Interference on WCET Analysis of Real-Time Embedded Systems. *IEEE Transactions on Computers*, 59(3):400–415, March 2010. `doi:10.1109/TC.2009.156`.

30 Francesco Poletti, Davide Bertozzi, Luca Benini, and Alessandro Bogliolo. Performance analysis of arbitration policies for SoC communication architectures. *Design Automation for Embedded Systems*, 8(2-3):189–210, 2003.

31 Ragunathan Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Multimedia Computing and Networking 1998*, volume 3310, pages 150–165. International Society for Optics and Photonics, 1997.

32 Thomas D Richardson, Chrysostomos Nicopoulos, Dongkook Park, Vijaykrishnan Narayanan, Yuan Xie, Chita Das, and Vijay Degalahal. A hybrid SoC interconnect with dynamic TDMA-based transaction-less buses and on-chip networks. In *VLSI Design, 2006. Held jointly with 5th International Conference on Embedded Systems and Design., 19th International Conference on*, pages 8–pp. IEEE, 2006.

33 Simon Schliecker, Mircea Negrean, Gabriela Nicolescu, Pierre Paulin, and Rolf Ernst. Reliable performance analysis of a multicore multithreaded system-on-chip. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pages 161–166. ACM, 2008.

34 Éricles Sousa, Deepak Gangadharan, Frank Hannig, and Juergen Teich. Runtime reconfigurable bus arbitration for concurrent applications on heterogeneous MPSoC architectures. In *Digital System Design (DSD), 2014 17th Euromicro Conference on*, pages 74–81. IEEE, 2014.

35 Jan Staschulat and Marco Bekooij. Dataflow models for shared memory access latency analysis. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 275–284. ACM, 2009.

36 Marcel Steine, Marco Bekooij, and Maarten Wiggers. A priority-based budget scheduler with conservative dataflow model. In *Digital System Design, Architectures, Methods and Tools, 2009. DSD'09. 12th Euromicro Conference on*, pages 37–44. IEEE, 2009.

37 Dimitrios Stiliadis and Anujan Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Transactions on networking*, 6(5):611–624, 1998.

38 Stylianos I Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions. *ACM Computing Surveys (CSUR)*, 51(3):56, 2018.

39 Xilinx. *Zynq UltraScale+ Device - Technical Reference Manual*, December 2017. UG1085.

40 Xilinx Inc. *Using Quality of Service (QoS) Capabilities in Zynq-7000 AP SoC Devices*, July 2015. XAPP1266.

41 Xilinx Inc. *AXI Interconnect, LogiCORE IP Product Guide*, 2018. PG059.

42 Xilinx Inc. *Convolutional Encoder, LogiCORE IP Product Guide*, 2018. PG026.

43 Xilinx Inc. *Fast Fourier Transform, LogiCORE IP Product Guide*, 2018. PG109.

44 Xilinx Inc. *FIR Compiler, LogiCORE IP Product Guide*, 2018. PG149.

45 Xilinx Inc. *SmartConnect, LogiCORE IP Product Guide*, 2018. PG247.

46 Ching-Chien Yuan, Yu-Jung Huang, Shih-Jhe Lin, and Kai-hsiang Huang. A reconfigurable arbiter for SOC applications. In *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, pages 713–716. IEEE, 2008.

47 H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, April 2013.

Hiding Communication Delays in Contention-Free Execution for SPM-Based Multi-Core Architectures

Benjamin Rouxel
Univ Rennes, Inria, CNRS, IRISA, France
benjamin.rouxel@irisa.fr

Stefanos Skalistis
Univ Rennes, Inria, CNRS, IRISA, France
stefanos.skalistis@irisa.fr

Steven Derrien
Univ Rennes, Inria, CNRS, IRISA, France
steven.derrien@irisa.fr

Isabelle Puaut
Univ Rennes, Inria, CNRS, IRISA, France
isabelle.puaut@irisa.fr

Abstract

Multi-core systems using ScratchPad Memories (SPMs) are attractive architectures for executing time-critical embedded applications, because they provide both predictability and performance. In this paper, we propose a scheduling technique that jointly selects SPM contents off-line, in such a way that the cost of SPM loading/unloading is hidden. Communications are fragmented to augment hiding possibilities. Experimental results show the effectiveness of the proposed technique on streaming applications and synthetic task-graphs. The overlapping of communications with computations allows the length of generated schedules to be reduced by 4% on average on streaming applications, with a maximum of 16%, and by 8% on average for synthetic task graphs. We further show on a case study that generated schedules can be implemented with low overhead on a predictable multi-core architecture (Kalray MPPA).

2012 ACM Subject Classification Computer systems organization → Embedded and cyber-physical systems; Computer systems organization → Real-time systems

Keywords and phrases Real-time Systems, Contention-Free Scheduling, SPM multi-core architecture

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.25

Funding This work was partially supported by ARGO (`http://www.argo-project.eu/`), funded by the European Commission under Horizon 2020 Research and Innovation Action, Grant Agreement Number 688131.

1 Introduction

The race for computer performance has always been limited by the memory bottleneck. To overcome this issue, hardware [28], software [23] and hybrid [20] prefetching methods have been proposed in the past to bring data closer to the processor before it is needed. However, most prefetchers are not designed for time-critical applications, where predictability is essential.

Compared to cache-based architectures, multi-cores with a private ScratchPad Memory (SPM) per core are a very attractive alternative for time-critical embedded applications. Via software-managed SPMs, they offer sufficient computational power and the necessary predictability. Software-managed SPMs enable data-movement decisions, from/to main memory, to be scheduled at design time (off-line), thus restricting or avoiding contention on shared resources. Examples of such architectures are the Cell multi-core architecture [19], the academic core Patmos [32] or the Kalray MPPA [11].

Efficient and predictable management of SPMs are facilitated by application models that offer a high-level view of parallel programs. We focus on applications modelled as directed acyclic task-graphs (DAGs), consisting of dependent tasks that exchange data through shared FIFO channels. In such application models, tasks are executed in three phases: 1) they *read* data from their input FIFOs, 2) *execute* their computation, and 3) *write* the results to their output FIFOs. This order of execution is in accordance with the PRedictable Execution Model (PREM) [29, 2] and the Acquisition Execution Restitution (AER) execution model [26]. Such execution models are well-suited for SPM-based architectures, as tasks can prefetch their input FIFOs from the shared memory into the private SPM and, after the task's execution, write-back the produced data to their output FIFOs. Using proper scheduling techniques, this can result in contention-free execution. These DAGs do not necessarily need to be built from scratch, which would require an important engineering effort. Automatic extraction of parallelism, for instance from a high level description of applications in model based design workflows [12], seems a much more promizing direction.

We believe that this combination of software (DAG with PREM) and hardware (SPM-based multi-cores) is essential to build efficient and predictable systems. In this paper, we propose a scheduling strategy that hides such delays by executing communications in parallel with computation. Our scheduling strategy relies in advancing (resp. postponing) the execution of read (resp. write) phase of a task such that it overlaps with the execution phase of another task, thus hiding the communication delay. The proposed scheduling strategy aims at minimizing the makespan of the total execution and includes an SPM allocation strategy ensuring that there is enough space in SPM at all times. The resulting schedules are contention-free to the shared bus, similarly to [3]. Additionally, and in comparison with most related works (such as [30, 8, 24, 37]), we fragment communication phases to augment communication hiding possibilities. In contrast with most other works dealing with SPM, e.g. [13, 4], that allow some information to stay in global main memory, our SPM allocation scheme imposes that *all* information accessed by a task is prefetched into SPM beforehand. In summary, the contributions of this work are the following:

1. We propose a strategy to map and schedule a task graph onto cores coupled with an SPM allocation scheme. The generated static contention-free non-preemptive schedules allow, when possible, to overlap communications and computations, through *non-blocking* loading/unloading of information into/from SPM. Communication phases are *fragmented* to maximize the duration of overlapping between communications and computations. The proposed strategy is formulated as a heuristic based on list-scheduling to produce schedules very fast.
2. We provide an experimental evaluation showing our method improves the overall makespan, up to 16%, compared to equivalent schedules generated with blocking communications.
3. We evaluate the impact of different granularities for communication fragments on the schedule makespan.
4. We experimentally show on a use case that generated schedules can be implemented with a low overhead on a predictable multi-core architecture (Kalray MPPA [11]).

The rest of this paper details the proposed strategy and is organized as follows. A motivating example is presented in Section 2, as well as the assumptions made on the hardware and software. Then, Section 3 presents the basic principles of the SPM allocation scheme. The scheduling/mapping/allocation heuristic technique is then detailed in Section 4. Section 5 presents experimental results, including an implementation on the Kalray MPPA platform. Finally, Section 6 presents related works, before concluding in Section 7.

2 Motivating Example and assumptions

2.1 Architecture Model

We consider multi-core architectures, where every core has access to a private dual-ported ScratchPad memory (SPM). Cores are connected through an arbitrated bus to a global external shared memory. Access requests are enqueued (one queue per core) and served according to the bus arbitration policy. While in the rest of the paper, we will assume a FAIR-round-robin arbitration [21], the proposed method is directly applicable for other policies with arbitration based on requests (e.g. first-come-first-served, fixed priority, etc.) and not on time (e.g. time division multiple accesses). All communications are non-preemptable and go through the shared global memory (no SPM to SPM communication). We further assume that the architecture supports loading of information in the dual-ported SPM, in parallel with computations. Provided support may be a hardware Direct Memory Access (DMA) engine or a specific core acting as a DMA software engine as in [9]. These assumptions are met in both academic and commercial processors (e.g. Patmos [32], Kalray MPPA [11]).

Communications can be implemented in *blocking* mode or *non-blocking* mode. In *blocking* mode, the CPU is in charge of transfers between SPM and the shared memory, and is then stalled during every transfer. In *non-blocking* mode, transfers are managed asynchronously, allowing the CPU to execute other jobs during memory transfers.

2.2 Application Model

We consider applications modeled as directed acyclic graphs (DAGs). A graph G is a pair (V, E) where the vertices in V represent the application's tasks and the edges in E represents the data dependencies between the tasks. This work supports multiple DAGs with same period as is, which is omitted due to space limitations. Extending our work to applications with different periods is deemed as a rather direct transposition, by making schedule generation operate on the hyperperiod. This extension is however left for future work.

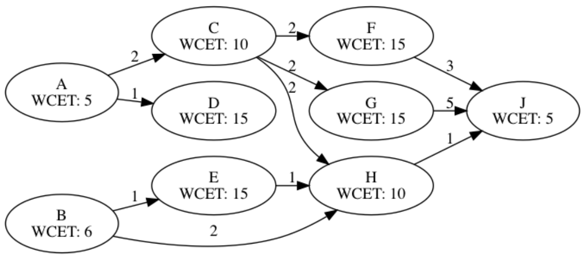


Figure 1 An example of a task-graph.

According to the semantics defined in PREM [29, 1] or AER [26], each task is divided in three phases, namely *read*, *exec* and *write*. The *read* phase reads/receives the mandatory code and/or data from the main memory to the SPM, such that the *exec* phase can proceed without access to the shared bus. Finally, the *write* phase writes the resulting data back from the SPM to the main memory. Using such an application and execution model is central in our method, as it allows to perform offline scheduling which precisely controls resource contention. The *exec* phase of tasks does not access the shared bus, and thus contentions when accessing the shared bus do not exist between *exec* phases and *read/write* phases; the off-line scheduler is in charge of scheduling communication phases in such a way that they do not conflict with one another; finally, the presence of a dual-ported SPM per core allows calculations and communications to proceed in parallel, provided that they access different address ranges.

Note that considered DAGs with *read-exec-write* semantics need not be built from scratch. They can be extracted automatically either from a high-level description of applications in model based design workflows [12], or from legacy code with [27].

As an extension to the original PREM/AER model, we split each communication into *fragments*. A *fragment* is some division of the total amount of data that a task produces or consumes. How the data are divided into fragments is determined by the fragmentation scheme. The default fragmentation scheme assumed throughout this paper is to have one fragment for each task communication (edges in the graph). Thus, instead of a task reading/writing all of its inputs/outputs at once, it is done on a per-task basis with the size of the fragment being as the size of the communication. Other fragmentation strategies will be detailed in Section 5.4. A task τ_i is a tuple $\tau_i =< F_i^r, \tau_i^e, F_i^w >$, where τ_i^e is the *exec* phase, and F_i^r (resp. F_i^w) is the set of fragments read (resp. written) by the task. The f-th fragment of τ_i that is read (resp. written) is denoted as $\tau_{(i,f)}^r \in F_i^r$ (resp. $\tau_{(i,f)}^w \in F_i^w$).

An example of a task-graph is illustrated in Figure 1. The figure gives for each task its name, the Worst Case Execution Time (WCET) of its *exec* phase, and for each edge the amount of data exchanged, among the tasks, in bytes. The WCET of the *exec* phase, denoted C_i, can be estimated in isolation from the other tasks considering a single-core architecture, as there is no access to the main memory (all the required data and code have been loaded into the SPM before the task's execution). In general, *read* and *write* fragments could suffer from contentions caused by concurrent accesses to the shared bus, however in this paper the proposed technique produces contention-free schedules.

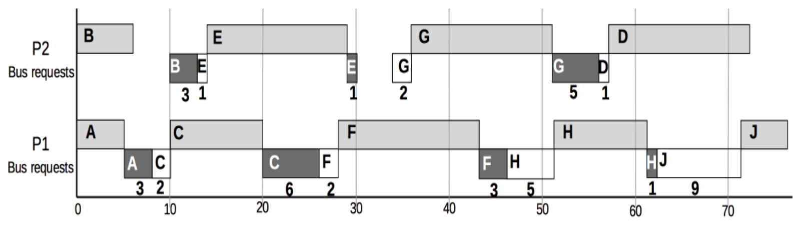
Since the code in our experimental evaluation, is generally small and likely to be reused along the execution of the application, for simplicity reasons we assume that the code is preloaded in the SPM at startup.

For simplicity when presenting the motivational example, we will assume the SPM to be large enough to store all information (code, data, communication buffers), this assumption will be relaxed in Section 3.

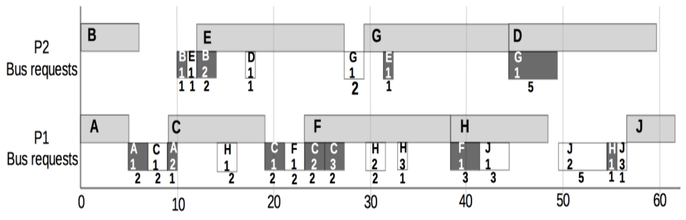
2.3 Motivating Example

Figure 2 motivates the use of *non-blocking, fragmented* communications for the application from Figure 1 assuming a dual-core architecture. Sub-figure 2b depicts the schedule obtained using non-blocking fragmented communications, with one fragment per outgoing edge in the graph, whereas sub-figure 2a depicts the schedule obtained using blocking communications. For each core, the top time-line depicts the scheduling of *exec* phases (grey boxes) and the bottom one depicts the scheduling of communications (*read*: white boxes, black font, *write*: dark boxes, white font). The communication cost is indicated below each communication phase.

In Figure 2a (*blocking* mode), all parts of the same task are scheduled contiguously on the same core, and the CPU is stalled when accessing the bus. The *read* and *write* phases are not fragmented as it would not bring any benefit in blocking mode. Precedence constraints are respected by ordering *read* phases after their preceding *write* phases, e.g. τ_C^r is scheduled after the completion of τ_A^w. There is no *read* phase for tasks A and B as they do not have predecessors, hence no data to fetch. The resulting schedule makespan (time at which the last task ends) is 76 time units. In Figure 2b (*non-blocking* mode), fragmented communication and *exec* phases overlap, e.g. $\tau_{(H,2)}^r$ and $\tau_{(H,3)}^r$ overlap with τ_F^e, thus hiding the communication delay. Having prefetched all required data into SPM, the *exec* phase of τ_H can start right after τ_F.



(a) Schedule in *blocking* mode (makespan of 76 time units).



(b) Schedule in *non-blocking* mode (makespan of 61 time units).

Figure 2 Schedules for the example task-graph on a dual-core. For each core, the top time-line depicts the schedule of *exec* phases (in grey), the bottom one depicts the schedule of *read* (in white) and write (in black) phases. The communication cost is indicated below each communication phase.

The gain in schedule length from Figure 2b is obtained by introducing the following flexibilities in the scheduling of communication fragments, while respecting *read-exec-write* phase's order: 1) communication phases of different tasks can have a different order than their respective *exec* phases, as long as there is no data dependencies between them, e.g. $\tau^r_{(D,1)}$ is scheduled before $\tau^w_{(E,1)}$, in reverse order compared to τ^e_D and τ^e_E. 2) communication phases and *exec* phase of the same task, do not need to be contiguous in time, e.g. $\tau^r_{(D,1)}$ and τ^e_D are not. 3) communications are fragmented, a task with multiple successors does not write all its data at once, e.g. τ_B has two successors, thus writing two fragments.

The last point (fragmented communications) is new compared to related work. Considering each fragment individually allows additional overlaps between communications and task execution that were impossible without fragmentation. In the example, it allows to hide part of the *write* phase of τ_A, and part of the *read* phase of τ_J, which was not possible without fragmentation. Thus, splitting communications allows each source/sink of the task graph to hide part of its communications. However, in the example from Figure 2b, the remaining first part A^w ($\tau^w_{(A,1)}$) can still not be hidden, but is however smaller than in Figure 2a. The overall makespan of the resulting schedule in *non-blocking* mode (Figure 2b) is 61 time units, resulting in a gain of 20%.

3 Principle of SPM allocation scheme

In our motivational example, we assumed the SPM large enough to store all information required to execute the entire application (code, data, communication buffers). To account for limited SPM capacity, our scheduling strategy comes with a SPM allocation strategy that allocates an SPM area (called hereafter *region*) to each communication fragment and execution phase. *Fragment-to-region mapping* is performed by the scheduler off-line. However, the same region can be used successively by different fragments, and the scheduler guarantees that the live ranges of the concerned fragments do not overlap. Region sizes vary according to the data stored by fragments/exec phases.

To isolate bus accesses from computation, we impose that *all* information accessed by a task is loaded into SPM beforehand. This comes in opposition to most SPM allocation policies that decide which information should be stored in the SPM and which information should remain in the global main memory (e.g. [13]). Our fragment-to-region mapping is inspired by the method proposed in [22].

The regions assigned to fragments F_i^r contain the input data, fetched from the main memory, which are required by the task's *exec* phase. These regions contain the data produced by all predecessor tasks. The unique region assigned to τ_i^e contains any kind of information used locally by the task (code, constants, local data, usually stack-allocated). The regions assigned to F_i^w contain the data produced by the task.

The size of a region obviously depends on the amount of data required by the associated fragment (i.e. amount of data produced by a predecessor in case of a *read* fragment). Considering a mapping of tasks to cores and a mapping of fragments to SPM regions, the sum of the sizes of regions on a core must not exceed the SPM size.

Let us consider the example of Figure 2b, in which for simplicity we concentrate on the communication fragments and ignore the execution phases. If the size of the SPM is 1 Kbytes then on processor $P2$ the SPM can be partitioned in seven regions $SPM = \{\tau^w_{(B,1)}, \tau^w_{(B,2)}, \tau^r_{(E,1)}, \tau^w_{(E,1)}, \tau^r_{(D,1)}, \tau^r_{(G,1)}, \tau^w_{(G,1)}\}$ with respective sizes in bytes $\{1, 2, 1, 1, 1, 2, 5\}$ (according to the amount of data exchanged between tasks, taken from Figure 1). The sum of the regions' sizes is 13 bytes, which is less than the SPM size. If we now restrict the SPM size to 10 bytes, the previous partitioning of SPM in regions is not valid anymore. However, once $\tau^w_{(B,1)}$ is completed, the data produced by τ_B has been committed to the global shared memory, therefore its assigned region can be reused. In this example, $\tau^r_{(G,1)}$ starts after the completion of $\tau^w_{(B,1)}$, as it is the case for $\tau^w_{(B,2)}$ and $\tau^w_{(G,1)}$. Thus, the fragments $\tau^w_{(B,1)}$,$\tau^r_{(G,1)}$ and $\tau^w_{(B,2)}$,$\tau^w_{(G,1)}$ can be assigned to the same SPM region, leaving a partitioning of five regions: $SPM = \{\{\tau^w_{(B,1)}, \tau^r_{(G,1)}\}, \{\tau^w_{(B,2)}, \tau^w_{(G,1)}\}, \tau^r_{(E,1)}, \tau^w_{(E,1)}, \tau^r_{(D,1)}\}$ with respective sizes (in bytes) $\{max(1, 2), max(2, 5), 1, 1, 1\}$. The sum of all regions sizes is 10 bytes, which can fit in the SPM.

In the example, both pairs $(\tau^w_{(B,1)}, \tau^r_{(G,1)})$ and $(\tau^w_{(B,2)}, \tau^w_{(G,1)})$ could share the same region, because their lifespan does not overlap. On the other hand, in Figure 2b, $\tau^r_{(D,1)}$ can not share the same region as $\tau^r_{(E,1)}$, because the data consumed by τ_E are in use from the start of the read phase F_E^r up to the end of the execution of τ_E^e. This leads to define the *live range* of regions for each type of fragment. Definition 1 defines the live range for a region assigned to a *read* fragment, while Definitions 2 and 3 give live ranges for regions assigned respectively to an *exec* and a *write* fragment.

▶ **Definition 1.** *Data fetched from the main memory by a read fragment are alive from its start time to the end of the corresponding exec phase.*

▶ **Definition 2.** *Local information used by an exec phase (code, stack data area) are alive for the whole execution time of the application.*

▶ **Definition 3.** *Data written back to main memory by a write fragment are alive from the start time of the corresponding exec phase to its transmission end time.*

We assume read/written data can be consumed/produced at any time in the *exec* phase of the task. Therefore, the live range in Definitions 1 and 3 include the duration of the *exec* phase.

The scheduler maps fragments to regions, but does not decide the addresses of the regions in the SPM, which is left to the compiler/code generator. Since the number and size of regions is decided off-line, address assignment is straightforward, and does cause *external* fragmentation. Fragmentation of the SPM can only arise inside regions (*internal* fragmentation) when two (or more) phases are assigned to the same region but store different amounts of data.

4 Joint-mapping/scheduling and SPM allocation

This section presents a heuristic algorithm based on forward list scheduling that integrates *fragmented non-blocking* communication and *SPM allocation.* The main outcome of the proposed algorithm is a static mapping, scheduling and SPM fragment-to-region allocation, for a single application represented as a DAG. The objective is minimizing the overall schedule's makespan. The generated schedule is free from contention. According to the terminology given in [10], the proposed scheduling techniques are partitioned, time-triggered and non-preemptive. Schedule generation operates at *task-level* (as opposed to job-level as defined in [10]).

Heuristics based on forward list scheduling first order input elements (in our specific case *exec* and *communication* phases), then add them one by one in the schedule without backtracking. We experimented with three topological sorting algorithms. The first algorithm is a vanilla Depth First Search (DFS) algorithm to walk-through the task graph. Second, we use the same DFS algorithm but we postpone *read* fragments to avoid too early reading that might delay other fragments in the schedule (further details will be given when describing Algorithm 4.1). The last algorithm is a vanilla Breath First Search (BFS) algorithm. For all three sortings, we used the element memory footprint as tie breaking rule (larger footprint to be scheduled first). Since no sorting algorithm consistently outperforms the others, we generate three schedules, each resulting from one sorting algorithm, and selected the one resulting in the shortest schedule makespan as the heuristic's solution.

4.1 Notations and assumptions

Table 1 summarizes the notations that will be used to describe the scheduling algorithm (sets, utility functions and constants).

Calculation of constants $DELAY^{r}_{(i,f)}$ and $DELAY^{w}_{(i,f)}$ requires knowledge of the bus arbitration strategy and of concurrent accesses to the bus. The considered bus is characterized by a maximum duration of T_{slot} allocated to each core in a round-robin fashion, with a writing rate of D_{slot} data word per time unit. T_{slot} defines the duration a core is granted the bus, and D_{slot} defines the amount of data transmittable in a T_{slot} duration. For the scope of this paper, we generate contention-free schedules, thus no contention delay is paid, and the duration of a data transfer of d bytes is trivially calculated by equation (1). This equation could be refined to account for DRAM access cost, as done in [22].

$$delay = \lceil d/D_{slot} \rceil \cdot T_{slot} \tag{1}$$

In the description of the scheduling algorithm, the cost for setting up non-blocking memory transfers (DMA initialization in case of a hardware DMA engine) will not appear explicitly and is considered included in the WCET of the *exec phase.* Determination of this cost will be described in Section 5.5.

4.2 Scheduling algorithm

The scheduling algorithm is sketched in Algorithm 4.1. It uses the task graph as input, sorts the elements to schedule (*exec* phases and *communication* fragments) to create the list (line 2). Then a loop iterates on each element while there exists elements to schedule

Table 1 Notations.

Sets	T	set of tasks
	P	set of processors/cores
	R	set of regions
	F_i^r, F_i^w	sets of τ_i fragments
	$F = F_i^r \cup F_i^w, \forall i \in T$	sets of all fragments from all tasks in T
Funcs	$i = task(f)$	utility to retrieve the task of a fragment, fragment f belongs to τ_i
	$(j,q) \in pred((i,f))$	(j,q) means $\tau^X_{(j,q)}$ is a direct predecessor of $\tau^X_{(i,f)}$
Constants	SS_i	local (stack) data size of τ_i^e
	CS_i	code size of τ_i^e
	C_i	τ_i execute phase WCET computed in isolation as stated in Section 2
	$D^r_{(i,f)}, D^w_{(i,f)}$	size in bytes of $\tau^r_{(i,f)}$
	$DELAY^r_{(i,f)}$ $DELAY^w_{(i,f)}$	fragment f of τ_i, *read/write* latency from Equation (1)
	$SPMSIZE_c$	SPM size of core c
Variables	$\rho^r_{(i,p)}$, ρ^e_i, $\rho^w_{(i,q)}$	start times of $\tau^r_{(i,p)}$, τ^e_i and $\tau^w_{(i,q)}$

(lines 5-20). This heuristic uses an *As Soon As Possible* (ASAP) strategy when mapping an element. If the element to schedule is a communication fragment (line 8), then there is no need to map it on a core, but it still must be scheduled to avoid interference. If it is an *exec* phase, then a core is selected and the mapping with the shorter the makespan is selected (line 15).

SPM regions can be assigned to elements (*exec* phases and *communication* fragments) only when all of its phases are properly scheduled and mapped to a core (lines 18-20). When scheduling the read fragments, the core mapping information is not yet available. Additionally, when mapping the *exec* phase, we still do not have the information regarding the *write* fragments that have not been scheduled yet. While assigning the region (lines 18-20), the *exec* phase goes first then the communication phases. This order is motivated to better handle resident code in SPM and avoid SPM space to be stolen by communication fragments. For example, if there are 5 units of free space (not assigned yet) and the *exec* needs 5 units while a *read/write* need 2 units each. Then the task can still be mapped. The *exec* phase will take the remaining free space, while the communication fragments can share an already created, but available (in time), region (see Definitions 1 and 3).

Scheduling an element

Algorithm 4.2 sketches the method to determine the start time of the considered element (*exec* phase or *communication* fragment). First, each element must start after its causal predecessors (line 2) Then, lines 3-9 enforce that no *exec* phases overlap on the same core and no fragments overlap on the bus. Condition at line 4 enforces the type of *cur_elt* and e to be identical, and if both are exec phases then they must be mapped on the same core. Finally, line 9 postpones *cur_elt* start time if overlapping with e.

Algorithm 4.1: Scheduling algorithm.

```
Input  : A task graph G and a set of processors
Output : A schedule
Function ListSchedule($G = (T, E), P$)
    $Qready \leftarrow$ TOPOLOGICAL_SORT_ELEMENTS($G$)
    $Qdone \leftarrow \emptyset$
    $schedule \leftarrow \emptyset$
    while $elt \in Qready$ do
        $Qdone \leftarrow Qdone \cup \{elt\}$
        $Qready \leftarrow Qready \backslash \{elt\}$
        /* $tmpSched$ contain the best schedule for the current task */
        if $elt$ is a read fragment $\vee$ $elt$ is a write fragment then
            SCHEDULE_ELEMENT($Qdone, elt, null$)
        else if $elt$ is an exec phase then
            $tmpSched \leftarrow \emptyset$ with $makespan = \infty$
            foreach $p \in P$ do
                $copy \leftarrow schedule$
                /* Set $\tau^e$ in $copy$ on $p$ the earliest in the schedule */
                SCHEDULE_ELEMENT($Qdone, elt, p$)
                $tmpSched \leftarrow min_{makespan}(tmpSched, copy)$
            $schedule \leftarrow tmpSched$
        if all fragments and exec phase of $\tau_i$ containing $elt$ are in $Qdone$ then
            ASSIGN_REGION($schedule, Qdone, \tau_i^e, SS_t + CS_t, 0, infinity$)
            $\forall f \in F_i^r$, ASSIGN_REGION($schedule, Qdone, f, D^r_{(i,f)}, \rho^r_{(i,f)}, \rho_i^e + C_i$)
            $\forall f \in F_i^w$, ASSIGN_REGION($schedule, Qdone, f, \rho_i^e, \rho^w_{(i,f)} + DELAY^w_{(i,f)}$)
    return $schedule$
```

Allocation of SPM regions

Algorithm 4.3 associates a SPM region to an element (*exec* phase, fragment). If there is data to store in the SPM (line 2), then it first tries to reuse an existing region (lines 4-6), thus minimizing the required memory size. If no existing region can be shared, then a new one is created (lines 7-8). Sharing a region imposes that the selected region is big enough to handle the current amount of data and free for use at the required time interval (line 4).

5 Experimental evaluation

The first presented experiments (Section 5.2) aim at validating the quality of the proposed scheduling technique as compared to a scheduling strategy based on Integer Linear Programming (ILP, see Section 5.1) that provides the optimal solution (shortest schedule makespan). Then, we validate the benefits of hiding communications using the heuristic technique (Section 5.3). In the above-mentioned experiments, the default fragmentation strategy (one fragment per edge in the task graph) is used. We subsequently compare different ways to fragment communications (Section 5.4). Finally, we show in Section 5.5 on a case study that generated schedules can be implemented with low overhead on a Kalray MPPA platform [11]. In Sections 5.2 to 5.4, scheduler and communication implementation overheads are neglected, but they are considered in Section 5.5.

Experiments have been conducted both on real code, in the form of the open-source Refactored StreamIT benchmark suite STR2RTS [31] and on synthetic task graphs, generated using Task-Graph For Free (TGFF) [14].

Algorithm 4.2: Scheduling of an element (exec, fragment).

```
Input  : the list of scheduled element, the current element to schedule, the current core or
         null if the element is a fragment
Output :
1 Function Schedule_Element($Qdone, cur\_elt, cur\_proc$)
      /* wct → Worst-Case Timing, $DELAY_\beta^\alpha$ or $C_\beta$                    */
      /* X and Y depend on the type of the corresponding element                       */
2     $\rho^X_{cur\_elt} \leftarrow max_{p \in pred(cur\_elt)}(\rho^Y_p + wct_p)$
3     foreach $e \in Qdone$ do
4         if cur_elt is a fragment and e is not a fragment
5         ∨ cur_elt is an exec phase and e is not an exec phase
6         ∨ cur_elt is an exec phase and e is not mapped on core cur_proc then
7             continue
8         if e overlaps in time with cur_elt then
9             $\rho^X_{cur\_elt} \leftarrow \rho^Y_e + wct_e$
```

Algorithm 4.3: Allocation of a SPM region to a phase.

```
Input  : A schedule, the list of scheduled element, the current task and properties of the
         phase to map on a region
Output : A schedule
1  Function Assign_Region(schedule, Qdone, cur_elt, dataSize, start, end)
2      if $data == 0$ then return
3      $proc \leftarrow getCore(schedule, cur\_elt)$
       /* Get the set of existing regions on core proc where :  size ≥ dataSize ∧
          last reservation time ends before start                                    */
4      $existing \leftarrow getExistingRegions(schedule, proc, dataSize, start)$
5      if $existing \neq \emptyset$ then
6          Assign the smallest existing region to cur_elt
7      else if free SPM size in proc ≥ dataSize then
           /* Create SPM region for cur_elt on proc with size data where the
              reservation time is [start; end]                                       */
8          $CreateRegion(cur\_elt, proc, dataSize, start, end)$
9      else
10         Throw Unschedulable
```

The STR2RTS applications[1] are modeled using fork-join graphs and come with timing estimates for each task and amount of data exchanged between them. We did not use all the benchmarks and applications provided in the suite as some are not parallel, they are made of a linear chain of tasks (i.e. CFAR, FIR, ComplexFIR, FTT6), making them uninteresting for multi-core platforms. This leaves us 18 benchmarks with 73 tasks in average and average memory footprint of 4 KB.

The synthetic task-graphs were generated with the latest version of the TGFF generation software. Generated task-graphs include chains of tasks with different lengths and widths, fork-join graphs and more evolved structures (e.g. multi-DAGs). The resulting task graph characteristics are presented in Table 2. The table includes the number of task-graphs, their

[1] A table describing each used benchmark is available in the appendix.

number of tasks, the maximum width of the task-graph, the range of WCET values for each task and the range of amount of exchanged data in bytes between pairs of tasks, the range of code size and stack size for each task, and the global ratio of WCET per amount of exchanged data. The TGFF parameters (average and indicator of variability) are set in such a way that the average values for task WCETs and volume of data exchanged between pairs of tasks correspond to the analogous average values for the STR2RTS benchmarks.

Table 2 Task-graph characteristics for synthetic task-graphs.

#Task-graphs	50
#Tasks	5, 69, 22
Max. width	3, 17,8
Exchanged data	$[0; 192]$

WCET	$[5; 6000[$
Code size	$[3; 3920[$
Local size	$[1; 60]$
Ratio $\frac{WCET}{data}$	10

All reported experiments have been conducted on several nodes from an heterogeneous computing grid with 138 computing nodes (1700 cores). In all experiments, the duration a core is granted the bus (T_{slot}) is set to 3 as in [21] and shown in [30] to have little impact on the schedule length. The transfer rate is one word (4 bytes) per time unit.

5.1 Baseline: Integer Linear Programming scheduling

An Integer Linear Programming (ILP) formulation consists of a set of integer variables, a set of constraints and an objective function. Constraints describe the problem to solve in the form of linear inequalities. Solving a problem consists in finding a valuation for each variable satisfying all constraints with the goal of minimizing/maximizing the objective function. Table 3 summarizes the variables used in the ILP formulation. For a concise presentation of constraints, the two logical operators $\vee, \wedge$ are directly used in the text of constraints. These operators can be transformed into linear constraints in order to properly use ILP solvers using simple transformation rules from [5].

Objective function

The objective is to obtain the shortest schedule, and so to minimize the makespan Θ, Equation (2a). Equation (2b) constrains the completion time of all tasks (starting of all *write* fragment $\rho^w_{(i,f)}$, plus its latency $DELAY^w_{(i,f)}$) to be inferior or equal to the schedule makespan.

$$minimize\ \Theta \tag{2a}$$

$$\forall i \in T; \forall f \in F^w_i; \rho^w_{(i,f)} + DELAY^w_{(i,f)} \leq \Theta \tag{2b}$$

Problem constraints

Some basic rules of a valid schedule are expressed in the following equations. Equation (3a) ensures the unicity of a task mapping ($p_{i,c} = 1$ τ_i is mapped on core c). Equation (3b) defines if two tasks are mapped on the same core ($m_{i,j} = 1$). When $a^{ee}_{i,j} = 1$ then τ^e_i is scheduled before τ^e_j, thus Equation (3d) forbids an order of phases (resp. fragments) and its reversed order to be both active but imposes to choose one; one of the $a^{ee}_{i,j}, a^{ee}_{j,i}$ must be equal to 1, but both can not be equal to 1. Equations (3e) unifies Equations (3b) and (3d) to order *exec* phases only on the same core. In Equation (3d), no equation enforces

Table 3 ILP variables.

Int. variables	Θ	schedule makespan
	$\rho^r_{(i,p)}$, ρ^e_i, $\rho^w_{(i,q)}$	start times of $\tau^r_{(i,p)}$, τ^e_i and $\tau^w_{(i,q)}$
	$spmsr^c_z$	computed size of SPM region z on core c
	$\sigma_{(i,f)}$, σ_i	*spm reservation* start times of $\tau^X_{(i,f)}$, τ^e_i
	$\omega_{(i,f)}$, ω_i	*spm reservation* end times of $\tau^X_{(i,f)}$, τ^e_i
Binary variables	$p_{i,c} = 1$	τ^e_i is mapped on core c
	$m_{i,j} = 1$	τ^e_i & τ^e_j are mapped on the same core
	$a^{ee}_{i,j} = 1$	τ^e_i is scheduled before τ^e_j ($\rho^e_i \leq \rho^e_j$)
	$a^{XY}_{(i,f),(j,g)} = 1$	$\tau^X_{(i,f)}$ is scheduled before $\tau^Y_{(j,g)}$, in the sense $\rho^X_{(i,f)} \leq \rho^Y_{(j,g)}$ $XY \in \{rr, ww, rw, wr\}$
	$am^{ee}_{i,j} = 1$	same as $a^{ee}_{i,j}$ but on the same core
	$am^{XY}_{i,j} = 1$	same as $a^{XY}_{i,j}$ but on the same core $XY \in \{rr, ww, rw, wr\}$
	$spmp_{z,i} = 1$ $spmp_{z,(i,f)} = 1$	τ^e_i is allocated to SPM region z $\tau^X_{(i,f)}$ is allocated to SPM region z
	$spmm_{(i,f),(j,g)} = 1$	$\tau^X_{(i,f)}$ and $\tau^X_{(j,g)}$ are assigned to the same region (similar to $m_{i,j}$)
	$spma_{(i,f),(j,g)} = 1$	$\tau^X_{(i,f)}$ is causally before $\tau^X_{(j,g)}$ (similar to $a_{i,j}$)
	$spmam_{(i,f),(j,g)} = 1$	$\tau^X_{(i,f)}$ is causally before $\tau^X_{(j,g)}$, and both are assigned to the same region (similar to $am_{i,j}$)

to have the same ordering for *exec* phases as for with *read* phases, because the solver does not have to chose an order between them (see Section 2). The same remark applies to *exec* phases and *write* phases.

$\forall (i,j) \in T \times T; XY \in \{rr, ww, rw, wr\}; \forall f \in F^X_i; \forall g \in F^Y_j; i \neq j$

$$\sum_{c \in P} p_{i,c} \qquad = 1 \tag{3a}$$

$$m_{i,j} = \sum_{c \in P} (p_{i,c} \wedge p_{j,c}) \quad \textit{and } m_{i,j} \qquad = m_{j,i} \tag{3b}$$

$$a^{ee}_{i,j} + a^{ee}_{j,i} \qquad = 1 \tag{3c}$$

$$a^{XY}_{(i,f),(j,g)} + a^{XY}_{(j,g),(i,f)} \qquad = 1 \tag{3d}$$

$$am^{ee}_{i,j} \quad = a^{ee}_{i,j} \wedge m_{i,j} \tag{3e}$$

$$\rho^e_i + C_i \leq \rho^e_j + \mathcal{M} \times (1 - am^{ee}_{i,j}) \tag{3f}$$

$$\rho^X_{(i,f)} + DELAY^X_{(i,f)} \leq \rho^Y_{(j,g)} + \mathcal{M} \times (1 - a^{XY}_{(i,f),(j,g)}) \tag{3g}$$

Equation (3f) forbids the overlapping of two *exec* phases when mapped on the same core by forcing one to execute after the other. Equation (3g) forbids to have more than one active memory transfer at a time to produce contention-free schedules. Equations (3f) and (3g) must be activated only if the two elements are scheduled in a specific order. Thus, a nullification method is applied by using the classical big-M notation (the big-M notation allows to force a constraint to hold depending on a condition as further explained in [18]). The selected value for the big-M constant is the makespan of a sequential schedule on 1 core, the sum of tasks' WCETs and communication delays, which is the worst scenario that can arise.

Read-exec-write semantics constraints

Equations (4a) and (4b) constrain the order of all phases of a task to be *read* phase, then *exec* phase, then *write* phase. But, these phases will not necessarily be scheduled contiguously. The start date of τ_i^e (ρ_i^e) must be some time after the completion of all *read* fragments (start of *read* fragment $\rho_{(i,f)}^r$ + latency $DELAY_{(i,f)}^r$). Similarly, each *write* fragment starts ($\rho_{(i,f)}^w$) some time after the end of the *exec* phase (start of *exec* phase ρ_i^e + WCET C_i).

$\forall i \in T$,

$$\forall f \in F_i^r, \rho_i^e \geq \rho_{(i,f)}^r + DELAY_{(i,f)}^r \tag{4a}$$

$$\forall f \in F_i^w, \rho_{(i,f)}^w \geq \rho_i^e + C_i \tag{4b}$$

Data dependencies in the task-graph

Equation (5) enforces data dependencies by constraining all *read* fragments to start after the completion of all their respective predecessors. For a *read* fragment its predecessor is the *write* fragment of the task that produced the corresponding data.

$$\forall i \in T, \forall f \in F_i^r, \forall (j,g) \in pred(i,f) \quad \rho_{(j,g)}^w + DELAY_{(j,g)}^w \leq \rho_{(i,f)}^r \tag{5}$$

Assigning SPM regions

Equations (6a) & (6b) force every *element* (exec phase and fragments) from τ_i to be mapped on one and only one region z. Identically to [22], we initially consider the number of regions to be equal to the number of elements (number of *exec phase* + number of fragments). With the limited capacity of the SPM, the solver will then be able to minimize the number of effectively used regions.

$$\forall i \in T; \quad \sum_{z \in R} spmp_{z,i} = 1 \tag{6a}$$

$$f \in F, i = task(f); \quad \sum_{z \in R} spmp_{z,(i,f)} = 1 \tag{6b}$$

Equations (7a) and (7b) set the size ($spmsr_z^c$) of region z on core c to be the largest amount of data that will be stored in it. The data stored by an *exec* phase includes the code size (CS_t) and local data (SS_t, stack data). The data stored by a *read* or *write* fragment ($D_{(i,f)}^X$) includes all data consumed (or produced) by a task from one predecessor (or one successor). To store data into a given region of a core, both mapping variables for the region $spmp_{z,(i,f)}$ and the core $p_{i,c}$ must be set to 1.

$\forall c \in P, \forall z \in R, \forall i \in T$,

$$spmsr_z^c \geq \ (SS_i + CS_i)\ (spmp_{z,i} \wedge p_{i,c}) \tag{7a}$$

$$\forall \chi \in \{r,w\}, \forall f \in F_i^\chi; spmsr_z^c \geq \ D_{(i,f)}^\chi\ (spmp_{z,(i,f)} \wedge p_{i,c}) \tag{7b}$$

Equation (8) limits the sum of size for each region for a core to the available SPM size.

$$\forall c \in P, \qquad \sum_{z \in R} spmsr_z^c \leq SPMSIZE_c \tag{8}$$

Delimiting the usage time of a region by an element relies on Definitions 1, 2 and 3. Equation (9a) sets the allocation start time $\sigma_{(i,f)}$ of $\tau^r_{(i,f)}$ to be equal to its schedule start time and the allocation end time $\omega_{(i,f)}$ to be the end of the corresponding *exec* phase. Equation (9b) forces the lifetime of the region used by the *exec* phase to be the whole duration of the schedule (recall that Θ represents the overall makespan). Equation (9c) sets the allocation start time $\sigma_{(i,f)}$ of $\tau^w_{(i,f)}$ equal to the beginning of the *exec* phase and the allocation end time $\omega_{(i,f)}$ equal to its start time.

$\forall i \in T$

$$\forall f \in F^r_i; \quad \sigma_{(i,f)} = \rho^r_{(i,f)} \quad and \quad \omega_i = \rho^e_i + C_i \tag{9a}$$

$$\sigma_i = 0 \quad and \quad \omega_i = \Theta \tag{9b}$$

$$\forall f \in F^w_i; \quad \sigma_{(i,f)} = \rho^e_i \quad and \quad \omega_{(i,f)} = \rho^w_{(i,f)} + DELAY^w_{(i,f)} \tag{9c}$$

Mapping elements (*exec* phases and communication fragments) to SPM regions is very similar to mapping tasks on cores. Therefore, following equations (10a), (10b), (10c) and (10d) mimic the behaviour of respectively (3b), (3d), (3e) and (3f) by replacing variables $m_{i,j}, a_{i,j}$ and $am_{i,j}$ with $spmm_{i,j}, spma_{i,j}$ and $spmam_{i,j}$. As a reminder, (10a) detects if two fragments are assigned to the same region from the same core, (10b) represents the causality of a fragment compare to another, and (10c) represents this causality on the same region. Finally, (10d) imposes the mutual exclusion of the reservation time.

$\forall (f,g) \in F \times F, f \neq g, i = task(f), j = task(g)$

$$spmm_{(i,f),(j,g)} = \sum_{z \in R} (m_{i,j} \wedge spmp_{z,(i,f)} \wedge spmp_{z,(j,g)}) \tag{10a}$$

$$spma_{(i,f)} + spma_{(j,g)} = 1 \tag{10b}$$

$$spmam_{(i,f),(j,g)} = spma_{(i,f)} \wedge spmm_{(i,j),(j,g)} \tag{10c}$$

$$\omega_{(i,f)} \leq \sigma_{(j,g)} + \mathcal{M} \times (1 - spmam_{(i,f),(j,g)}) \tag{10d}$$

5.2 Quality of the heuristic compared to the ILP

The following experiments aim at estimating the gap between makespans of schedules generated by the heuristic opposed to the optimal solutions provided by the ILP solver. We expect this gap to be small. Due to the intrinsic complexity of solving our scheduling problem using ILP, we need for these experiments a large number of small task-graphs, such that the ILP is solved in reasonable time. We thus used synthetic task graphs generated using TGFF (see Table 2). For each graph, we varied the number of cores in $\{2, 4, 8, 12\}$ and the sizes of the SPM vary in $\{2KB, 4KB\}$. SPM sizes allow to cover three situations: 1) all test-cases fit in the SPM (4KB size), 2) some test-cases do not entirely fit in SPM (2KB size), 3) some test-cases are too large, hence unschedulable (2KB size, biggest benchmarks).

The ILP solver used is CPLEX v12.7.1 configured with a timeout of 24 hours. The heuristic is implemented in C++ with a 60 minutes timeout.

Table 4 Degradation of the heuristic compared to the ILP on the synthetic task-graphs.

% of exact results (ILP only)	degradation <min,max,avg> %
68%	0%, 20%, 3%

Table 4 presents the combined results for all different configurations. First, it shows the number of optimal (including infeasible) results the ILP solver is able to find in the given timeout – 68%. The remaining 32% includes all other cases where the solver reaches the timeout without neither an optimal solution nor an infeasibility verdict. Then Table 4 presents the minimum/maximum and the average degradation induced by the heuristic over the ILP. As displayed, the average degradation is low thus showing the quality of our heuristic.

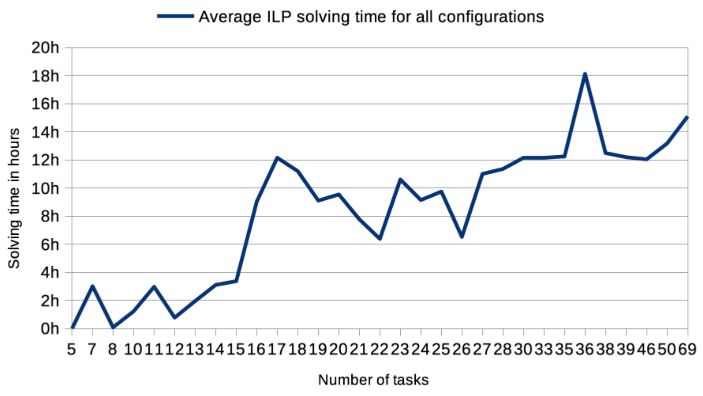


Figure 3 Average ILP solving time for all configurations per number of tasks.

Figure 3 shows that solving an ILP problem does not scale with the growing number of tasks. In contrast, we believe that the proposed scheduling technique does, given its low running times: for the synthetic graphs the average schedule generation times are always less than one second, while for the SRT2RTS benchmarks (up to 340 tasks), the heuristic needs 4 minutes on average.

5.3 Blocking vs non-blocking communications

To compare the benefit of hiding communication latency, the proposed scheduling technique must be opposed to a scheduler that does not hide it. We preferred to modify our heuristic to implement both the *blocking* and *non-blocking* methods instead of reusing a state-of-the-art algorithm. The main reason, as detailed in Section 6, is that related work have characteristics that are hardly compatible with our proposal: different task model [35], SPM big enough to store all code/data [30, 3], lack of information on SPM management [4], different interconnect [16]. Another reason for this choice is to guarantee that the deviation between the results from the two communication modes will not be affected by any other technical implementation decision (e.g.: sorting algorithm).

To summarize the modifications applied to the heuristic in order to get the blocking mode: 1) we forbid to have more than one phase active at a time (both communication and computation as in the example of Figure 2a) 2) we do not fragment communications. We varied the number of cores in $\{2, 4, 8, 12\}$, and the SPM sizes in $\{4KB, 2MB\}$ ($2MB$ is the SMEM (Shared MEMory) size in one cluster of the Kalray MPPA [11]). All aforementioned three situations regarding the SPM size are covered with these configurations. Note that STR2RTS benchmarks are larger in term of memory space than synthetic benchmarks. We then calculate the gain of the *non-blocking* mode versus the *blocking* mode that we expect to be positive.

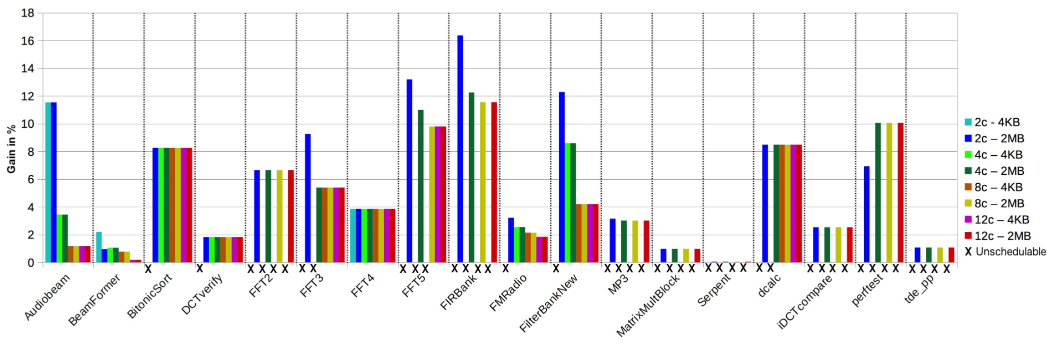


Figure 4 Gain of *non-blocking* communications over *blocking* on STR2RTS benchmarks per cores/SPM configuration – avg: 4%.

Figure 4 presents the average gain per benchmark for all configurations, e.g. *2c-2MB* stands for *2-cores and SPM size of 2MB*. Unfeasible configurations are denoted by the symbol “x”. The maximum gain is 16% (FIRBank on 2 cores with 2MB SPM), whereas the average is 4%.

Figure 4 shows that some benchmarks are *unschedulable* for some configurations, e.g. FFT2 with 2c-4KB. This comes from a lack of SPM space to place all code and all data. This might be relaxed with code pre-fetching in *read* phase, which is left for future work.

Lower gains are observed when the amount of parallelism is low due to the lack of opportunity to hide communications. For example, Serpent is a chain of fork-joins containing 2 concurrent tasks only, as opposed to FIRBank which includes only one fork-join construct with several long chains of tasks. In addition, higher gains are observed on hardware configurations with lower number of cores – i.e. 6% on average with 2-cores as opposed to 4% with 12-cores.

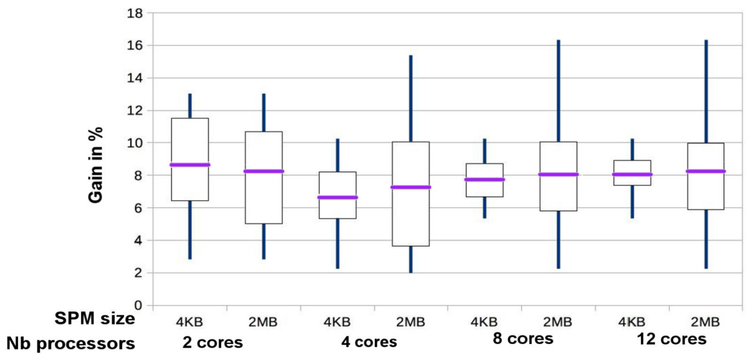


Figure 5 Gain of *non-blocking* communications over *blocking* on TGFF benchmarks.

To evaluate the impact of graph shapes on gains, we experimented our heuristic technique on synthetic task graphs, the ones used previously to validate the heuristic. In contrast to STR2RTS graphs, that are fork-join graphs, synthetic task graphs are arbitrary directed acyclic graphs. Results are depicted in Figure 5. We observe these graphs offer more opportunities to hide communication, with an average gain of 8% in total.

5.4 Impact of fragmentation strategy

Through the paper, we have split *read/write* phases according to tasks dependencies (one fragment per edge in the task). We experimented with two more fine-grain splitting strategies:

- splitting by D_{slot}: each fragment will fit in a T_{slot} bus period, each transmitting D_{slot} bytes – a task transmitting 5 floats (20 bytes) with a $T_{slot} * D_{slot}$ of $3 * 4$ bytes per request will result to 2 fragments, generating 2 communications.
- splitting by *data-type unit (DTU)*: an application exchanging only *floats* will have a DTU of 1 float (4 bytes). If a task produces 5 floats, then there is 5 fragments.

We conducted the experiments by applying our heuristic on the STR2RTS benchmarks, with the very same experimental setup as before. We include in the comparison scheduling in *non-blocking* mode without communication fragmentation (label *no frag* in Figure 6). We expect the gain to increase as the fragment granularity gets smaller.

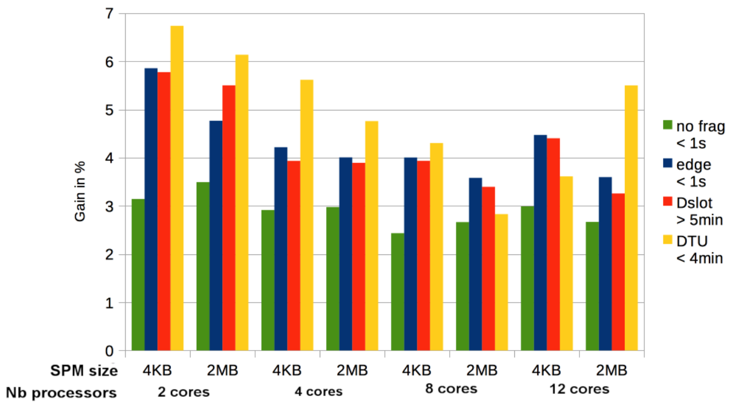


Figure 6 Average gain of *non-blocking* over *blocking* depending on fragmentation strategy.

Figure 6 presents the results with four granularities: *no frag*, *edge* (default configuration), D_{slot} (12 bytes) and *DTU* (4 bytes). Fragmenting communications always result in shorter schedules than the *no frag* configuration. In addition, in most cases the smaller granularity results in higher gains. However the better the results are, the higher the schedule generation time is, as given in the legend of the figure. Schedules are generated in less than 1 second on average for *no frag* and *edge*, whereas several minutes are required on average for fine-grain fragments.

5.5 Schedule implementation on a Kalray MPPA platform

We successfully implemented schedules generated with our heuristic targeting one cluster of a Kalray MPPA Bostan platform [11]. The final code is largely auto-generated (only the code of the exec phase of each task has to be inserted manually in the generated code). At the time of writing, we managed to run benchmark *BeamFormer_12ch_4b* from the STR2RTS benchmark suite [31]. Benchmark *BeamFormer_12ch_4b* is made of 56 tasks with a DAG width of 12. The Kalray MPPA platform includes 16 clusters, each containing 16 cores and a SMEM of 2MB. Four I/O clusters, containing 4 cores each, access either the off-chip global memory or the Ethernet. Clusters are connected through a Network On Chip (NoC).

Following is a summary of the implementation on the Kalray MPPA. Implementation was done at the *bare metal* level. The SMEM is configured in *banked* address mapping mode (consecutive addresses are mapped to the same memory bank), with memory banks are split between computing cores at compile time to have a single-bank considered as SPM per core, as assumed in Section 2.

Data exchanges between cores and the off-chip memory walk through the architecture's NoC, which in our experiments is free from interferences as we only use one cluster of the architecture[2]. Communications are implemented using the Kalray *channel connectors* (one channel for reading, one channel for writing), kept open for the whole execution of the application.

Each core runs one thread, in charge of implementing the schedule of *exec* phases generated off-line, by interleaving a *sched* function between *exec* phases. The *sched* uses *busy-waiting* (reads the local clock of the core to wait for tasks' start time). The worst-case measured overhead of the *sched* function due to clock reading is 32 cycles. An *ad hoc* protocol using barriers is used to re-synchronize local clocks at application start. A specific core is reserved to act as a software DMA engine and is in charge of implementing the schedule of communications (*read* and *write* phases) determined off-line, in a contention-free manner. Implementation of communication phases schedule is identical to the one of computation phases. Moreover, the I/O receiving core follows the schedule to receive and store data to the main memory or to send it to the cluster.

We were able to generate the following versions of the benchmark: 1) *blocking* mode (S_{bl}), 2) *non-blocking* mode without communication fragmentation (S_{nbl}), 3) *non-blocking* mode with fragmentation by edges (S_{nbl}^{edge}), 4) *non-blocking* mode with fragmentation by D_{slot} (12 bytes) (S_{nbl}^{dslot}), 5)*non-blocking* mode with fragmentation by DTU, fragment size is one 4-bytes word (1 float) (S_{nbl}^{dtu}).

In terms of implementation overheads, there is no overhead to set up the software-implemented DMA at run-time, since *channel connectors* are initialized only once at application start. The overhead of 32 cycles due to the scheduler implementation is taken into account.

For this experiments, WCETs of computations and communications were estimated using measurements, adding an arbitrarily chosen margin of 20% for safety. Taking into account implementation overheads, as expected, the overall schedule makespans are: $S_{bl} > S_{nbl} > S_{nbl}^{edge} > S_{nbl}^{dslot}$. The gain of S_{nbl} schedule over the S_{bl} schedule is 1%, the gain of S_{nbl}^{edge} schedule over the S_{nbl} schedule is 36%, and the gain of S_{nbl}^{dslot} over S_{nbl} is 22%.

However, the finer fragmentation policy suffers from an overhead on this platform. The degradation of S_{nbl}^{dtu} over S_{nbl}^{edge} is 24%. The source of this overhead mainly originates from *read* phases measured time where reading one float takes as much time as reading four floats. Nevertheless, we observe a small decrease in *write* phases measured time depending on the amount of data exchanged (approximately 1000 cycles on average).

[2] Note that our abstract architecture model from Section 2 uses a bus. Using a NoC in the Kalray MPPA only changes the overall communication delay computed in Equation 1 since the NoC is free from contentions.

6 Related work

Accessing the global shared memory has always been a performance bottleneck. To overcome this issue, prefetching mechanisms bring information closer to the processor before it is actually needed. Hardware prefetchers will speculatively request data or instructions based on memory access patterns [28]. Software prefetchers give control to the developer or compiler to introspect the code and add prefetching instructions [23]. In this paper we propose a *software* prefetcher that adds prefetching based on a schedule generated off-line.

Most of other works considering SPM aim at deciding what should be stored into the SPM and when to evict data, and in cases some information cannot be stored in SPM it stays in main memory. Considered metrics for SPM allocation are *average-case* performance [15, 25], *power consumption* [36], *WCET* [13], or schedule makespan [4]. In contrast to these studies, our work, in order to control resource contention, requires all information to be stored in SPM.

Wasly and Pellizzoni [38] add a hardware component, named RSMU, to manage the SPM. This RSMU acts similarly as a Memory Management Unit (MMU), except it also uses a previously computed schedule for loads/unloads of code/data from mixed-critical tasks. To use our method, no specific hardware component needs to be added. Giorgi et al. [17] introspect the code to add control of the RSMU, in order to prefetch global data from the global external memory into a local memory on a many-core architecture. They modified the compiler to isolate loads into specific basic blocks and added synchronization if the mandatory data are not yet ready for use. However, their study does not include any real-time guarantee on blocking times. We can guarantee the data will be ready for use without blocking time.

Kim et al. [22] present an algorithm to map a function to a specific SPM region, that inspired our phase to region mapping step. They aim at storing the basic blocks into the SPM in order to improve the WCET of an application on a single-core. We improve their work to map multiple tasks on multi-cores.

Cheng et al. [7] derive a speed-up factor and a resource augmentation factor when partitioning memory banks with minimum interference. At the opposite we have a complete off-line schedule with phase to region allocation on single bank SPM memory.

The PRedictable Execution Model (PREM) from Pellizzoni et al. [29, 1] exposes parallelism by splitting tasks in communication/computation phases. PREM has been widely used – e.g. [38, 3, 39, 4] – because it increases the predictability of an application by isolating memory accesses. Coupling this principle with a software-managed memory (SPM) drastically improves the predictability of the application and so improves its estimated WCET. The authors of [27] present a method to automatically adapt any application to the PREM model, which then allows the application of any SPM load/unload technique including ours. The studies we could find exploiting both the SPM and the PREM model usually fuse the *write* phase of a task with the next activated *read* phase on the same core [38, 39, 2]. As opposed to them, we follow the Acquisition-Execution-Restitution principle from [26] which adds more freedom to schedule generation.

On a single-core, using PREM, Soliman et al. [34] hide the communication latency at the basic-block level thanks to a modification of the LLVM compiler toolchain. Wastly and Pellizzoni [39] proposed to dynamically co-schedule, without preemption, DMA accesses and sporadic tasks on a SPM-based single-core. The SPM is split in 2 parts: one assigned to the currently executing task, while the other load information for the next scheduled task. Our work makes a better use of the SPM by allowing more than two regions alive at the same time. This last work has been extended to multi-core in [2].

Rouxel et al. [30] presented a co-scheduling and mapping of computation and communication phases from task-graph for multi-cores. They limited their work to *blocking* communication whereas we use *non-blocking* ones and we fragment them to add flexibility in the schedule. They assume an infinite SPM size, which looks to us unrealistic, therefore we relaxed this assumption in our scheduling method. In addition, they showed that their scheduling method with an accurate contention model exhibits similar gain and a larger solving times than contention-free ones. Hence, we use a contention-free model in this paper.

The technique proposed in [4] generates contention-free off-line schedules with periodic dependant tasks. Dealing with the SPM, they aim at deciding if a task should be resident in SPM or be fetched before each execution from the global memory. Unfortunately they do not provide information on SPM allocation, raising questions about address allocation and SPM fragmentation. With our region allocation scheme, an SPM allocation scheme that manages fragmentation is proposed.

A technique to hide transfers behind calculations is presented in [35]. Similarly to [39] and [2], the SPM is split in two regions, one used by the application while the other is being loaded. Our work differs from the work in [35] by the task model under use (dependant tasks in our work, sporadic independent tasks in their work). Moreover, our work make better use of SPM by allowing more than two SPM regions to be alive simultaneously.

The work presented in [16] proposes an off-line scheduling scheme for flight management systems using a PREM-like task model. The proposed schedule avoid interferences to access the communication medium. However, in contrast to our work, there are still interferences in their schedule, due to communications between tasks assigned to different cores.

Other works very close to our research, such as [24, 9, 37, 33], statically schedule applications represented by synchronous data flow graphs with some form of buffer checking. However, they do not use the PREM/AER model like us [37, 33], and none of them fragment the communications, which allows us to drastically increase the hiding opportunities. The research presented in [6] proposes a feasibility test that verifies whether scratchpad memories are large enough to contain the maximum memory backlog that may be generated by an application modeled as a task graph. In contrast to [6], our work focuses not only on memory usage feasibility but also on timing feasibility.

7 Conclusion

In this work, we have shown how to minimize the impact of the communication latency when mapping/scheduling a task graph on a multi-core, by overlapping communications and computations. We also argued this kind of technique should always be coupled with a memory allocation scheme to guarantee the integrity of the accessed data. Thus we formulated such allocation scheme in our scheduler. Our experimental results show that, compared to a scenario not overlapping communications and computations, our approach improves the schedule makespan by 4% on average on streaming application (8% on synthetic task graphs). As future work, we plan to improve the accesses of the global main memory such as the DRAM where the scheduler accounts for the locality in this memory. For example, the fragments could be designed to exploit DRAM row locality and read/write switching of the communications. In the near future, we intend to extend this work to applications integrating multiple DAGs. Finally, we plan to strengthen our implementation on the Kalray MPPA platform, especially on the SMEM management.

References

1 Ahmed Alhammad and Rodolfo Pellizzoni. Time-predictable execution of multithreaded applications on multicore systems. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6. IEEE, 2014.

2 Ahmed Alhammad, Saud Wasly, and Rodolfo Pellizzoni. Memory efficient global scheduling of real-time tasks. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pages 285–296. IEEE, 2015.

3 Matthias Becker, Dakshina Dasari, Borislav Nicolic, Benny Akesson, Vincent Nélis, and Thomas Nolte. Contention-free execution of automotive applications on a clustered many-core platform. In *Real-Time Systems (ECRTS), 2016 28th Euromicro Conference on*, pages 14–24. IEEE, 2016.

4 Matthias Becker, Saad Mubeen, Dakshina Dasari, Moris Behnam, and Thomas Nolte. Scheduling multi-rate real-time applications on clustered many-core architectures with memory constraints. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 560–567, January 2018. doi:10.1109/ASPDAC.2018.8297382.

5 Gerald G Brown and Robert F Dell. Formulating integer linear programs: A rogues' gallery. *INFORMS Transactions on Education*, 7(2):153–159, 2007.

6 Daniel Casini, Alessandro Biondi, Geoffrey Nelissen, and Giorgio C. Buttazzo. Memory Feasibility Analysis of Parallel Tasks Running on Scratchpad-Based Architectures. In *2018 IEEE Real-Time Systems Symposium, RTSS 2018, Nashville, TN, USA, December 11-14, 2018*, pages 312–324, 2018.

7 Sheng-Wei Cheng, Jian-Jia Chen, Jan Reineke, and Tei-Wei Kuo. Memory Bank Partitioning for Fixed-Priority Tasks in a Multi-core System. In *Real-Time Systems Symposium (RTSS), 2017 IEEE*, pages 209–219. IEEE, 2017.

8 Junchul Choi, Hyunok Oh, Sungchan Kim, and Soonhoi Ha. Executing synchronous dataflow graphs on a spm-based multicore architecture. In *Proceedings of the 49th Annual Design Automation Conference*, pages 664–671. ACM, 2012.

9 Yoonseo Choi, Yuan Lin, Nathan Chong, Scott Mahlke, and Trevor Mudge. Stream compilation for real-time embedded multicore systems. In *Code generation and optimization, 2009. CGO 2009. International symposium on*, pages 210–220. IEEE, 2009.

10 Robert I. Davis and Alan Burns. A survey of hard real-time scheduling algorithms for multiprocessor systems. *in ACM Computing Surveys*, 2011.

11 Benoît Dupont De Dinechin, Duco Van Amstel, Marc Poulhi‘es, and Guillaume Lager. Time-critical computing on a single-chip massively parallel processor. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6. IEEE, 2014.

12 Steven Derrien, Isabelle Puaut, Panayiotis Alefragis, Marcus Bednara, Harald Bucher, Clément David, Yann Debray, Umut Durak, Imen Fassi, Christian Ferdinand, Damien Hardy, Angeliki Kritikakou, Gerard Rauwerda, Simon Reder, Martin Sicks, Timo Stripf, Kim Sunesen, Timon ter Braak, Nikolaos Voros, and Jürgen Becker. WCET-Aware Parallelization of Model-Based Applications for Multi-Cores: the ARGO Approach. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2017*. IEEE, 2017.

13 Jean-Francois Deverge and Isabelle Puaut. WCET-directed dynamic scratchpad memory allocation of data. In *Real-Time Systems, 2007. ECRTS'07. 19th Euromicro Conference on*, pages 179–190. IEEE, 2007.

14 Robert P Dick, David L Rhodes, and Wayne Wolf. TGFF: task graphs for free. In *Proceedings of the 6th international workshop on Hardware/software codesign*, pages 97–101. IEEE Computer Society, 1998.

15 Boubacar Diouf, Can Hantacs, Albert Cohen, "Ozcan "Ozturk, and Jens Palsberg. A decoupled local memory allocator. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):34, 2013.

16 Guy Durrieu, Madeleine Faugere, Sylvain Girbal, Daniel Gracia P'erez, Claire Pagetti, and Wolfgang Puffitsch. Predictable flight management system implementation on a multicore processor. In *Embedded Real Time Software (ERTS'14)*, 2014.

17 Roberto Giorgi, Zdravko Popovic, and Nikola Puzovic. Exploiting DMA to enable non-blocking execution in Decoupled Threaded Architecture. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

18 Igor Griva, Stephen G. Nash, and Ariela Sofer. *Linear and Nonlinear Optimization, Second Edition.* Society for Industrial Mathematics, 2008.

19 James A Kahle, Michael N Day, H Peter Hofstee, Charles R Johns, Theodore R Maeurer, and David Shippy. Introduction to the cell multiprocessor. *IBM journal of Research and Development*, 49(4.5):589–604, 2005.

20 Md Kamruzzaman, Steven Swanson, and Dean M. Tullsen. Inter-core Prefetching for Multicore Processors Using Migrating Helper Threads. *SIGPLAN Not.*, 46(3):393–404, March 2011. doi:10.1145/1961296.1950411.

21 Timon Kelter, Tim Harde, Peter Marwedel, and Heiko Falk. Evaluation of resource arbitration methods for multi-core real-time systems. In *WCET*, pages 1–10, 2013.

22 Yooseong Kim, David Broman, Jian Cai, and Aviral Shrivastaval. WCET-aware dynamic code management on scratchpads for software-managed multicores. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 179–188. IEEE, 2014.

23 Alexander C Klaiber and Henry M Levy. An architecture for software-controlled data prefetching. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 43–53. ACM, 1991.

24 Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *ACM SIGPLAN Notices*, volume 43, pages 114–124. ACM, 2008.

25 Lian Li, Jingling Xue, and Jens Knoop. Scratchpad memory allocation for data aggregates via interval coloring in superperfect graphs. *ACM Transactions on Embedded Computing Systems (TECS)*, 10(2):28, 2010.

26 Cl'audio Maia, Luis Nogueira, Luis Miguel Pinho, and Daniel Gracia P'erez. A closer look into the aer model. In *Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on*, pages 1–8. IEEE, 2016.

27 Renato Mancuso, Roman Dudko, and Marco Caccamo. Light-PREM: Automated software refactoring for predictable execution on COTS embedded systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*, pages 1–10. IEEE, 2014.

28 Pierre Michaud. Best-Offset Hardware Prefetching. In *International Symposium on High-Performance Computer Architecture*, Barcelona, Spain, March 2016. doi:10.1109/HPCA.2016.7446087.

29 Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for COTS-based embedded systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279. IEEE, 2011.

30 Benjamin Rouxel, Steven Derrien, and Isabelle Puaut. Tightening contention delays while scheduling parallel applications on multi-core architecture. In *Embedded Software (EMSOFT), 2017 International Conference on.* ACM, 2017.

31 Benjamin Rouxel and Isabelle Puaut. STR2RTS: Refactored StreamIT Benchmarks into Statically Analyzable Parallel Benchmarks for WCET Estimation & Real-Time Scheduling. In Jan Reineke, editor, *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 57 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–12, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2017.1.

32 Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch. Patmos reference handbook. *Technical University of Denmark, Tech. Rep*, 2015.

33 Stefanos Skalistis and Alena Simalatsar. Near-optimal deployment of dataflow applications on many-core platforms with real-time guarantees. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 752–757. IEEE, 2017.

34 Muhammad Refaat Soliman and Rodolfo Pellizzoni. WCET-Driven Dynamic Data Scratchpad Management With Compiler-Directed Prefetching. In Marko Bertogna, editor, *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:23, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.ECRTS.2017.24`.

35 Rohan Tabish, Renato Mancuso, Saud Wasly, Ahmed Alhammad, Sujit S Phatak, Rodolfo Pellizzoni, and Marco Caccamo. A real-time scratchpad-centric os for multi-core embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016 IEEE*, pages 1–11. IEEE, 2016.

36 Hideki Takase, Hiroyuki Tomiyama, and Hiroaki Takada. Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pages 1124–1129. IEEE, 2010.

37 Pranav Tendulkar, Peter Poplavko, Ioannis Galanommatis, and Oded Maler. Many-core scheduling of data parallel applications using SMT solvers. In *Digital System Design (DSD), 2014 17th Euromicro Conference on*, pages 615–622. IEEE, 2014.

38 Saud Wasly and Rodolfo Pellizzoni. A dynamic scratchpad memory unit for predictable real-time embedded systems. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 183–192. IEEE, 2013.

39 Saud Wasly and Rodolfo Pellizzoni. Hiding memory latency using fixed priority scheduling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 75–86. IEEE, 2014.

A STR2RTS benchmark suite

Following Table 5 characterise the used benchmarks from STR2RTS benchmark suite. The first column presents the number of tasks and the second column the width of the graph. Then it gives the average data in bytes sent along all edges. Following is the average memory footprint of all tasks withing a benchmark, it includes the code size and the stack size. Last column shows the average, among all tasks, of WCET estimates. All this information are shipped with the benchmark suite and target a Patmos single core architecture [32].

Table 5 Benchmarks characteristics.

Name	#Tasks	Width	avg data (bytes)	avg task's memory footprint	avg task's WCET
Audiobeam	20	15	12 B	108 B	41
Beamformer	56	12	18 B	246 B	2718
BitonicSort	122	8	49 B	109 B	30
DCTverify	7	2	513 B	506 B	10045
FFT2	26	2	551 B	2 KB	618
FFT3	82	16	84 B	208 B	120
FFT4	10	2	6 B	32 B	11
FFT5	115	16	52 B	1 KB	38
Firbank	340	12	505 B	2 KB	670
FMRadio	67	20	6 B	191 B	235
FilterbankNew	53	8	35 B	180 B	144
MP3	116	36	3502 B	19 KB	12222
MatrixMultiBlock	23	2	793 B	1 KB	726
Serpent	234	2	1013 B	709 B	922
dcalc	84	4	106 B	685 B	174
IDCTcompare	13	3	454 B	685 B	4557
perftest	16	4	8267 B	21 KB	5269
tde_pp	55	2	25344 B	16 KB	2931

Slot-Based Transmission Protocol for Real-Time NoCs – SBT-NoC

Borislav Nikolić
Institute of Computer and Network Engineering, TU Braunschweig, Germany
nikolic@ida.ing.tu-bs.de

Robin Hofmann
Institute of Computer and Network Engineering, TU Braunschweig, Germany
hofmann@ida.ing.tu-bs.de

Rolf Ernst
Institute of Computer and Network Engineering, TU Braunschweig, Germany
ernst@ida.ing.tu-bs.de

Abstract

Network on Chip (NoC) interconnects are some of the most challenging-to-analyse components of multiprocessor platforms. This is primarily due to the following two reasons: (i) NoCs contain numerous shared resources (e.g. routers, links), and (ii) the network traffic often concurrently traverses multiple of those resources. Consequently, complex contention scenarios among traffic flows might occur, some of the important implications being significant performance limitations, and difficulties when performing the real-time analysis.

In this work, we propose a slot-based transmission protocol for NoCs (called SBT-NoC), and an accompanying analysis method for deriving worst-case traffic latencies. The cornerstone of SBT-NoC is a contention-less slot-based transmission, arbitrated via a protocol running on a dedicated network medium. The main advantage of SBT-NoC is that, while not requiring any sophisticated hardware support (e.g. virtual channels, a flit-level arbitration), it makes NoCs amenable to real-time analysis and guarantees bounded low latencies of high-priority time-critical flows, which is a *sine qua non* for the inclusion of NoCs, and multiprocessors in general, in the real-time domain. The experimental evaluation, including both synthetic workloads and a use-case of an autonomous driving vehicle application, reveals that SBT-NoC offers a plethora of configuration opportunities, which makes it applicable to a wide range of diverse traffic workloads.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Embedded systems

Keywords and phrases Real-Time Systems, Embedded Systems, Network-on-Chip, Protocols

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.26

1 Introduction

Even though multiprocessors are now ubiquitous in almost all computing areas, they are still often considered as a new frontier technology in the real-time domain. Traditionally, in the real-time analysis of multiprocessors, the emphasis is on a single type of shared resources – processing elements (cores). However, due to the core proliferation trend in the multiprocessor area, contentions for other shared resources, such as an interconnect medium, become more apparent. This implies that, in order to perform the timing analysis of real-time applications deployed on multiprocessors, it is no longer sufficient to only take their computation requirements into account, but communication and memory traffic need to be considered as well. Therefore, the real-time analysis of network interconnects became a crucial prerequisite for the integration of multiprocessors in the real-time domain.

The Network-on-Chip architecture [2] (NoC) is among the prevalent choices for interconnects in contemporary multiprocessors, mostly due to its good performance and scalability potential [14]. However, the real-time analysis of NoCs is a challenging topic. This is primarily due the following two reasons: (i) NoCs are composed of numerous shared resources (e.g. routers, links), and (ii) network traffic flows often concurrently traverse multiple of those resources. As a consequence, infamous contention scenarios among traffic flows might occur (e.g. *head of line blocking* [5] and *backpressure* [27]), causing significant performance degradations as well as making the worst-case timing analysis difficult to perform.

Interestingly, the worst-case timing analysis of NoCs with single-channel ports and a fixed-priority packet-level arbitration (referred to as *basic NoCs* hereafter) is still an open research topic. This can be largely attributed to the aspects from the previous paragraph, implying that basic NoCs "out-of-the-box" (i.e. without any enhancements) are not amenable to real-time analysis, and hence do not represent satisfactory solutions for the real-time domain. This is discussed in more detail in Section 6 (Experiment 1).

How to make NoCs viable interconnect choices for the real-time domain? Intuitively, mechanisms are needed to ensure that a low priority traffic interferes with a high priority one only slightly (ideally not at all), thus effectively providing low latencies for high-priority time-critical flows, and big latencies for low-priority ones – which may not even have any timing requirements.

One promising solution to these problems is to use sophisticated hardware support and advanced router functionalities (e.g. virtual channels [5, 6] and a flit-level arbitration [23]), and in that way make NoCs both more performant and more amenable to the real-time analysis (e.g. [11, 17, 26]). Some other notable solutions revolve around (i) organising NoC accesses in a time-division-multiplexing manner (e.g. [18]), (ii) explicitly reserving entire paths before transmissions, called the *virtual circuit* method (e.g. [4]), and (iii) utilising single-cycle multihop transfers to bypass intermediate routers (e.g. [8]). All these approaches suffer from one or more of the following limitations: complex and/or pessimistic timing analysis, expensive hardware requirements, inefficient use of resources, limited applicability to certain workload types, and finally, inability to meet real-time requirements (e.g. bounded low latencies of time-critical flows). This is discussed in more detail in Section 2.

Contribution. In this work, we propose a novel approach for making NoCs more applicable to the real-time domain. Specifically, we propose a slot-based transmission protocol, called SBT-NoC, and an accompanying analysis method for deriving worst-case traffic latencies. SBT-NoC ensures a contention-less slot-based transmission, arbitrated via a protocol running over a dedicated communication medium. By explicitly separating arbitration and data transmission domains, infamous contention scenarios are prevented by design, which contributes to a less complex and less pessimistic worst-case analysis, and perhaps even more importantly, to a better utilisation of network resources. At the same time, a slot-based transmission represents an efficient means to protect high priority flows from the lower-priority interference, thus guaranteeing their bounded low latencies, which is an essential real-time requirement. Finally, it is worth mentioning that besides a dedicated arbitration medium, SBT-NoC does not require any sophisticated router functionalities, nor any additional hardware support (e.g. virtual channels, flit-level arbitration), which implies that SBT-NoC can be easily adopted by next-generation commercial multiprocessors.

2 Related Work

All approaches for the integration of NoCs in the real-time domain can be broadly classified into two categories: contention-less and contention-aware. The former category supports uninterrupted transmissions by implementing a temporal and/or spatial allocation of NoC resources to individual traffic flows. One popular strategy is to allocate resources in a time-division multiplexing (TDM) manner (e.g. [9,10,13,18,20]). The three limitations of TDM-based approaches are: (i) it might be challenging to find an efficient slots assignment configuration, (ii) significant buffer space might be required to store flits in traversed routers, and (iii) providing bounded low latencies for time-critical flows might require either resource over-provisioning (large slots for those flows), or workload type restrictions (e.g. strictly periodic traffic sources). The other approach from this category is a virtual circuit method. It achieves contention-less transmissions by reserving an entire path before a transmission (e.g. [4]). One downside is that a path establishing stage might be time consuming, and hence assuring bounded low latencies for time-critical flows might be non-trivial.

On the other hand, contention-aware approaches allow contentions among traffic flows, which are resolved via in-router arbitration mechanisms. One such method called *round-robin* is particularly popular among hardware manufacturers (e.g. [1,25]). Its popularity comes from the fact that it offers a fair treatment to all traffic flows, thus promoting good performance. Several worst-case analysis methods for NoCs with the round-robin mechanism have been proposed, including the *Network Calculus* framework (e.g. [19]), the *Compositional Performance Analysis* framework (e.g. [24]) and the *Recursive Calculus* framework (e.g. [15]). However, one limitation of this mechanism is that it does not have any means to establish a notion of priorities, and it may be difficult to achieve low latencies of time-critical flows.

Another class of contention-aware approaches uses a priority-based flit-level arbitration via dedicated virtual channels. Currently, there exist several worst-case analysis methods, e.g. [11,17,26]. Although this type of NoCs fulfils almost all requirements of the real-time domain, its biggest limitations are substantial hardware requirements in a form of dedicated virtual channels to each traffic flow within each router along its path. Another relevant point is that dedicated virtual channels and the accompanying logic elevate power requirements, thus rendering such NoCs inapplicable in scenarios where low power consumption is required (e.g. embedded domain). Moreover, these approaches require routing mechanisms that guarantee a single continuous contention domain between any two interfering flows (e.g. dimension-ordered routing), which may be a limiting factor in some scenarios.

Recently, a novel type of interconnect architecture called SMART NoC [8] was introduced. This approach aims to avoid complex in-network interference patterns by utilising a router bypass mechanism which allows single-cycle multihop transmissions. However, one limitation of this approach is that it does not have a means to enforce prioritisation among traffic flows, and hence it may be challenging to achieve low latencies of time-critical flows.

Finally, it is worth mentioning that the arbitration protocol of SBT-NoC is, to an extent, inspired by CAN [7], Byteflight [3] and FlexRay [16] technologies, which are used for bus-based communication, predominantly in automotive in-vehicle networks. The common aspect of these approaches is that, during the arbitration phase, all traffic sources indicate their transmission requests, and at the end of the arbitration cycle one of them is granted the permission to transmit. The arbitration in SBT-NoC is performed in a similar way, with the following two fundamental distinctions: (i) instead of being interleaved with transmission phases, the arbitration is performed continuously over a dedicated arbitration medium, and (ii) instead of concluding the arbitration phase with a single transmission permission, multiple traffic sources are able to gain the transmission permission and subsequently concurrently transmit. The second aspect is of particular importance, because it allows to exploit the full potential of an underlying NoC architecture. More details on the SBT-NoC arbitration protocol (and the approach in general) are available in Section 5.

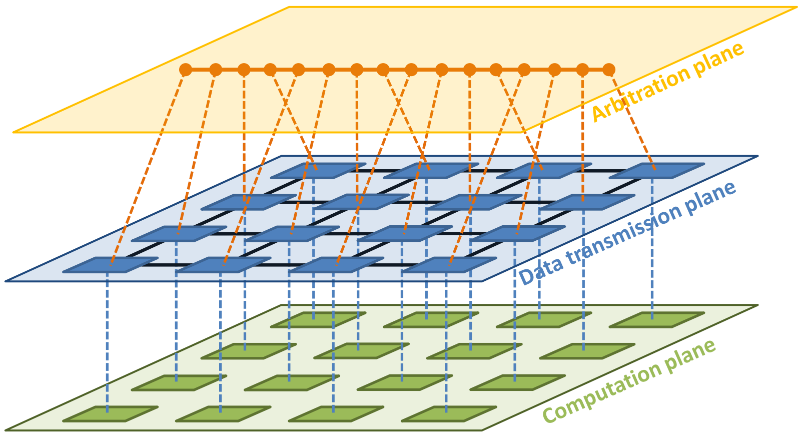


Figure 1 Illustrative example of assumed platform architecture.

3 System Model

3.1 Platform Architecture

The assumed platform architecture is a multiprocessor system $\mathcal{M}$, comprising: (i) a computation plane, (ii) a data transmission plane and (iii) an arbitration plane. One example of the assumed platform is illustrated in Figure 1.

A *computation plane* consists of $m \cdot n$ potentially heterogeneous processing elements (cores) $\{\mu_1, \mu_2, ..., \mu_{m \cdot n}\}$.

A *data transfer plane* consists of $m \cdot n$ interconnected mutually synchronised routers $\{\rho_1, \rho_2, ..., \rho_{m \cdot n}\}$, with identical physical characteristics. Routers are connected in a way to form a 2-D mesh topology, and each router ρ_i is, depending on its position, directly connected with 2, 3 or 4 neighbouring ones. Each two neighbouring routers ρ_i and ρ_j are connected via two unidirectional links of opposite directions, and it is assumed that all network links have the same capacity, where d_L stands for the transmission latency of one flit across one link. The connection between the computation and the data transfer plane is also organised in the form of two unidirectional links between each core μ_i and its corresponding router ρ_i. These links have the same physical characteristics as router-to-router links. As in the vast majority of NoCs, a data transfer is organised with the wormhole switching technique; prior to transmission, a packet is divided into small elements of fixed size called flits, which are successively injected into the NoC, where they travel to their destination in a pipelined manner. Moreover, routers have single-channel ports (no virtual channels necessary), where per-port buffers have the capacity to store at least two flits, so as to ensure a pipelined transmission. Buffer overflows are prevented with credit-based flow control. As a routing mechanism, any static technique can be applied (including source routing), with the only requirement that a flow should not put itself in a deadlock. The packet routing overhead is denoted by d_R, and only the first flit of the packet (header) experiences this delay. Finally, all routers have the same constant frequency ψ. Note that the link transfer latency d_L and the routing latency d_R are typically expressed as a number of cycles (e.g. in Intel's SCCC [12] $d_L = 1$ and $d_R = 3$).

An *arbitration plane*[1] consists of a bus system, to which all routers of the data transmission plane are connected. The bus frequency is assumed to be the same as the frequency of the routers. The term d_B denotes the worst-case latency of one router writing one bit on the bus, and all connected routers reading it. Similar to d_R and d_L, we also assume that d_B can be expressed as a number of cycles.

3.2 Workload

In this work, we take a communication-centric approach, and model the workload as a sporadic traffic flow-set Φ, which is a collection of z flows $\Phi = \{\phi_1, ..., \phi_z\}$. Each flow ϕ_i is characterised by: (i) a source core/router μ_i^{src}/ρ_i^{src}, (ii) a destination core/router μ_i^{dst}/ρ_i^{dst}, (iii) a path $\mathcal{L}_i$, expressed as a set of traversed network links (including those connecting μ_i^{src} and μ_i^{dst} to the NoC), (iv) a payload size σ_i, expressed as a number of bytes, (v) a minimum inter-arrival time T_i, (vi) a constrained deadline[2] $D_i \leq T_i$, and (vii) a unique priority P_i.

During each inter-arrival time, a flow may release at most one packet (consisting of a header flit, payload flit(s) and a tail flit). If it can be analytically proven that each packet of ϕ_i can complete its transfer before its deadline, even in the worst-case conditions, then ϕ_i is considered to be *schedulable*. If all flows of Φ are schedulable, then Φ itself is considered to be *schedulable*.

4 Problem Formulation

Given a platform $\mathcal{M}$ and workload Φ, propose a transmission protocol, and an accompanying worst-case timing analysis method, such that the schedulabtility of Φ on $\mathcal{M}$ can be evaluated. Additional requirements are as follows:

- The transmission protocol should exploit the full potential of the underlying platform by accommodating concurrent transmissions of multiple flows, whenever possible.
- The transmission protocol should ensure low worst-case traversal times (WCTT) of high priority time-critical flows, possibly at the expense of increased WCTTs of low priority ones.
- The timing analysis method should provide safe and tight upper-bounds on WCTTs, so as to avoid resource over-provisioning.

5 SBT-NoC

In this section, we introduce a slot-based transmission protocol for real-time NoCs, called *SBT-NoC*. As already mentioned, SBT-NoC provides contention-less packet transmissions. Before explaining how the contentions are prevented, let us first discuss inter-flow relations.

5.1 Inter-flow Relations

NoC routers and links are shared resources, and it often happens that two packets, belonging to different flows, simultaneously request to access the same resource. In such cases, a higher priority packet should be transmitted as soon as possible, while the lower priority one should

[1] An arbitration plane can be implemented in many different ways. In this work, we focus on one possible implementation strategy – a bus system. Investigating other options is a potential future work.
[2] Extending our approach to include arbitrary deadlines is a potential future work.

be deferred until the shared resource is available. When reasoning about the interference that packets of one flow may suffer, it is important to consider all potentially interfering flows, i.e. all flows that share at least one link[3] with the analysed one.

Let $\phi_i \in \Phi$ be the analysed flow. We classify all its interfering flows into two disjoint sets, namely $\mathcal{F}_i^H$ and $\mathcal{F}_i^L$. The former set is formally introduced with Definition 1.

▶ **Definition 1** (Set of directly interfering flows – $\mathcal{F}^H$). *Consider $f_i \in \Phi$. Set $\mathcal{F}_i^H$ is a set of directly interfering flows of f_i,* **iff** *(if and only if) $\mathcal{F}_i^H$ contains all flows from Φ that have higher priorities than f_i, and share at least one link with it.*

Set $\mathcal{F}_i^H$ can be formally described as follows:

$$\forall f_i, f_j \in \Phi \mid P_j > P_i \land \mathcal{L}_j \cap \mathcal{L}_i \neq \emptyset \iff f_j \in \mathcal{F}_i^H$$

Analogously, the latter set of flows $\mathcal{F}_i^L$ is formally introduced with Definition 2.

▶ **Definition 2** (Set of directly interfered flows – $\mathcal{F}^L$). *Consider $f_i \in \Phi$. Set $\mathcal{F}_i^L$ is a set of directly interfered flows of f_i,* **iff** *$\mathcal{F}_i^L$ contains all flows from Φ that have lower priorities than f_i, and share at least one link with it.*

Set $\mathcal{F}_i^L$ can be formally described as follows:

$$\forall f_i, f_j \in \Phi \mid P_j < P_i \land \mathcal{L}_j \cap \mathcal{L}_i \neq \emptyset \iff f_j \in \mathcal{F}_i^L$$

In order to achieve low latencies of packets of ϕ_i, it is essential to ensure that: (i) a transmission of any packet of ϕ_i can be delayed *only* by flows from $\mathcal{F}_i^H$, and (ii) a transmission of any packet of ϕ_i can delay transmissions of all flows from $\mathcal{F}_i^L$. These two aspects are the cornerstone of SBT-NoC.

5.2 Basic SBT-NoC

After defining flow relations, let us introduce a basic SBT-NoC variant. The advanced SBT-NoC variants have additional configuration possibilities, and they are described in Section 5.3.

5.2.1 Arbitration Mechanism (Basic SBT-NoC)

The arbitration process for SBT-NoC is a continuous activity comprising a potentially infinite sequence of *arbitration slots*. During one arbitration slot, all flows indicate their intentions to transmit packets. An arbitration slot concludes with transmission permissions granted to highest priority flows with pending requests, whose transmissions can be concurrently accommodated without causing any mutual in-network contentions. Upon the completion of one arbitration slot, and before the beginning of the next one, optionally, there might exist a short *pause* termed Δ of duration d_Δ. During the pause, all entities participating in the arbitration should make sure that the decisions derived during the previous arbitration slot have been implemented (e.g. flows that are granted a transmission permission should inject their packets), and that everything is ready for the next arbitration slot. An illustrative example of the basic SBT-NoC arbitration is illustrated in Figure 2.

[3] Due to the crossbar switching fabric inside routers, *router sharing* is only a necessary condition for interference between two flows, because two packets from different input ports can be transferred to different output ports simultaneously. Conversely, *link sharing* (and hence *port sharing*) is both a necessary and a sufficient condition for interference.

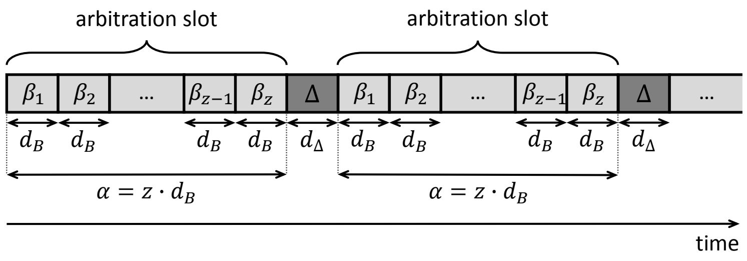

arbitration slot
arbitration slot
β_1 β_2 ... β_{z-1} β_z Δ
d_B d_B d_B d_B d_Δ
$\alpha = z \cdot d_B$
time

Figure 2 Illustrative example of basic SBT-NoC arbitration.

Each arbitration slot has a fixed duration termed α, and it contains a sequence of z dedicated *arbitration intervals* $\{\beta_1, \beta_2, ..., \beta_z\}$, one for each flow (recall that z denotes the number of flows in the flow-sets). All arbitration intervals have an equal duration d_B, and they are assigned to flows in order, with respect to their priorities, non-decreasingly, i.e. β_1 to the highest priority flow, and β_z to the lowest priority one. For the ease of exposition, let us assume that flow indexes are also assigned in the same manner, i.e. ϕ_1 and ϕ_z are the highest and the lowest priority flows, respectively.

During the arbitration interval β_i, a transmission of packets of ϕ_i is assessed. Similar to the CAN protocol, we assume a dominant "0" and a recessive "1" on the arbitration bus. At the beginning of β_i, the bus is in the recessive state. If ϕ_i has packets ready for transmission, its source router ρ_i^{src} will not attempt to change the state of the bus. The recessive "1" indicates a transmission request. In the opposite case (no packets of ϕ_i ready for transmission), ρ_i^{src} will, on behalf of ϕ_i, change the state of the bus to "0", and in that way indicate that it does not have packets ready for transmission.

However, ϕ_i (via ρ_i^{src}) is not the only flow that can manipulate the bus state during β_i. In fact, all flows from $\mathcal{F}_i^H$ can do that. The fact that these flows have higher priorities than ϕ_i implies that their respective arbitration intervals have already concluded, and that these flows are already either granted or denied transmission permissions. If flow $\phi_h \in \mathcal{F}_i^H$ was granted a transmission permission (manifested by a recessive "1" on the bus during β_h), then it will enforce, via ρ_h^{src}, a dominant "0" on the bus during β_i, and in that way deny transmission requests of ϕ_i. Conversely, if ϕ_h does not have packets ready for transmission, it will not manipulate the bus state during β_i. Additionally, if ϕ_i and ϕ_h originate from the same core, and ϕ_h already has a transmission permission, then during β_i the precedence will be given to ϕ_h to set the dominant state "0" on the bus via the common router, regardless of transmission intentions of ϕ_i.

In summary, from the perspective of flow ϕ_i, during arbitration interval β_i, only ϕ_i and flows from $\mathcal{F}_i^H$ are able to manipulate the bus state. Additionally, if the resulting bus state at the end of β_i is a recessive "1" (ϕ_i received a permission to transmit), ϕ_i will also manipulate the bus state during arbitration intervals dedicated to flows from $\mathcal{F}_i^L$ by setting a dominant state "0" (transmission denied).

5.2.2 Transmission Mechanism (Basic SBT-NoC)

All transmissions granted during one arbitration slot should start during a subsequent pause Δ, and should occur concurrently with the next arbitration slot. Granted transmissions must complete before the next pause. Large packets, which cannot complete an entire transfer during one arbitration slot, are transmitted in several stages. During the first transmission stage, the maximum number of flits that could complete the transfer before the next pause are selected, and those flits are transmitted as one *sub-packet*. A transmission of remaining

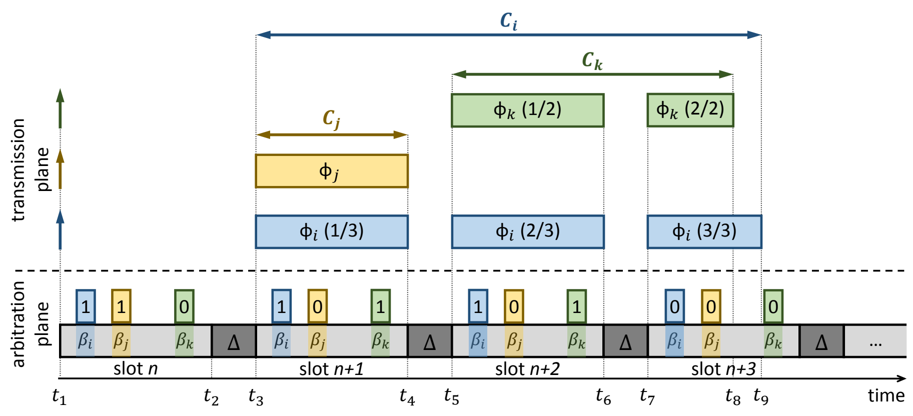


Figure 3 Illustrative example of basic SBT-NoC transmission.

flits is requested during the current arbitration slot. If the permission is granted, during the second transmission stage the maximum number of remaining flits that could complete the transfer before the next pause are selected, and those flit are transmitted as one sub-packet. This process is repeated until eventually the entire packet is transferred.

An illustrative example of a transmission process in SBT-NoC is presented in Figure 3. Three flows ϕ_i, ϕ_j and ϕ_k have packets ready for transmission at time instant t_1 (illustrated with three upward arrows). A packet of ϕ_i is the largest and it requires 3 transmission slots, a packet of ϕ_k requires 2 slots, while a packet of ϕ_j requires only one slot. Flow ϕ_i does not interfere with the remaining two flows, while $\phi_j \in \mathcal{F}_k^H$.

During arbitration slot n, all three flows try to indicate their transmission requests by setting a recessive "1" during their intervals β_i, β_j and β_k. In cases of ϕ_i and ϕ_j, the recessive value remains, i.e. $\beta_i = \beta_j = 1$, while in the case of ϕ_k, due to a potential contention, ϕ_j overrides the value with a dominant "0", i.e. $\beta_k = 0$. Consequently, during slot $n+1$, the first sub-packet of ϕ_i and a packet of ϕ_j are transmitted.

During slot $n+1$, flow ϕ_j does not participate because it does not have packets ready for transmission, and just sets a dominant "0" during β_j. Flows ϕ_i and ϕ_k have sub-packets ready for transmission, so they set recessive "1" during β_i and β_k. The bus remains in the recessive state during those intervals. Consequently, during slot $n+2$, the second sub-packet of ϕ_i and the first one of ϕ_k are being transmitted.

A similar scenario occurs during slot $n+2$, and therefore, during slot $n+3$, the third sub-packet of ϕ_i and the second sub-packet of ϕ_k are being transmitted.

Finally, during slot $n+3$, all three flows yield transmission opportunities to other flows by setting a dominant "0" during their respective intervals, because none of them have packets ready for transmission during slot $n+4$.

In Figure 3, we also illustrated the transmission latencies of the analysed flows, denoted by C_i, C_j and C_k. These latencies are formally introduced in the next section, and they represent the basic components for deriving the worst-case timing analysis method for SBT-NoC.

5.2.3 Packet Splitting and Transmission Latencies (Basic SBT-NoC)

In the previous section, it was mentioned that transmissions of large packets are performed in several stages, each including a transfer of one sub-packet. Before we discuss the process of packet splitting, let us introduce *NoC transmission latency.*

▶ **Definition 3** (NoC transmission latency). *Consider one packet of flow ϕ_i. NoC transmission latency of ϕ_i, termed C_i, is the time interval between the injection of a header flit from μ_i^{src} into the NoC, and the arrival of a tail flit at μ_i^{dst}, where a packet traversed its path without interference.*

NoC transmission latency is often also called the *isolation latency.* It can be computed by solving Equation 1.

$$C_i = \overbrace{(|\mathcal{L}_i| - 1) \cdot d_R}^{\text{header routing}} + \overbrace{|\mathcal{L}_i| \cdot d_L}^{\text{header traversal}} + \overbrace{\left(\left\lceil \frac{\sigma_i}{\sigma_F} \right\rceil + 1 \right) \cdot d_L}^{\text{payload and tail traversal}} \tag{1}$$

Term $|\mathcal{L}_i|$ denotes the number of elements of $\mathcal{L}_i$, also called the *number of hops*, σ_F represents a size of a flit, in bytes, while d_L and d_R and σ_i were introduced in Section 3. NoC transmission latency is equal to the latency of a header flit reaching a destination (the first two terms of Equation 1), augmented by a traversal of payload flits and a tail flit across the last link, due to the pipelined transmission (the last term of Equation 1).

In SBT-NoC, each transmission needs to start during the pause, and complete before the end of the subsequent arbitration slot, which imposes a limit on the amount of payload that can be transferred within a single (sub-)packet. That limit, denoted by $\widehat{\sigma_i}$, can be computed by solving Equation 2.

$$\widehat{\sigma_i} = \left(\frac{\alpha - (|\mathcal{L}_i| - 1) \cdot d_R}{d_L} - |\mathcal{L}_i| - 1 \right) \cdot \sigma_F \tag{2}$$

Equation 2 was derived from Equation 1 by substituting σ_i with $\widehat{\sigma_i}$ and C_i with α. Recall that α denotes a slot duration.

Now we can obtain the minimum number of sub-packets ω_i, which are needed to transfer one packet of ϕ_i (Equation 3).

$$\omega_i = \left\lceil \frac{\sigma_i}{\widehat{\sigma_i}} \right\rceil \tag{3}$$

In SBT-NoC, large packets are transmitted with the minimum number of sub-packets in the following way: the first $\omega_i - 1$ sub-packets have the maximum payload size $\widehat{\sigma_i}$, and the last sub-packet has the payload size $\sigma_i - (\omega_i - 1) \cdot \widehat{\sigma_i}$, also denoted by $\sigma_i^{\dashv}$. Incidentally, these packet splitting and transmission rules also apply to flows ϕ_i, ϕ_j and ϕ_k from Figure 3, where a packet of ϕ_i is transmitted via 3 sub-packets, a packet of ϕ_j via a single sub-packet, and a packet of ϕ_k via 2 sub-packets.

In order to reason about transmission latencies in SBT-NoC, we need to slightly revise Definition 3, so as to account for large packets transmitted in several stages (Definition 4).

▶ **Definition 4** (SBT-NoC transmission latency). *Consider one packet of flow ϕ_i. SBT-NoC transmission latency of ϕ_i is the time interval between the injection of the header flit of the first sub-packet from μ_i^{src} into the NoC, and the arrival of the tail flit of the last sub-packet at μ_i^{dst}, where all transmission requests of ϕ_i during that interval were granted.*

Now we can express transmission latencies of flows in SBT-NoC. If an entire packet of a flow can be transmitted during α, then its transmission latency can be computed by solving Equation 1, i.e., a packet is transmitted in the same way as if it was a regular NoC. This is the case for flow ϕ_j from Figure 3. Conversely, if a packet of a flow is large and its transmission cannot finish during α, then its transmission latency can be computed by solving Equation 4. This is the case for flows ϕ_i and ϕ_k from Figure 3.

$$C_i = \overbrace{(\omega_i - 1) \cdot (\alpha + d_\Delta)}^{\text{transmission of first } \omega_i - 1 \text{ sub-packets}} \overbrace{+ (|\mathcal{L}_i| - 1) \cdot d_R + |\mathcal{L}_i| \cdot d_L + \left(\left\lceil \frac{\sigma_i^{\dashv}}{\sigma_F} \right\rceil + 1 \right) \cdot d_L}^{\text{transmission of the last sub-packet}} \quad (4)$$

5.2.4 Worst-case Analysis (Basic SBT-NoC)

In this section, we provide a method to obtain upper bounds on WCTTs of flows in SBT-NoC. Several factors can contribute to the WCTT of the analysed flow, and in order to derive a safe upper bound, we need to cover all of them.

First, a packet release of ϕ_i may occur after its arbitration interval, in which case its router ρ_i^{src} must wait for the next arbitration slot to indicate the transmission request. In the worst case, a packet may arrive just after its arbitration interval and will have to wait for the remaining part of the slot, augmented by the pause, before it will be able to participate in the arbitration process. This delay is denoted by O_i.

$$O_i = \alpha - i \cdot d_B + d_\Delta \quad (5)$$

Moreover, before a packet can start traversing the NoC, a recessive "1" must be indicated during β_i in one of the subsequent arbitration slots. Once ϕ_i gains a transmission permission, during the next pause its packet is injected into the network. Assuming no higher priority interference, the worst-case delay of acquiring a transmission permission and preparing a packet for transmission is denoted by A_i (Equation 6).

$$A_i = \alpha + d_\Delta \quad (6)$$

Note, regardless of a packet size, only the delay of obtaining the first transmission permission needs to be considered. For large packets, remaining permissions are acquired concurrently with transmissions of preceding sub-packets (e.g. ϕ_i and ϕ_k in Figure 3).

Finally, a transmission of a packet of ϕ_i can be delayed by higher priority flows. This happens when $\phi_h \in \mathcal{F}_i^H$ also has a packet ready for transmission and participates in the same arbitration slot as ϕ_i. Consequently, ϕ_h prevents a transmission of ϕ_i by setting a dominant "0" during the β_i arbitration interval, and ϕ_i has to wait for the next arbitration slot to again attempt to gain a transmission permission.

The delay that higher priority flows inflict on ϕ_i is equal to the cumulative duration of full arbitration slots (augmented with respective pauses), in which a transmission request of ϕ_i was denied with a dominant "0" during β_i. Incidentally, each of these arbitration slots corresponds to one subsequent (sub-)packet transmission of higher priority flows. Therefore, the delay that one packet of a higher priority flow ϕ_h can cause to ϕ_i can be computed by multiplying the number of sub-packets of ϕ_h, with the full number of slots (augmented with respective pauses), i.e. $\omega_h \cdot (\alpha + d_\Delta)$.

After computing the interference that one packet of ϕ_h can cause to ϕ_i, now we need to compute the maximum number of packets of ϕ_h that can interfere with one packet of ϕ_i. An assumption that each two consecutive packets of ϕ_h interfering with ϕ_i must be at least T_h apart may be unsafe. This is due to the *indirectly interfering flows* (Definition 5).

▶ **Definition 5** (Indirectly interfering flow). *Consider three flows ϕ_g, ϕ_h and ϕ_i, where $\phi_g \in \mathcal{F}_h^H$ and $\phi_h \in \mathcal{F}_i^H$, but $\phi_g \notin \mathcal{F}_i^H$. Flow ϕ_g is an indirectly interfering flow of ϕ_i.*

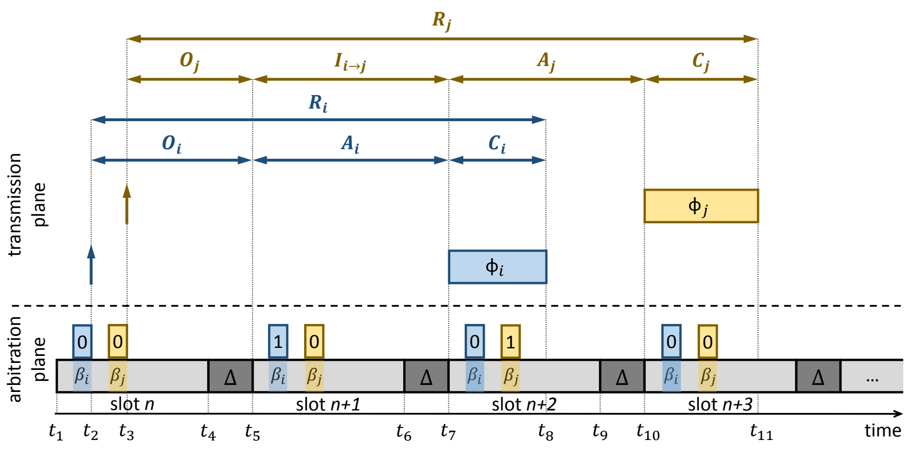


Figure 4 Illustrative example of transmissions of two flows.

Even though ϕ_g cannot cause direct interference to ϕ_i (no common parts of the path), ϕ_g can still cause indirect interference to ϕ_i in the following way: by interfering with ϕ_h, it may cause two consecutive packets of ϕ_h to interfere with ϕ_i within a time interval which is shorter than T_h.

In order to take indirect interference effects into account, from the perspective of ϕ_i, the first occurrence of ϕ_h should be assumed as late as possible, while remaining occurrences should be assumed as early as possible. Under the assumption that ϕ_h is schedulable, the effects of indirect interference on ϕ_i can be modelled with jitter $J_{h\to i}$ (Equation 7), which corresponds to the difference between the latest and the earliest time instants when a packet of ϕ_h can interfere with ϕ_i.

$$J_{h\to i} = \begin{cases} \overbrace{R_h - C_h}^{\text{latest occurrence}} - \overbrace{A_h - d_\Delta}^{\text{earliest occurrence}}, & \text{if } \exists \phi_g \mid \phi_g \in \mathcal{F}_h^H \wedge \phi_g \notin \mathcal{F}_i^H \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

Now, the worst-case interference that ϕ_h causes to one packet of ϕ_i, termed $I_{h\to i}$, can be obtained from Equation 8.

$$I_{h\to i} = \overbrace{\left\lceil \frac{R_i + J_{h\to i}}{T_h} \right\rceil}^{\text{maximum number of packets}} \cdot \overbrace{\omega_h \cdot (\alpha + d_\Delta)}^{\text{per-packet interference}} \tag{8}$$

In Equation 7 and Equation 8, the terms R_h and R_i denote the WCTTs of ϕ_i and ϕ_h, respectively.

Finally, the WCTT of ϕ_i can be obtained from Equation 9, which should be solved iteratively, until reaching a fixed converging point (if it exists).

$$R_i = O_i + A_i + C_i + \sum_{\forall \phi_h \in \mathcal{F}_i^H} I_{h\to i} \tag{9}$$

An illustrative example of transmissions of two flows ϕ_i and ϕ_j, and their WCTT components are shown in Figure 4.

5.3 Advanced SBT-NoC Variants

From the previous discussion it is noticeable that R_i is to a large extent affected by the duration of the arbitration slot α. This is because O_i and A_i directly depend on α, while the interference from higher priority flows is inflicted in multiples of $\alpha + d_\Delta$ intervals. Shorter α would lead to shorter O_i and A_i, but also to increased transmission times due to sequential transmissions of numerous sub-packets. On the other hand, longer α would increase O_i and A_i, but would lead to fewer sub-packets and more efficient transmissions.

Basic SBT-NoC operates under the assumption that an arbitration slot size α is fixed, and consists of z arbitration intervals, each dedicated to a single flow, i.e. $\alpha = z \cdot d_B$. However, based on the workload characteristics, in some scenarios it may be beneficial to either increase or decrease the duration of the arbitration slot. In this section, we present and discuss two advanced SBT-NoC variants, which allow to modify the size of the arbitration slot.

5.3.1 Advanced SBT-NoC with Slot Extension

One viable strategy to extend the duration of the arbitration slot is to introduce empty (non-used) arbitration intervals. These intervals should be added after the used, per-flow dedicated intervals. Therefore, assuming that γ intervals are added, the extended arbitration slot α_E consists of $z + \gamma$ intervals, of which only the first z are being used. The duration of the extended arbitration slot is $\alpha_E = (z + \gamma) \cdot d_B$.

Regardless of the number of added arbitration intervals, the worst-case analysis can be performed in the same way as for the basic SBT-NoC, with the only difference that instead of the basic arbitration slot α, the extended arbitration slot α_E should be used. This SBT-NoC variant is evaluated in Section 6 (Experiment 3).

Note that, an alternative approach for increasing the duration of the arbitration slot to a desired value α_E is to reduce the bus frequency and in that way increase the bus read/write latency to $d_B^* > d_B$. Such an approach can in fact be perceived as the basic SBT-NoC, because in this case the equality $\alpha_E = z \cdot d_B^*$ would remain valid.

5.3.2 Advanced SBT-NoC with Slot Reduction

One viable strategy to reduce the duration of the arbitration slot is to allow *arbitration interval sharing* among different lower-priority flows. This allows to achieve $\alpha_R < z \cdot d_B$, where α_R denotes the desired arbitration slot length. This SBT-NoC variant is explained in detail in the reminder of this section and evaluated in Section 6 (Experiment 2).

5.3.2.1 Arbitration Mechanism (Advanced SBT-NoC with Slot Reduction)

In this variant, each flow ϕ_i has two additional parameters. The first is ν_i, which defines how frequently ϕ_i participates in the arbitration. For example, $\nu_i = 1$ means that ϕ_i participates in every arbitration slot, $\nu_i = \frac{1}{2}$ every second, $\nu_i = \frac{1}{4}$ every fourth, etc. The closer ν_i is to 1, the more frequently ϕ_i is able to participate in the arbitration, and consequently, the smaller its WCTT is (analysed in Section 5.3.2.3). Therefore, we assume that ν values are assigned to flows according to their priorities (and indexes) non-increasingly. Note that the basic variant is a special case of this advanced variant where $\nu_i = 1, \forall \phi_i \in \Phi$.

The second parameter is $\theta_i \in [0, 1, ... \frac{1}{\nu_i})$, and together with ν_i, it defines exactly in which arbitration slots ϕ_i participates. The arbitration eligibility condition for ϕ_i is expressed with Equation 10, where n denotes the number of an arbitration slot.

$$n - \lfloor n \cdot \nu_i \rfloor \cdot \frac{1}{\nu_i} = \theta_i \qquad (10)$$

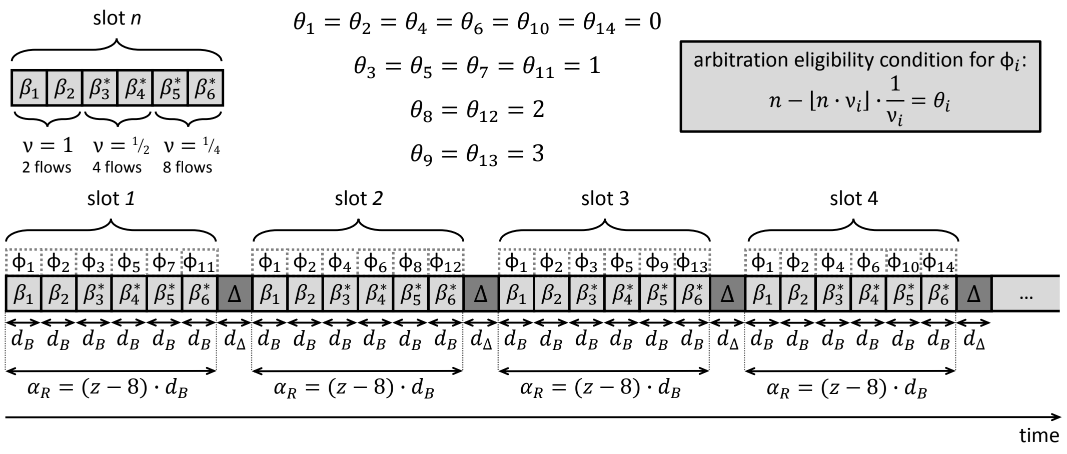


Figure 5 Illustrative arbitration example of advanced SBT-NoC with slot reduction.

In Figure 5 is illustrated an example of 14 flows $\phi_1 - \phi_{14}$. Flows ϕ_1 and ϕ_2 have $\nu_1 = \nu_2 = 1$. The next four flows $\phi_3 - \phi_6$ have $\nu_3 = \nu_4 = \nu_5 = \nu_6 = \frac{1}{2}$. Finally, the remaining eight flows $\phi_7 - \phi_{14}$ have $\nu_7 = ... = \nu_{14} = \frac{1}{4}$. Moreover, θ values are also illustrated in Figure 5. Flows participate only in slots for which their arbitration eligibility condition is fulfilled. For example, ϕ_1 and ϕ_2 participate in all slots, ϕ_3 and ϕ_5 in odd ones, ϕ_4 and ϕ_6 in even ones, ϕ_7 and ϕ_{11} in slots $\{1, 5, 9, 13, ...\}$, ϕ_8 and ϕ_{12} in slots $\{2, 6, 10, 14, ...\}$, etc.

5.3.2.2 Packet Splitting, Transmission Mechanism and Transmission Latencies (Advanced SBT-NoC with Slot Reduction)

The packet splitting in this variant is similar to that of the basic one. For flow ϕ_i, the maximum sub-packet payload size $\widehat{\sigma}_i$ and the number of sub-packets ω_i can be obtained from Equations 2-3 (Section 5.2.3), where α is replaced by α_R.

The transmission mechanism in this variant is also very similar to the one from the basic variant. After a flow receives a transmission permission during one arbitration slot, a transfer of one of its (sub-)packets is accommodated in the subsequent slot. One important difference from the basic variant is that, for flows with $\nu < 1$, two successive arbitration slots are separated from each other by $\frac{1}{\nu_i} - 1$ slots, and hence the corresponding transmission slots will be separated from each other as well. This implies that transmission latencies between the basic and this variant may differ, and assuming the latter case, the transmission latency of ϕ_i can be obtained from Equation 11. For flows with $\nu = 1$, Equation 11 becomes Equation 4.

$$C_i = \overbrace{(\omega_i - 1) \cdot (\alpha_R + d_\Delta) \cdot \frac{1}{\nu_i}}^{\text{transmission of first } \omega_i - 1 \text{ sub-packets}} \overbrace{+ (|\mathcal{L}_i| - 1) \cdot d_R + |\mathcal{L}_i| \cdot d_L + \left(\left\lceil \frac{\sigma_i^{\dashv}}{\sigma_F} \right\rceil + 1 \right) \cdot d_L}^{\text{transmission of the last sub-packet}} \quad (11)$$

5.3.2.3 Worst-case Analysis (Advanced SBT-NoC with Slot Reduction)

In this section, we provide a method to obtain upper-bounds on WCTTs of flows in advanced SBT-NoC with slot reduction. Several components constitute the WCTT, and in order to derive a safe upper bound, we need to cover all of them.

Recall from Section 5.2.4 that O_i corresponds to the time interval between a packet release and a beginning of the next slot when ϕ_i can participate in the arbitration. In the worst-case, a packet may arrive just after its arbitration interval, and has to wait until the next slot in which it can participate in the arbitration. This is covered with Equation 12, where i stands for the index of the arbitration interval of ϕ_i in α_R. For flows with $\nu = 1$, Equation 12 becomes Equation 5.

$$O_i = \overbrace{\alpha_R - i \cdot d_B + d_\Delta}^{\text{until beginning of next slot}} + \overbrace{\left(\frac{1}{\nu_i} - 1\right) \cdot (\alpha_R + d_\Delta)}^{\text{until beginning of next } \phi_i\text{-eligible slot}} \tag{12}$$

The worst-case delay of acquiring a transmission permission and preparing a packet for transmission, denoted by A_i, requires a single slot (augmented by the pause). This term is the same as in basic SBT-NoC (Equation 6), where α is replaced by α_R.

The last component contributing to the WCTT of ϕ_i is the higher priority interference. Let us first discuss the effects of indirect interferences. Similar to the basic variant, the effects of the indirect interference from indirectly interfering flows to ϕ_i via ϕ_h can be modelled with jitter $J_{h\to i}$. The term $J_{h\to i}$ can be computed as before (Equation 7), because, relative to a packet release of ϕ_h, terms $R_h - C_h$ and $A_h - d_\Delta$ cover the latest and the earliest time instants, respectively, when a packet of ϕ_h may interfere with ϕ_i.

Now, let us obtain the interference that one packet of higher-priority flow ϕ_h can cause to ϕ_i. We have to analyse several cases:

Case 1 ($\nu_h = \nu_i = 1$): In this scenario, ϕ_i can suffer interference only from flows with $\nu = 1$. From the perspective of ϕ_i, the system behaves in the same way as the basic SBT-NoC, and the maximum interference caused by ϕ_h to a packet of ϕ_i can be obtained from Equation 8, where α is replaced by α_R.

Case 2 ($\nu_h = 1 \ \wedge \ \nu_i < 1 \ \wedge \ \nexists \phi_g \mid \phi_g \in \mathcal{F}_h^H \wedge \phi_g \notin \mathcal{F}_i^H$): In this scenario, ϕ_i does not suffer indirect interference via ϕ_h. Therefore, apart from flows in $\mathcal{F}_i^H$, there exist no other flows which can disrupt successive transmission requests of sub-packets of ϕ_h. This allows to compute the maximum interference from ϕ_h to a packet of ϕ_i by considering that sub-packets of ϕ_h traverse in consecutive slots (Equation 13).

$$I_{h\to i} = \overbrace{\left\lceil \frac{R_i}{T_h} \right\rceil}^{\text{maximum number of packets}} \overbrace{\cdot \lceil \omega_h \cdot \nu_i \rceil \cdot \frac{1}{\nu_i} \cdot (\alpha_R + d_\Delta)}^{\text{per-packet interference}} \tag{13}$$

In Equation 13, a multiplication by ν_i, a ceiling operator, and a division by ν_i are needed, because ϕ_i participates in the arbitration in every $\frac{1}{\nu_i}$th slot.

Case 3 ($\nu_h = \nu_i \wedge \theta_h \neq \theta_i$): In this scenario, ϕ_i and ϕ_h participate in different arbitration slots and hence ϕ_h cannot cause interference to ϕ_i (Equation 14).

$$I_{h\to i} = 0 \tag{14}$$

Case 4 (All other scenarios): In these scenarios, it is not safe to assume that sub-packets of ϕ_h are transmitted in consecutive slots, either due to the existence of indirectly interfering flows, or due to $\nu_h < 1$. Separated transmissions of successive sub-packets of ϕ_h may cause more interference to ϕ_i, than what would otherwise be caused by their transmissions in consecutive slots. A safe assumption is that, as long as ϕ_h has sub-packets ready for transmission, it will participate in the same arbitration slots with ϕ_i, and after each of them, transmit a single sub-packet. By following this reasoning, an upper-bound on the interference from ϕ_h to a packet of ϕ_i can be obtained by solving Equation 15.

$$I^{\circ}_{h\to i} = \overbrace{\left\lceil \frac{R_i + J_{h\to i}}{T_h} \right\rceil}^{\text{maximum number of packets}} \overbrace{\cdot \, \omega_h \cdot \frac{1}{\nu_i} \cdot (\alpha_R + d_\Delta)}^{\text{per-packet interference}} \tag{15}$$

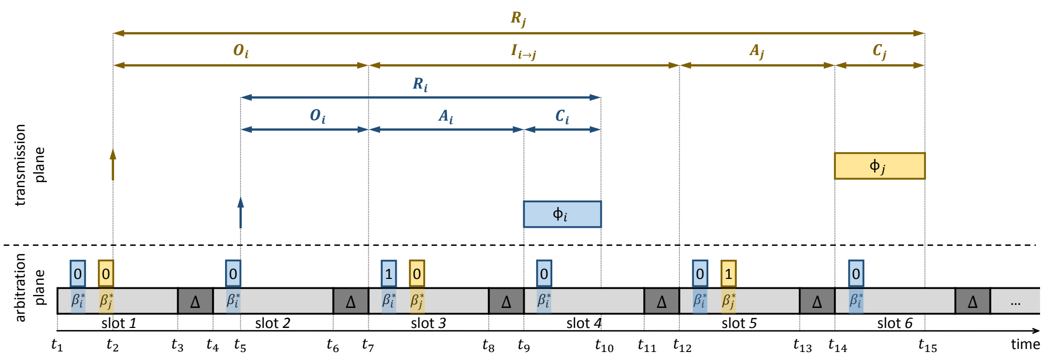


Figure 6 Illustrative example of transmissions of two flows.

Additionally, assuming that ϕ_h is schedulable, the distance between the first and the last sub-packets of its one packet is limited by R_h. Thus, by considering that sub-packets of ϕ_h may interfere with ϕ_i during the entire interval R_h, yet another upper bound on the interference can be derived (Equation 16).

$$I^{\bullet}_{h\to i} = \overbrace{\left\lceil \frac{R_i + J_{h\to i}}{T_h} \right\rceil}^{\text{maximum number of packets}} \cdot \overbrace{\left\lceil \left\lceil \frac{R_h}{\alpha_R + d_\Delta} \right\rceil \cdot \nu_i \right\rceil \cdot \frac{1}{\nu_i}(\alpha_R + d_\Delta)}^{\text{per-packet interference}} \tag{16}$$

Note that Equation 16 is derived using similar reasoning to that of Equation 13. The difference is that instead of ω_h slots, it is conservatively assumed that $\left\lceil \frac{R_h}{\alpha_R + d_\Delta} \right\rceil$ slots are needed to transfer one packet of ϕ_h.

Since both these bounds are safe, the minimum of them can be used (Equation 17).

$$I_{h\to i} = \min\{I^{\circ}_{h\to i}, I^{\bullet}_{h\to i}\} \tag{17}$$

Finally, the WCTT of ϕ_i can be computed by summing all components, as in the basic variant (Equation 9).

In Figure 6 are shown transmissions of two flows ϕ_i and ϕ_j for the advanced SBT-NoC with slot reduction, where $\phi_i \in \mathcal{F}^H_j$, $\nu_i = 1$, $\nu_j = \frac{1}{2}$ and $\theta_j = 1$. Moreover, the components contributing to the WCTTs of ϕ_i and ϕ_j are also illustrated.

6 Experimental Evaluation

In this section, we present the results of the experimental evaluation of SBT-NoC. The relevant NoC parameters which are common to all experiments are summarised in Table 1. The experiment-specific parameters are separately introduced in the context of each experiment. An asterisk sign denotes a randomly generated value, assuming a uniform distribution. Flow source and destination cores/routers are assigned randomly, with a restriction that they have to be different entities, i.e. $\forall \phi_i \in \Phi : \mu_i^{src} \neq \mu_i^{dst} \wedge \rho_i^{src} \neq \rho_i^{dst}$.

6.1 Experiment 1: SBT-NoC Run-time Performance Evaluation

In order to evaluate the performance of SBT-NoC, we have implemented a simulator of a multiprocessor platform. The simulator supports two different types of NoCs: (i) a regular NoC architecture with single-channel ports and 2-flit buffers, utilising the fixed-priority

Table 1 Analysis and simulation parameters.

NoC topology	2-D mesh
Routing mechanism	X-Y
Router frequency (ψ)	100MHz
Router latency (d_R) + link latency (d_L)	3 + 1 cycles
Bus writing/reading latency (d_B)	1 cycle
Pause between arbitration slots (d_Δ)	0 cycles
Link width = flit size (σ_F)	4B
Flow source core/router (μ_i^{src} / ρ_i^{src})	Random
Flow destination core/router (μ_i^{dst} / ρ_i^{dst})	Random
Flow deadline (D_i) = flow period (T_i)	[10ms - 50ms]*
Flow priority assignment policy	Rate monotonic

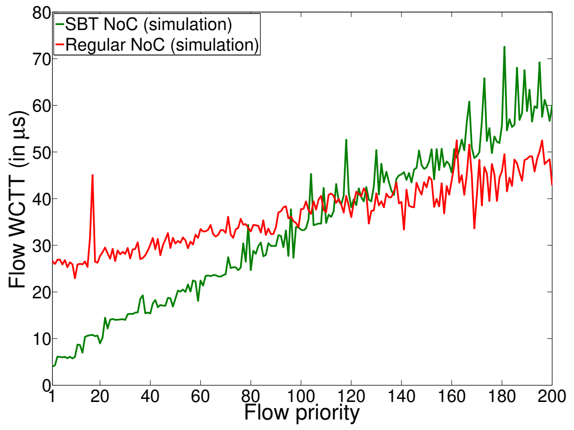


Figure 7 WCTTs of regular NoC (simulation) and basic SBT-NoC (simulation).

packet-level arbitration mechanism (in Section 1 referred to as the *basic NoC*), and (ii) the SBT-NoC with the basic arbitration variant (introduced in Section 5.2). In both cases, NoC parameters are identical to those from Table 1. The assumed NoC size is 4 × 4.

The workload characteristics are as follows. There exist 200 flows with unique priorities assigned with the rate-monotonic policy. Smaller numbers represent higher priorities. The flow periods and deadlines are as specified in Table 1. Regarding flow sizes, the following trend applies: higher priority flows have smaller sizes. The highest priority flow has the smallest payload size of 500B, the lowest priority flow has the biggest payload size of 10kB, and the sizes of intermediate flows are assigned equidistantly.

We run the simulations of the two aforementioned approaches, each for 100 seconds of simulated time. For each approach, we recorded the observed WCTTs of all 200 flows.

The results are illustrated in Figure 7. It is evident that basic NoCs do not have efficient mechanisms to leverage high priorities to achieve low WCTTs, which is one of the basic real-time requirements. In fact, when a lower priority flow starts its transmission through a shared router, it can block any later arriving higher priority flow for the duration of its entire traversal through that router. During a single transmission, a higher priority flow can experience blocking from multiple lower priority flows across different routers. Consequently, there exists an almost negligible difference in WCTTs of flows with highest and lowest priorities, despite the fact that higher priority ones have significantly smaller

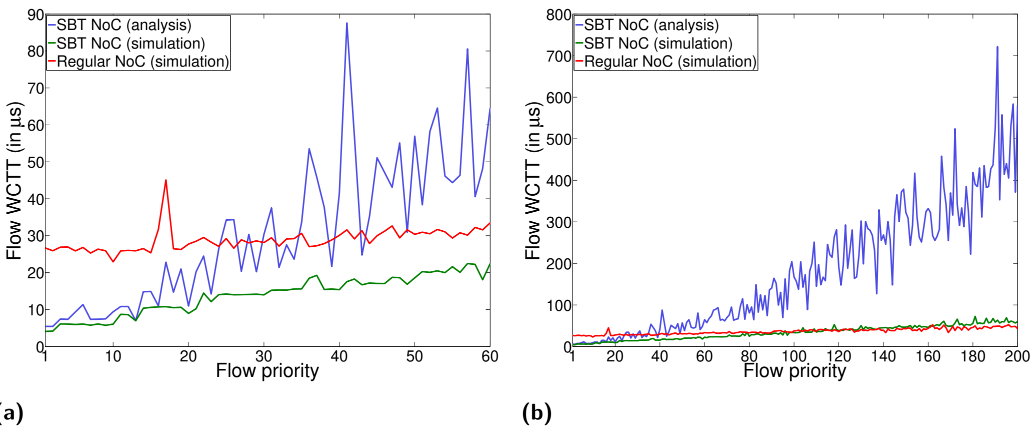


Figure 8 WCTTs of regular NoC (simulation) and basic SBT-NoC (simulation & analysis).

sizes. This result coincides with the statement from Section 1 that basic NoCs without any enhancements do not represent satisfactory solutions for the real-time domain, and further motivates research activities in the area of real-time oriented NoCs.

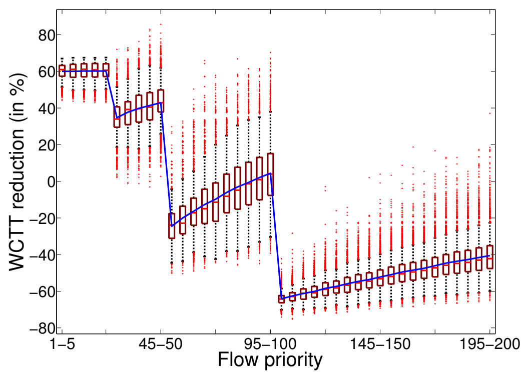
SBT-NoC demonstrates a significantly different behaviour. For high priority flows, SBT-NoC provides substantially smaller WCTTs than a regular NoC, and for the highest priority ones WCTTs are even several times smaller. These WCTT reductions are achieved at the expense of the lower priority traffic, and for low priority flows SBT-NoC provides bigger WCTTs than a regular NoC. The observed WCTTs suggest that SBT-NoC does not have negative effects on the run-time performance (no unusual and unexpected spikes in WCTTs), thus we conclude that SBT-NoC fulfils the first two objectives from Section 4.

Additionally, we derived the analytical WCTT upper-bounds by applying the method proposed in Section 5.2.4. A comparison of analytical and simulation results is illustrated in Figure 8a, where the focus is on 30% of flows with high priorities, (priorities 1 to 60). Interestingly, for the highest priority flows, even the SBT-NoC analytical method provides smaller WCTTs than the regular NoC simulations. The trends remain until priority 30, and imply that SBT-NoC, in conjunction with an analysis method, can produce low **bounded**[4] WCTTs of high priority flows. This means that the third objective from Section 4 is fulfilled.

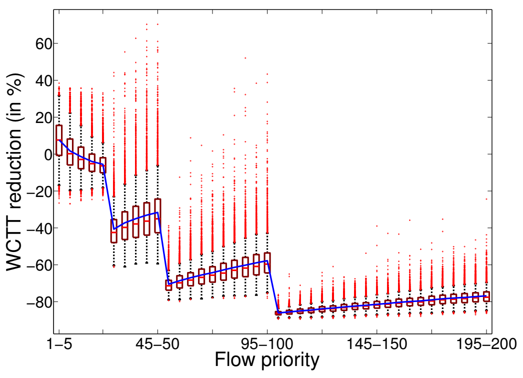
For completeness, we have extended the observation interval to include all flows (Figure 8b). Unsurprisingly, as priorities decrease (i.e. bigger numbers on the X-axis), a difference between analytical and simulation results grows. This can be attributed to the fact that simulations are performed for only a limited time, which makes it unlikely that all worst-case scenarios were indeed captured. Moreover, the analysis method might contain a certain degree of pessimism, and reducing it is a potential future work activity.

Summary. SBT-NoC provides efficient means to achieve low WCTTs of high-priority flows at the expense of increased WCTTs of low priority ones. SBT-NoC does not have a negative effect on the run-time performance, which makes it a promising solution for soft real-time systems, where good performance might also be one of the requirements. Perhaps even more importantly, SBT-NoC provides low bounded WCTTs of high priority flows, which makes it a viable choice for hard real-time systems as well.

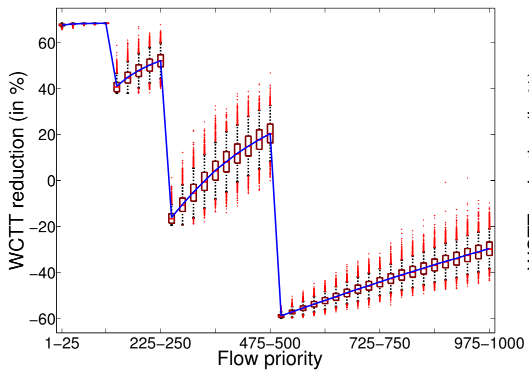
[4] Recall that for a regular NoC there exists no worst-case analysis method.



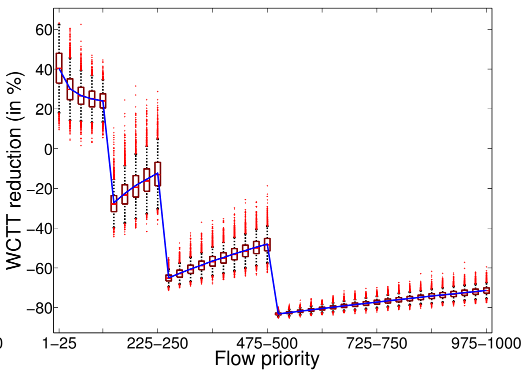
(a) C1: $\sigma_i \in [8B - 256B], \forall\phi_i \in \Phi$.



(b) C2: $\sigma_i \in [1kB - 4kB], \forall\phi_i \in \Phi$.



(c) C3: $\sigma_i \in [8B - 256B], \forall\phi_i \in \Phi$.



(d) C4: $\sigma_i \in [1kB - 4kB], \forall\phi_i \in \Phi$.

Figure 9 Relative ruction in analytically derived WCTTs of the advanced SBT-NoC with slot reduction – "A" against the basic SBT-NoC – "B".

6.2 Experiment 2: SBT-NoC Analytical Evaluation (Synthetic Workload)

In this experiment, we compare analytical results of the basic SBT-NoC variant – "B", and the advanced SBT-NoC variant with slot reduction – "A", for different workload configurations. "A" is configured as follows: 12.5% flows with the highest priorities have $\nu = 1$, the next 12.5% have $\nu = \frac{1}{2}$, the next 25% have $\nu = \frac{1}{4}$ and the remaining 50% have $\nu = \frac{1}{8}$. Phases (θ values) are derived from flow priorities in the following way: $\theta_i = P_i - \lfloor P_i \cdot \nu_i \rfloor \cdot \frac{1}{\nu_i}$.

The NoC size is extended to 8x8. Flow deadlines, periods, priorities, source and destination cores are assigned in the same way as in Experiment 1 (Table 1). Moreover, the evaluation includes 2 different setups for the workload size: (i) $z = 200$ flows and (ii) $z = 1000$ flows, as well as 2 different setups for the payload size: (i) $\sigma_i \in [8B - 256B]$ and (ii) $\sigma_i \in [1kB - 4kB]$. Assuming a certain payload size range, flow payloads are randomly generated values (a uniform distribution). The combinations of payload and workload size setups produce 4 distinctive evaluation configurations (C1-C4). For each of them, we generate 1000 flow-sets and analytically obtain WCTT upper-bounds with both evaluated SBT-NoC variants. We compare derived bounds by calculating the relative reduction in WCTTs achieved by "A" against "B". In cases where "B" outperforms "A", the WCTT reduction has negative values.

The results are illustrated in Figure 9. It is visible that a selection of the parameter ν has a significant impact on WCTTs. Therefore, let us analyse different sub-domains independently. For $\nu = 1$ (highest priority flows), in all configurations except C2, "A" derives tighter bounds. This is expected, because "A" utilises shorter arbitration slots, and as discussed in Section 5.3.2.3, this reduces several WCTT components. As a number of flows increases, so do the improvements, because the difference in the durations of arbitration slots of "A" and "B" also grows. On the other hand, the increase in payload sizes leads to reduced improvements. This is because shorter slots yield more sub-packets, which limits the effects of a pipelined flit traversal and increases transmission overheads (more header and tail flits). These factors cause longer transmission latencies of analysed flows, and also inflate the interference they suffer from higher priority flows.

For flows with $\nu = \frac{1}{2}$, the observations regarding the effects of flow numbers and payload sizes on WCTT improvements are similar to those for flows with $\nu = 1$. In configurations C1 and C3, "A" is beneficial for flows with $\nu = \frac{1}{2}$. This implies that in scenarios with numerous flows and relatively small payload sizes, a strategy of assigning $\nu = \frac{1}{2}$ to flows with intermediate priorities may still lead to more favourable conditions for them, than what they could experience in "B". Another interesting observation from all evaluated configurations is that, within an observed sub-domain ($\nu = \frac{1}{2}$), improvements of "A" over "B" increase with decreasing priorities. This is because in "B" a flow can suffer interference from all higher priority flows sharing a part of the path with it, while in "A" that is not the case. In fact, if two flows in "A" have the same parameter ν, they can interfere only if they have the same θ. For $\nu = \frac{1}{4}$, "A" outperforms "B" only in C3, and in rare cases in C1.

Finally, for flows with $\nu = \frac{1}{8}$, in none of scenarios "A" produces smaller WCTTs than "B". This is expected, because the improvements of "A" over "B" for high priority flows were in fact achieved at the expense of increased latencies of the low priority traffic.

Summary. Variant "A" allows to even further reduce WCTTs of highest priority flows by decreasing a duration of an arbitration slot via an arbitration interval sharing among low priority flows. The improvements against variant "B" are the most significant for the highest priority flows ($\nu = 1$), while depending on the nature of the workload, significant WCTT reductions can also be achieved for flows with $\nu < 1$. Of course, WCTT reductions for high priority flows are achieved at the expense of increased WCTTs of the low priority traffic.

6.3 Experiment 3: SBT-NoC Analytical Evaluation (Use Case of Autonomous Driving Vehicle Application)

In this experiment, we perform the analytical evaluation of SBT-NoC. The workload is modeled after the use-case of the autonomous driving vehicle application [22]. The use-case consists of 33 functionalities producing 38 traffic flows in total. For a more detailed description of the use-case, a reader is advised to consult the work of Shi et al. [22].

The evaluation is performed in the following way. First, assuming the basic SBT-NoC variant – "B", the WCTT upper bounds of all flows are analytically obtained. Then, the same is performed for the two configurations of the advanced SBT-NoC with slot extension. The first one, referred to as "E10", has the slot length α_{E10} which is 10 times bigger than the slot length α_B of the basic variant, i.e. $\alpha_{E10} = 10 \cdot \alpha_B$. The second one, referred to as "E100", has the slot length α_{E100}, where $\alpha_{E100} = 10 \cdot \alpha_{E10} = 100 \cdot \alpha_B$. The incentive to evaluate "E10" and "E100" comes from the fact that there exist only 38 flows in this use-case, and the approach "B" is likely to lead to inefficient sequential transmissions of lots of small sub-packets, causing large WCTTs. Finally, in order to compare the performance of SBT-NoC with some other available approaches for real-time NoCs, we included priority preemptive NoCs with flit-level arbitration and per flow dedicated virtual channels in this evaluation [21], hereafter referred to as "PP". The WCTTs of all flows are obtained using the latest available analysis for such NoCs [17].

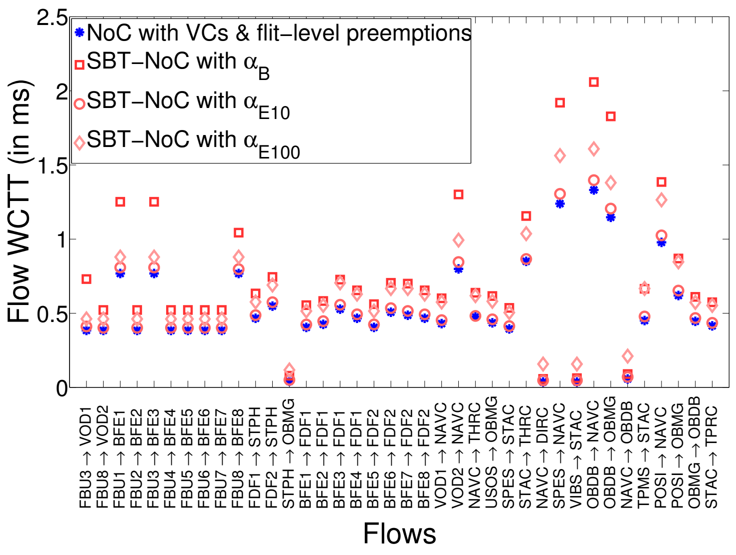


Figure 10 Analytically derived WCTTs of flows of the autonomous driving vehicle application [22].

The evaluation results are illustrated in Figure 10. As expected “B” displays the worst performance. Even though it offers short arbitration slots and short out-of-interval-arrival penalties (the first two terms in Equation 9), the transmissions are performed via numerous short sequentially transmitted sub-packets, thus causing large transmission latencies of entire packets (the third term in Equation 9). On the other hand, “E10” utilises 10 times longer arbitration slots, which may lead to 10 times longer out-of-interval-arrival penalties. However, “E10” performs transmissions with fewer sub-packets, which causes significantly shorter transmission latencies of entire packets. Consequently, “E10” produces 17.47 – 43.75% smaller WCTTs than “B”. The average WCTT reduction is 26.25%.

In the case of “E100”, arbitration slots and out-of-interval-arrival penalties are 10 times larger than in “E10”. On the other hand, “E100” performs more efficient (shorter) packet transmissions via fewer larger sub-packets, however, the achieved gains cannot compensate for the penalties arising from the increased slot size. Therefore, “E100” produces 8.72 – 233.28% larger WCTTs than “E10”. The average WCTT increase is 39.37%.

Finally, compared to the priority-preemptive approach “PP”, the best performing SBT-NoC scheme “E10” produces 1.17 – 31.21% larger WCTTs. The average WCTT increase is 7.33%. However, it is fair to point out that “PP” has more substantial hardware requirements than SBT-NoC schemes. Specifically, it operates under the assumption that there exist per-flow dedicated virtual channels in each of the traversed router ports, and that the arbitration is performed on a flit level. Implementing these features requires a sophisticated in-router logic and buffer space, which are typically not available in commercial NoCs. On the other hand, SBT-NoC requires a dedicated bus-based arbitration mechanism, which we believe is less costly and less demanding to implement.

Please note that “E10” may not be the most efficient SBT-NoC configuration for this use-case. It is only the best performing of the three SBT-NoC variants covered in this preliminary evaluation. In order to uncover the full potential of SBT-NoC, more detailed evaluations are necessary, and these activities are a potential future work.

Summary. A decision regarding arbitration slot sizes should be thoughtfully derived, because it significantly impacts the efficiency of SBT-NoC. The important aspects are platform architecture properties and a workload structure. Too short slots may lead to packet fragmentation into numerous sequentially transmitted sub-packets, which may cause longer WCTTs. On the other hand, too large slots may lead to significantly longer arbitration procedures, and longer out-of-interval-arrival penalties, both contributing to longer WCTTs. When compared with the priority-preemptive NoC scheme, the best of the three evaluated SBT-NoC approaches shows comparable results, and given the substantially higher hardware requirements associated with the former scheme, we believe that SBT-NoC is an attractive alternative, and a promising communication solution for real-time NoCs.

7 Conclusions and Future Work

In this work, we presented SBT-NoC – a slot-based transmission protocol for NoCs, and the accompanying worst-case analysis. SBT-NoC features contention-less slot-based transmissions, arbitrated via a protocol running on a dedicated network medium. SBT-NoC provides bounded low latencies of high-priority time-critical flows, at the expense of low priority ones. In this work, an SBT-NoC implementation via a dedicated bus medium was presented.

Moreover, this work includes a preliminary experimental evaluation of SBT-NoC. The initial results suggest that the proposed approach fulfils several important requirements of the real-time domain, and that it presents a viable choice for interconnect mediums in next-generation real-time-oriented multiprocessors. SBT-NoC offers a plethora of configuration options, of which only few have been evaluated in this work. Our future work plans include investigations of alternative technologies for an arbitration medium (e.g. a NoC interconnect), a practical implementation, and a design space exploration for deriving the most efficient SBT-NoC configurations for given platform and workload characteristics. This includes finding answers to the following questions: how to assign priorities, configure slot durations and assign parameters ν and θ?

References

1 Adapteva. *Epiphany Architecture.* URL: `www.adapteva.com/docs/epiphany_arch_ref.pdf`.

2 L. Benini and G. De Micheli. Networks on chips: a new SoC paradigm. *The Comp. J.*, 35(1):70–78, January 2002.

3 Josef Berwanger, Martin Peller, and Robert Griessbach. byteflight - A new protocol for safety-critical applications. In *FISITA World Automotive Congress*, 2000.

4 T. Bjerregaard and J. Sparso. Implementation of guaranteed services in the MANGO clockless network-on-chip. *IEE Proc. - Computers & Digital Techniques*, 153(4):217–229, July 2006.

5 W.J. Dally. Virtual-channel flow control. *Trans. Parall. & Distr. Syst.*, 3(2):194–205, March 1992.

6 W.J. Dally and C.L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *Trans. Computers*, 1987.

7 Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Syst. J.*, 2007.

8 K. Duraisamy and P. P. Pande. Enabling High-Performance SMART NoC Architectures Using On-Chip Wireless Links. *Trans. Very Large Scale Integration Syst.*, 2017.

9 K. Goossens, J. Dielissen, and A. Radulescu. AEthereal network on chip: concepts, architectures, and implementations. *IEEE Design & Test Computers*, 2005.

10 Tim Harde, Matthias Freier, Georg von der Brüggen, and Jian-Jia Chen. Configurations and Optimizations of TDMA Schedules for Periodic Packet Communication on Networks on Chip. In *26th RTNS*, 2018.

11 Leandro Soares Indrusiak, Alan Burns, and Borislav Nikolić. Buffer-aware bounds to multi-point progressive blocking in priority-preemptive NoCs. In *21st DATE*, 2018.

12 Intel. *Single-Chip-Cloud Computer*, 2010. URL: `www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-single-chip-cloud-article.pdf`.

13 E. Kasapaki, M. Schoeberl, R. B. Sørensen, C. Müller, K. Goossens, and J. Sparsø. Argo: A Real-Time Network-on-Chip Architecture With an Efficient GALS Implementation. *Trans. Very Large Scale Integration Syst.*, 2016.

14 N. K. Kavaldjiev and G. J. M. Smit. A Survey of Efficient On-Chip Communications for SoC. In *4th Symp. Emb. Syst.*, 2003.

15 M. Liu, M. Becker, M. Behnam, and T. Nolte. A tighter recursive calculus to compute the worst case traversal time of real-time traffic over NoCs. In *22nd ASPDAC*, 2017.

16 R. Makowitz and C. Temple. Flexray - A communication network for automotive control systems. In *Int. WS Factory Comm. Syst.*, 2006.

17 Borislav Nikolić, Sebastian Tobuschat, Leandro Soares Indrusiak, Rolf Ernst, and Alan Burns. Real-time analysis of priority-preemptive NoCs with arbitrary buffer sizes and router delays. *Real-Time Syst. J.*, 2018.

18 C. Paukovits and H. Kopetz. Concepts of Switching in the Time-Triggered Network-on-Chip. In *14th RTCSA*, pages 120–129, 2008.

19 Yue Qian, Zhonghai Lu, and Wenhua Dou. Analysis of worst-case delay bounds for best-effort communication in wormhole networks on chip. In *NOCS*, 2009.

20 Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, Andrć Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *J. Syst. Arch.*, 2015.

21 Zheng Shi and A. Burns. Real-Time Communication Analysis for On-Chip Networks with Wormhole Switching. In *NOCS*, 2008.

22 Zheng Shi, Alan Burns, and Leandro Soares Indrusiak. Schedulability Analysis for Real Time On-Chip Communication with Wormhole Switching. *Int. J. Emb. & Real-Time Comm. Syst.*, 2010.

23 Hyojeong Song, Boseob Kwon, and Hyunsoo Yoon. Throttle and preempt: a new flow control for real-time communications in wormhole networks. In *1997 Int. Conf. Parall. Processing*, August 1997.

24 S. Tobuschat and R. Ernst. Real-time communication analysis for Networks-on-Chip with backpressure. In *20th DATE*, 2017.

25 D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C. Miao, J. F. Brown III, and A. Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *MICRO*, 2007.

26 Q. Xiong, F. Wu, Z. Lu, and C. Xie. Extending Real-Time Analysis for Wormhole NoCs. *Trans. Computers*, 66(9), 2017.

27 Qin Xiong, Zhonghai Lu, Fei Wu, and Changsheng Xie. Real-Time Analysis for Wormhole NoC: Revisited and Revised. In *26th ACM Great Lakes Symp. VLSI*, 2016.

Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms

Giovani Gracioli
Technical University of Munich, Germany
Federal University of Santa Catarina, Brazil
g.gracioli@tum.de

Rohan Tabish
University of Illinois at Urbana-Champaign, IL, USA
rtabish@illinois.edu

Renato Mancuso
Boston University, MA, USA
rmancuso@bu.edu

Reza Mirosanlou
University of Waterloo, Canada
rmirosan@uwaterloo.ca

Rodolfo Pellizzoni
University of Waterloo, Canada
rpellizz@uwaterloo.ca

Marco Caccamo
Technical University of Munich, Germany
mcaccamo@tum.de

Abstract

Multiprocessor Systems-on-Chip (MPSoC) integrating hard processing cores with programmable logic (PL) are becoming increasingly common. While these platforms have been originally designed for high performance computing applications, their rich feature set can be exploited to efficiently implement mixed criticality domains serving both critical hard real-time tasks, as well as soft real-time tasks.

In this paper, we take a deep look at commercially available heterogeneous MPSoCs that incorporate PL and a multicore processor. We show how one can tailor these processors to support a mixed criticality system, where cores are strictly isolated to avoid contention on shared resources such as Last-Level Cache (LLC) and main memory. In order to avoid conflicts in last-level cache, we propose the use of cache coloring, implemented in the Jailhouse hypervisor. In addition, we employ ScratchPad Memory (SPM) inside the PL to support a multi-phase execution model for real-time tasks that avoids conflicts in shared memory. We provide a full-stack, working implementation on a latest-generation MPSoC platform, and show results based on both a set of data intensive tasks, as well as a case study based on an image processing benchmark application.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Embedded systems; Computer systems organization → Other architectures

Keywords and phrases Mixed-criticality systems, SoC Heterogeneous platforms, FPGA, real-time computing

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.27

Funding This work has been supported in part by the NSF under grant number CNS-1646383, by ONR N00014-17-1-2783, by NSF grant number CNS 18-15891, by NSERC and by CMC Microsystems. Marco Caccamo was also supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.

Acknowledgements We would like to thank the anonymous reviewers for their valuable feedback, and our shepherd for helping to significantly improve this paper.

1 Introduction

Recently there has been an uptrend in the demand for high-performance real-time applications. The increasing interest in emerging technologies like self-driving cars, drones, cube satellites, and smart manufacturing, to name a few, has determined a shift in the type of workload that has to be considered "real-time" [5]. Traditional CPU-intensive tasks comprise a small percentage of the real-time workload in modern high-criticality systems, while increasingly more memory- and I/O-intensive applications have been brought into the picture. Additionally, hardware manufacturers have anticipated the demand for high-performance embedded systems by introducing increasingly more feature-rich multiprocessor systems-on-chip (MPSoC) platforms.

In the race to provide the future de-facto standard for pervasive high-performance embedded systems, hardware manufacturers have experimentally introduced a plethora of architectural features. A number of these features have a proven track record in the general-purpose computing domain and multiple indicators suggest their long-term adoption in the embedded market [10]. Such features include hardware support for virtualization, the presence of multiple, potentially heterogeneous processing elements, a rich ecosystem of high-bandwidth I/O devices and communication channels, and more recently the co-location of traditional CPUs and programmable logic (PL) implemented using Field Programmable Gate Array (FPGA) technology.

The presence of on-chip "soft" PL, tightly coupled with a group of "hard" embedded CPUs, represents a game-changer for systems that need to be tailored to a well-known application scenario [21]. This is indeed often the case for real-time systems. In fact, this new class of platforms offers the unprecedented ability to define new hardware components to complement the high-performance profile of the embedded cores. If it is possible to devise PL-defined components that mitigate the non-determinism in high-performance CPUs; the result can be an ideal trade-off between processing power and real-time guarantees [21].

In this paper, we study how it is possible to leverage latest-generation partially reconfigurable embedded platforms for a system design that combines high-performance and strict real-time requirements. In our approach, we define multiple *criticality domains* to be intended as subsystems of the computing system. Each criticality domain may be designed with a different trade-off between high-performance and strict temporal determinism. For instance, a high-performance domain may run a general-purpose OS with a complex I/O infrastructure. Conversely, a high-criticality domain is comprised of a Real-Time Operating System (RTOS) supporting time-sensitive applications.

We demonstrate that it is possible to instantiate a critical core of PL-defined components to (i) relieve interference on the shared memory hierarchy and achieve temporal isolation among criticality domains; (ii) support efficient inter-domain communication; (iii) co-locate a traditional task execution model with a multi-phase execution model; and (iv) overcome typical limitations of traditional memory partitioning techniques. In summary, this paper makes the following contributions:

1. We demonstrate that it is possible to leverage partially reconfigurable embedded platforms to instantiate a system where high-performance and time-sensitive applications co-exist under strict temporal isolation. Compared to the ideal case (*i.e.,* task running alone in the system), our set of hardware/software techniques ensures that execution time of

a time-sensitive task does not suffer from the potentially large interference caused by memory-intensive tasks running on different cores (only 6% of an increase in the execution time, instead of a large interference).

2. We design and implement a hardware block, named address translator, that prevents the problem of memory waste when cache partitioning based on page coloring is used.
3. We provide a working implementation on one of the latest-generation partially reconfigurable embedded MPSoCs. Our implementation is full-stack, with adaptations at an OS- and application-level, extensions to a partitioning hypervisor, and generation of PL-defined hardware blocks. We demonstrate the feasibility of the implementation by using a case study on image processing and show the hard real-time bounds achieved by our system design.

Organization of the paper. The remainder of this paper is organized as follows. Section 2 reviews the related work. Section 3 presents the adopted system model and assumptions. Section 4 discusses the design principles and overviews the proposed approaches. Section 5 presents a design space exploration of a modern MPSoC platform through a set of experiments. It also shows the proposed hardware and software design to support mixed-criticality applications on top of the platform. Section 6 details the system implementation with multiple criticality domains. Section 7 presents some experimental results carried out to evaluate the proposed real-time computing framework. Finally, Section 8 states the conclusions and outlines some future work.

2 Related Work

Several recent works have proposed techniques to deal with shared resources in multicore real-time systems at both OS and hypervisor levels. Mancuso *et al.* profiled the source code to extract memory access patterns for each task, allowing frequently-used pages to be locked in cache in order to avoid cache evictions [19]. Combined with cache partitioning based on page coloring, their approach significantly improves predictability. Following the same line, some works used coloring to partition the cache in multicore real-time systems [15, 13, 11]. Other works focused on making DRAM accesses more predictable [38, 14, 17]. MemGuard regulates each core's memory request rate by using hardware performance counters to account for the memory access usage [39]. The work defines a threshold and when the number of memory accesses reaches the threshold, an overflow interrupt is generated to keep the specified memory bandwidth. One assumption for MemGuard is that all cores have access to the same memory bus, while in our work we explore the existence of the programmable logic to define dedicated memory interfaces. We also show how page coloring can co-exist with the programmable logic memories and how to prevent wasting cache space due to page coloring.

The use of hypervisors in multicore real-time systems is a recent trend. Modica et al. proposed a hypervisor-based architecture targeting critical systems similar to ours [22]. Cache partitioning provided spatial isolation, while a DRAM bandwidth reservation mechanism provided temporal isolation. Both cache partitioning and memory reservation mechanisms were implemented in the XVISOR open-source hypervisor [24] and tested in a quad-core ARM A7 processor. Our proposed hypervisor-based approach, instead, uses an MPSoC platform, which gives us the ability to explore other features, such as specific FPGA direct memory access (DMA) blocks (for instance, to handle data transfers between the processing system and programmable logic sides) and data prefetching.

MARACAS addressed shared cache and memory bus contention through multicore scheduling and load-balancing on top of the Quest OS [37]. MARACAS used hardware performance counters information to throttle the execution of threads when memory contention exceeded a certain threshold. The counters were also used to derive an average memory request latency to reduce bus contention. vCAT used the Intel's Cache Allocation Technology (CAT) to achieve core-level cache partitioning for the hypervisor and virtual machines running on top of it [36]. vCAT was implemented in Xen and $LITMUS^{RT}$. Although interesting, this approach is architecture dependent and uses non-real-time basic software support (Linux and Xen).

Kim and Rajkumar proposed a predictable shared cache framework for multicore real-time virtualization systems [16]. The proposed framework introduced two hypervisor techniques (vLLC and vColoring) that enabled cache-aware memory allocation for individual tasks running in a virtual machine. CHIPS-AHOy is a predictable holistic hypervisor [23]. It integrates shared hardware isolation mechanisms, such as memory partitioning, with an observe-decide-adapt loop to achieve predictability and energy and thermal management.

Crespo *et al.* used hardware performance counters within the hypervisor to regulate the memory bandwidth of critical and non-critical cores [6]. The work used control theory to do the regulation and presented a set of experiments to tune the controller parameters. Awan *et al.* proposed a memory regulation mechanism for mixed-criticality applications [2]. Mendez *et al.* also proposed to use FPGA together with a processing system to reduce interference of mixed-criticality applications. However, in their system model, the authors did not consider multicore processors or shared caches [21].

SPM-centric OS combined scratchpad, resource specialization, scheduling of shared resources as well as a three-phase model to achieve predictability in multicore real-time systems [28]. The three-phase model is also used in this work. It consists of a load phase, in which code/data is loaded from main memory to the scratchpad (SPM), an execution phase, and an unload phase in which code/data is unloaded from the SPM to main memory. The model relies on a DMA engine to support the load/unload phases. The idea is to load data/code for a task using a DMA before it starts and to unload it after completion, as depicted in Figure 1. Because the SPM is divided into two halves, while one task is executing in one half, DMA is active on the another one. Up arrows in the figure represent the release times of three tasks. While for simplicity we draw the figure assuming that all load and unload phases take an equal amount of time to complete, in reality, their length can vary on a per-task basis. Note that each job starts executing on the core after it is loaded in the scratchpad and the previous job finishes executing, whichever happens last. Also note that while load phases have higher priority over unloads, at time $t = 3$ (and $t = 5$) an unload must be performed first in order to free Partition A for the successive load phase of task τ_3. If there are multiple ready tasks, the decision of which task to schedule is made when starting a load phase; hence, while τ_1 has a higher priority than τ_3, the latter is executed at time 5 because τ_1 is released right after the start of another load phase at $t = 4$. This behavior causes blocking time on the higher priority task and must be accounted for in a schedulability analysis.

The work in [28] used a time division multiple access (TDMA) arbitration among cores based on a fixed slot size for DMA transfers. Only a single DMA operation (either a task load or unload) was carried out during a slot. The TDMA slot size length was designed so that it was possible to load or unload an entire scratchpad partition within one slot. Hence, it always allowed to transfer an amount of data equal to the largest scratchpad size, which is undesirable if the SPM size is different per-core. On the contrary, in this work the DMA scheduling employs variable memory phase sizes, similarly to what has been described in [33]. Notice, though, that the work in [33] mainly targets single-core processors, and it does not provide a working implementation for multi-core systems.

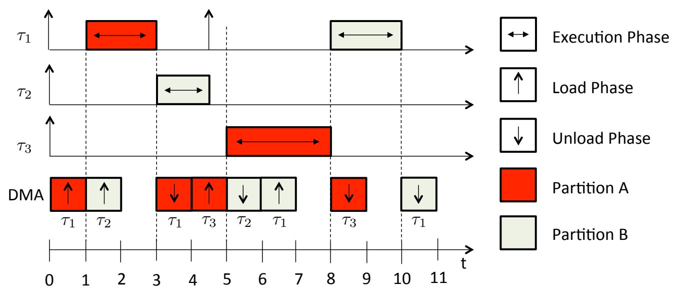


Figure 1 Example of a schedule using the three-phase model.

3 System Model and Assumptions

In this section we summarize the system model and assumptions of the proposed architecture.

3.1 Criticality Domains

Our goal is to implement multiple *criticality domains* on a single multicore SoC. Given C, the total number of cores in the SoC, we target a system with up to C criticality domains, so that each criticality domain is statically assigned to at least one core. One of the key design principles is that criticality domains are isolated from each other, both in time and space [5]. In other words, we minimize the impact that the activity of applications in one criticality domain can have on tasks in a different criticality domain.

Domain Types. Albeit strong isolation exists between criticality domains, each domain may have different requirements in terms of performance, amount of memory resources, and runtime environment. We envision three types of criticality domains. First, a *low-criticality domain* is used to perform I/O with complex devices, processing of large amounts of data, using general-purpose libraries and applications. A low-criticality domain may run a generic operating system (OS) – e.g. Linux – and require a large amount of memory with fast-on-average performance. While applications in this domain are shielded from interference from the rest of the system, no strong temporal guarantees can be expressed due to the best-effort nature of the software stack. Second, a *high-criticality domain* consolidates all the hard real-time tasks of the system and interfaces with simple I/O devices. In this domain, applications have strong timing guarantees. Finally, a third *mid-criticality domain* is used to process tasks with intermediate criticality. Within this domain, and unlike the low-criticality domain, temporal guarantees for real-time tasks are still provided; however, the degree of hardware resource isolation offered to the mid-criticality domain is lower when compared to the high-criticality one. The number of cores allocated to high- and mid-criticality domains is $M \leq C$.

3.2 Processor and Programmable Logic

We consider an embedded MPSoC platform with two main subsystems, the processor subsystem (PS) and the programmable logic (PL), and a communication engine, as exemplified by Figure 2.

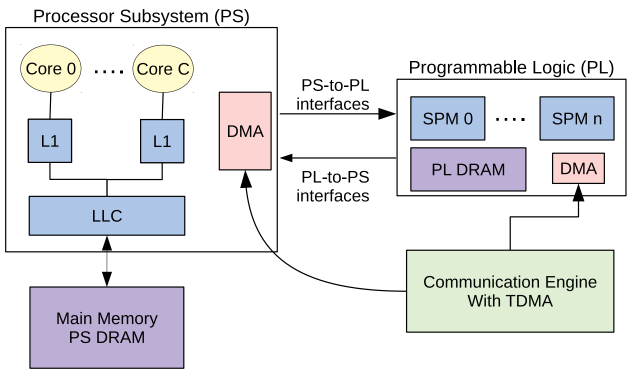


Figure 2 Overview of the platform with the main components..

Processor Subsystem (PS). The PS has a multicore embedded processor with C cores. Each core has a private Level-1 (L1) cache, and all the cores share a Level-2 (L2) cache which is also the last-level cache (LLC). While other organizations for the memory hierarchy are possible, we adopt a widespread model in modern multicore embedded systems. A key difference in the considered class of partially reconfigurable systems is the following. A miss in LLC causes a memory transaction to be performed towards either the main memory (PS DRAM) or the Programmable Logic (see Figure 2). This behavior depends on the exact physical memory address being accessed. Because our goal is to define strongly isolated criticality domains, we assume that hardware support for virtualization exists in the PS.

Programmable Logic (PL). The PL is an on-chip block of Field-Programmable Gate Array (FPGA) cells that coexists with the embedded PS cores. We consider systems where high-bandwidth, low-latency memory interfaces connect the PS to the PL and vice-versa, as demonstrated in Figure 2. Such a feature for the emerging class of partially reconfigurable embedded systems is of crucial importance, and manufacturers [35] are well aware of it. While we assume that one or more PS-PL interfaces exist, it cannot be assumed that at least C interfaces are available. The number and capacity, in terms of memory throughput, of the PL-PS interfaces directly impact the performance and degree of temporal isolation that can be enforced among criticality domains. The FPGA can also provide different memory blocks, such as scratchpad (SPM) and PL-side DRAM. Examples of existing MPSoC platforms that fit into our system model are the Intel Stratix 10 SoC FPGA, Intel Arria 10 SoC FPGA, Intel Cyclone SoC FPGA, Xilinx Ultrascale+ ZCU102, and Xilinx Zynq-7000.

Communication Engine. We assume that a communication engine capable of accessing and transferring memory from/to PL and PS memories is available. Usually, a Direct Memory Access (DMA) component is available in either the PS or the PL and it can act as the communication engine. Its main role is to provide means for the load and unload phases of the three-phase task model. Differently from the previously implemented three-phase solution in [28], which used TDMA arbitration with fixed slot sizes, we propose a TDMA mechanism with finer granularity and per-core slots of different sizes. In this scheme, each real-time core j is assigned a slot size σ_j, with $\Sigma = \sum_{j=1}^{M} \sigma_j$ being the length of the TDMA round. We do not require the slots to be sized based on the SPM dimension; instead, if a DMA phase cannot finish within a slot, we break it down into multiple transfers and perform them over multiple TDMA rounds. The price we pay is extra overhead: since it takes some time to re-program the DMA controller, during each slot we can only perform DMA transfers for a maximum of $\bar{\sigma}_j$ time. Hence, $(\sigma_j - \bar{\sigma}_j)$ represents the DMA overhead. Assume that two consecutive unload/load phases (refer to Figure 1) require k TDMA slots. Then it is easy to see that the total transfer time Δ is upper bounded by:

$$\Delta = k \cdot \Sigma + \sigma_j; \qquad (1)$$

the core receives one slot every Σ time, but its initial slot can be wasted if the first memory phase arrives just after the beginning of the slot.

3.3 Application Model

Because multiple criticality domains exist in the system, we make different assumptions on applications in different domains. We make no assumption on the behavior of applications operating in low-criticality domains. They can perform complex I/O operations and they can be arbitrarily memory intensive.

Conversely, we assume that mid- and high-criticality applications adhere to more conservative assumptions. Mid-/high-criticality applications are structured as real-time tasks: a sequence of jobs whose activation is time- (periodic) or event-triggered (sporadic). Mid-/high-criticality applications are also statically assigned to cores, and locally scheduled using non-preemptive rate-monotonic scheduling (RM). Inter-task communication is performed via message passing. Only input data – from other tasks or devices – available by a given job's activation instant are used by the job itself. Similarly, output data are produced by a job only at its completion.

We assume that the memory footprint of mid-/high-criticality tasks is limited. On one hand, this allows to place code and data of real-time applications onto local memories of constrained size. On the other hand, it allows to load and unload applications in and out of local memories – following scheduling decisions – without incurring into high overheads. Tasks follow the three-phase model, as discussed in Section 2. Since we employ similar scheduling rules with variable time memory phases, we argue that the analysis in [33] can be adapted to provided scheduling guarantees for our proposed system, after using Equation 1 to bound the length of memory phases. Due to space limitations, we defer a complete schedulability evaluation to future work, while in this paper we focus on the hardware and software design of the computing platform.

4 Design Principles and Approach Overview

Our design revolves around the idea of *freedom from interference* among criticality domains [5]. The ideal software stack and assignment of resources to domains is depicted in Figure 3. We hereby provide a short description of the main challenges and techniques used to achieve a close approximation of what is depicted in Figure 3 by using a commercially available MPSoC embedded platform. We describe additional important implementation details in Section 6.

Inter-domain Interference. Temporal interference between criticality domains should be limited. More specifically, it is fundamental that any interference from a lower-criticality domain towards a higher-criticality domain is prevented – solid vertical lines in Figure 3. It is desirable that higher-criticality applications do not interfere with lower-criticality domains. But some degree of interference is acceptable in this case – dashed vertical lines in Figure 3. The paradigm follows traditional safety-critical systems certification guidelines [21]. High-criticality applications need to be certified regardless of the behavior of lower-criticality workload. Conversely, some degree of knowledge of higher-criticality applications can be assumed when certifying lower-criticality applications.

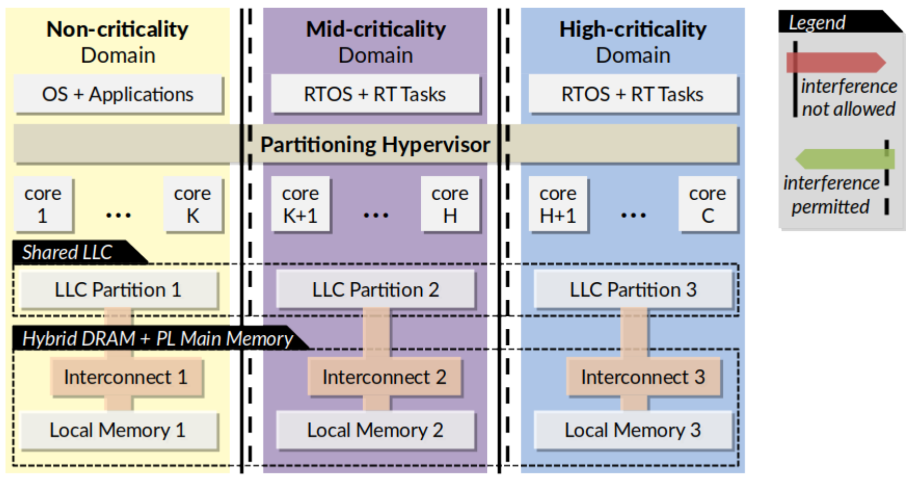


Figure 3 Ideal software and hardware stack organization.

Partitioning Hypervisor. Applications in different domains operate in self-contained address spaces, with inter-domain communication channels handled at the hypervisor level. Hardware resources (*e.g.*, cores, cache partitions, main memory storage, I/O devices) are statically assigned to criticality domains. As such, we employ a thin partitioning hypervisor which does not perform any online scheduling. The partitioning hypervisor has a number of roles, including (1) providing spatial isolation for RTOSes that do not support virtual memory; (2) partitioning cores to criticality domains; (3) enforcing LLC partitioning via memory coloring; (4) performing tasks' relocation to/from DRAM into local memories; and (5) providing message-passing channels for inter-domain communication.

LLC Partitioning. We rely on LLC partitioning based on page coloring[1] [10]. Hypervisor-level coloring has been proposed in [4, 16, 18]. An extensive discussion of the subtle practical challenges to enforce coloring at the hypervisor level is provided in [18]. In this work, we use the same hypervisor as in [18].

Preventing Memory Waste. A well known drawback of cache partitioning via coloring is memory waste. Coloring enforces a restriction on the physical addresses, and hence actual memory, that can be assigned to applications. For instance, if one wants to assign one-fourth of a shared LLC to a guest OS, then one-fourth of available main memory cannot be assigned to any other OS. This represents a significant drawback. The problem is even more severe when local memories like scratchpads are used. In fact, the size of scratchpads is typically very limited – a few hundreds of kilobytes to a few megabytes. Enforcing coloring essentially cripples the ability of applications to access the majority of an already limited memory resource. In this work, we leverage the Programmable Logic (PL) and propose a technique to prevent coloring-induced memory waste. Specifically, we introduce a *bus translator* that acts on transactions forwarded to local memories. In short, the component redirects colored – and hence scattered – memory accesses to contiguous memory locations.

[1] In this work we use the terms cache coloring and page coloring interchangeably.

Main Memory Partitioning. Partitioning main memory among guest OSes is necessary due to the problem of shared memory contention. Allowing the access to the same memory bank from cores allocated to different criticality domains would violate the requirement of enforcing strict isolation. Additionally, multiple domains can saturate the shared bus and/or memory controller, experiencing significant contention and delays. Both problems are well known. Solutions based on coloring have been proposed for the former [38, 18, 14]. Software [20, 1] and hardware [40] solutions based on bandwidth regulation have been explored to address the latter. In this work, we propose and explore an alternative approach to both issues. Our approach is made possible by the capability of defining new hardware components in the Programmable Logic (PL). First, we instantiate dual-ported memories that are only accessible by a single criticality domain. Next, we dedicate a PL-PS interface to criticality domains, and on each PL-PS interface we instantiate two memory controllers inside the PL. The first controller is used for memory accesses generated by applications running on the processor. Whereas, the second controller is used for memory transactions originated by the communication engine.

Handling Tasks' Relocation. As described in Section 3, tasks' code and data are moved to/from local memories defined in the PL by the communication engine. To implement task relocation (for loading/unloading), a possible approach consists in compiling applications using position-independent code (PIC) [28]. However, compilation as PIC results in less optimized binaries [28]. Additionally, migrating a running task to a different memory region is challenging [2]. In this work, we propose to compile tasks against absolute intermediate physical addresses (IPA). Then, after the communication engine has located a new task at a potentially new physical location in local memory, a hypervisor routine is invoked to map the new physical addresses (PAs) to the set of IPAs against which tasks have been compiled.

5 Design Space Exploration for Mixed-Criticality in a Modern MPSoC

In this section, we first describe the architectural overview of the considered platform. We then describe the experimental setup and different scenarios that were evaluated to justify our final design.

5.1 Architectural Overview of the Chosen Platform

For our implementation, we have used the Xilinx UltraScale+ ZCU102 MPSoC [34]. On this platform, the PS comprises two ARM Cortex-R5 cores, each having its own tightly coupled memory of 128 KB. There are also four application (ARM Cortex-A53) cores, each having its own local instruction and data cache (32 KB each). The Last-Level Cache (LLC) of 1 MB is shared by all application cores. There is no dedicated SPM provided for the application cores. This is in line with many high-performance embedded multicore processors. The PS includes a DDR4-2666 (main memory) controller with a data bus width of 64-bit, which on our reference board is connected to a 4GB DDR4 memory module. The PL also includes a separate, 16-bit synthesized memory controller, which on our board is wired to a 512 MB DDR4 memory module.

[2] This is because registers and stack in a saved context may contain absolute addresses.

Multiple interfaces between the PL and the processor subsystem (PS) exist. There are three interfaces going from the PS [3] to the PL. Out of the three, two are high performance master interfaces (HPM0 and HPM1) whereas the third interface is the low performance domain (LPD) interface. There are also interfaces from the PL to the PS, specifically the high-performance coherent (HPC) and high-performance (HP – non-coherent). Finally, there are 3 MB of block RAM (BRAM) inside the PL. For the rest of the paper we will use BRAM and SPM interchangeably.

5.2 Experiments

When exploring the characteristics of modern MPSoC platforms, it is easy to realize that there are many possible designs one can create to achieve predictability for mixed criticality domains. For the Xilinx ZCU102, for instance, the communication between the PL and the PS can have one or two high performance master ports (HPM0 and HPM1). Tasks running on the application cores (A53) can use the PS or PL DRAMs or even access the block RAM (BRAM) in the FPGA. We have designed a set of experiments to evaluate the behavior of different configurations under stress. Based on the related work [31, 28, 20], we chose two memory-intensive applications (disparity and mser) from the San Diego Visual Benchmark Suite (SD-VBS) [32] to be used in the evaluations. We chose the SD-VBS benchmark suite, because it provides vision applications similar to those used in autonomous cars. Thus, they represent real-time applications that demand both predictability and performance. We then ported `disparity` and `mser` to Erika RTOS/Jailhouse (we describe Erika and Jailhouse later in Section 6) and executed them with SQCIF (128×96) input data size. To stress the memory subsystem, we used a bandwidth benchmark (BW) from [12]. This benchmark is tailored to issue writes to the main memory (DRAM) or SPM (i.e., block RAM in the PL) by ensuring that every write is a miss in the LLC.

Using the memory intensive and bandwidth benchmarks, we evaluate the scenarios described in Table 1. We consider two legacy (Lcy) scenarios, and three scenarios in which our solution is used (Our). In the first legacy scenario (Lcy-Solo), the benchmark under analysis (`disparity` or `mser`) runs solo from the PS DRAM without cache coloring on top of Linux (kernel 4.14). Note that it does not use any high performance master (HPM) port, because it does not access the PL. In the second legacy scenario (Lcy-Stress), contention is added. Specifically, three bandwidth benchmark instances access the PS DRAM also from different cores in Linux. This scenario represents the simplest possible design in the platform since no special technique is used to avoid contention. Next, we consider our solution. In scenario Our-Solo, the benchmark under analysis runs alone in the system using a dedicated HPM port and accessing an SPM in FPGA. In this and the following cases, the cache has also been partitioned via coloring. In scenario Our-Mid, the benchmark under analysis runs from the mid-criticality domain and three contending bandwidth benchmark instances are added. The first runs in the low-criticality domain (Linux), the second in the high-criticality domain, and the third in the mid-criticality domain. The latter shares an HPM port with the benchmark under analysis. Finally, in scenario Our-High, the benchmark under analysis executes from a high-criticality domain, using a dedicated HPM port. Two contending bandwidth benchmark instances run from a mid-criticality domain and share a single HPM port, while an additional contending bandwidth benchmark instance runs in the low-criticality domain.

[3] Here the direction of the interface indicates which side of the system can initiate transactions towards the other side. On an interface from PS to PL, the PS is the master of the interface, while the PL is the slave.

Table 1 Summary of the five scenarios considered for the evaluation.

Scenario	Experiment	Accessed Memory	Coloring	HPM Port	Contention Type
Lcy-Solo	Solo	PS DRAM	No	Not used	None
Lcy-Stress	Contention	PS DRAM	No	Not used	3× BW
Our-Solo	Solo	SPM	Yes	Dedicated	None
Our-Mid	Contention	SPM	Yes	Shared	1× BW from low-crit. 1× BW from mid-crit. 1× BW from high-crit.
Our-High	Contention	SPM	Yes	Dedicated	1× BW from low-crit. 2× BW from mid-crit.

Figure 4 and Figure 5 depict the results for the `mser` and `disparity` applications, respectively, under the five aforementioned scenarios. Each experiment reports the result of 1000 executions. In both figures, the x-axis reports the execution time in clock cycles[4]. On the left y-axis we present the experimentally derived execution time distribution, while on the right y-axis we show the cumulative distribution function (CDF). The vertical dashed line shows the average, while the vertical dotted line corresponds to the observed WCET. The annotation in each plot provides the numerical values for average, WCET, best-case execution time (BCET), and variability window – which is a metric of predictability and is computed as $(WCET - BCET)/WCET$.

A few important trends can be highlighted in the results for `mser` (Figure 4). First, for the two legacy scenarios, in Lcy-Stress (Figure 4b) the application exhibits a drastic 1.73× increase in WCET compared to Lcy-Solo (Figure 4a) due to added contention. Moreover, the execution time in the Lcy-Stress case becomes unstable, with a variability window of 27.8%. Next, when executing in a mid-criticality (or high-criticality) domain without contention (Our-Solo case – Figure 4c), the performance of the application under analysis is comparable to the Lcy-Solo case. If the application is deployed in a mid-criticality domain (Our-Mid case – Figure 4d), a sharp improvement in predictability and WCET is observed compared to the Lcy-Stress case. In fact, the variability window is reduced by 42% and the WCET is reduced by 31%. Finally, in Figure 4e, the application is run inside a high-criticality domain, and hence with a dedicated HPM port – Our-High case. By considering the Lcy-Stress case as the baseline, we observe a 58% reduction in variability window, as well as a 37% reduction in WCET. Additionally, note that in the Our-High case the application performance is remarkably close to what is observed in the Our-Solo case.

The results for `disparity` reported in Figure 5 follow similar trends. First, the WCET shows a 1.27× increase between Lcy-Solo and Lcy-Stress, reported in Figure 5a and Figure 5b respectively. When the benchmark is executed alone in the system in a mid-criticality (or high-criticality) domain (case Our-Solo in Figure 5c), its WCET and average execution time increases only slightly by 1.04× and 1.07× respectively. Intuitively, this is because the SPM is a slower memory compared to the PS DRAM. Next, consider the Our-Mid (Figure 5d) case where `disparity` is executed in a mid-criticality domain with contention from the rest of the system. Compared to the Lcy-Stress case, we observe a 8% reduction in WCET and a 32% decrease in variability. When the application runs in a high-criticality domain (case Our-High in Figure 5e), its WCET is minimally affected by contending workload, with a 1.06× increase compared to the Our-Solo case. Notably, the variability window in the Our-High case is lower than in the Lcy-Solo case.

[4] 1 clock cycle is equal to 0.01 us.

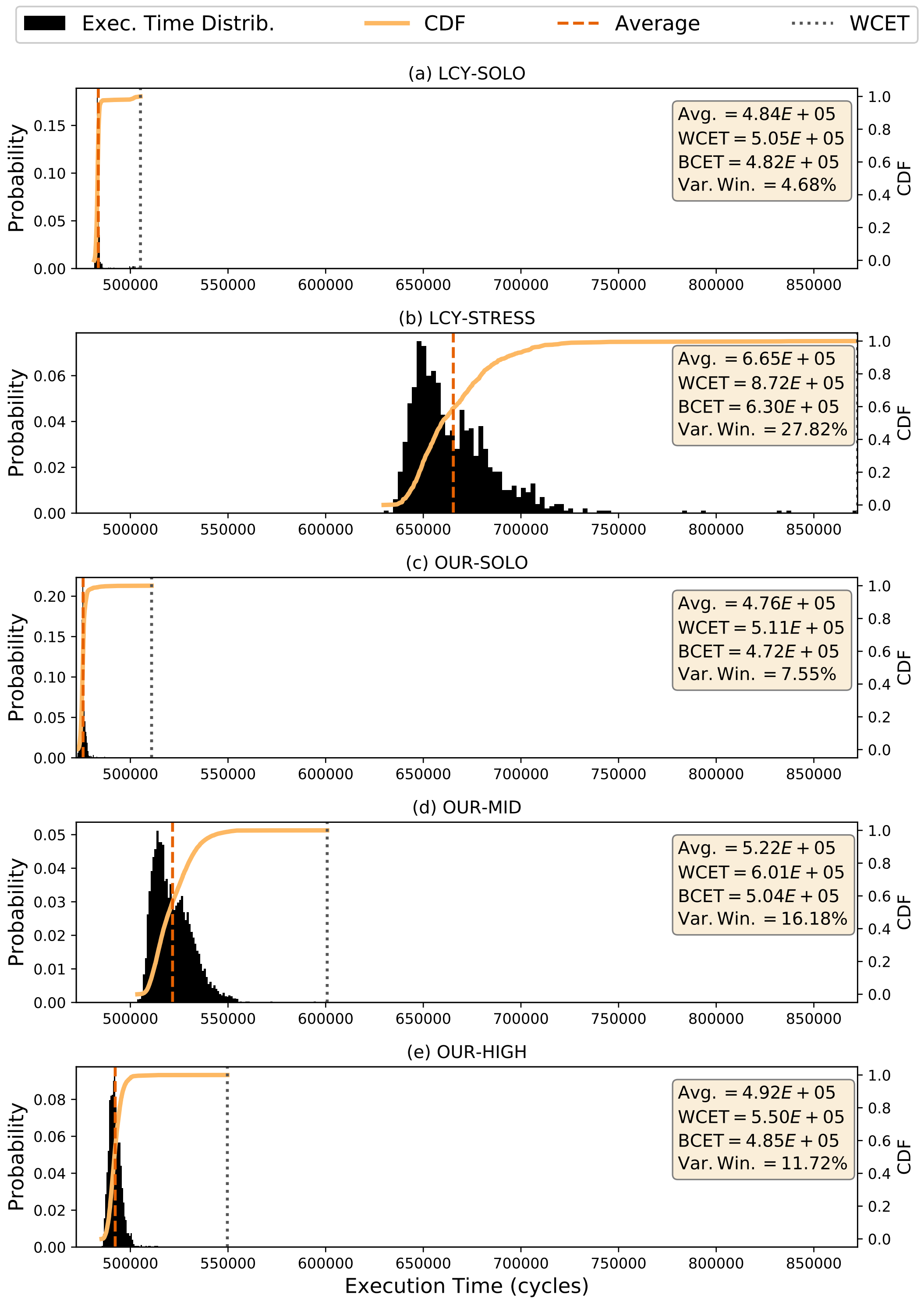


Figure 4 Results for `mser` application. See the summary of the scenarios in Table 1.

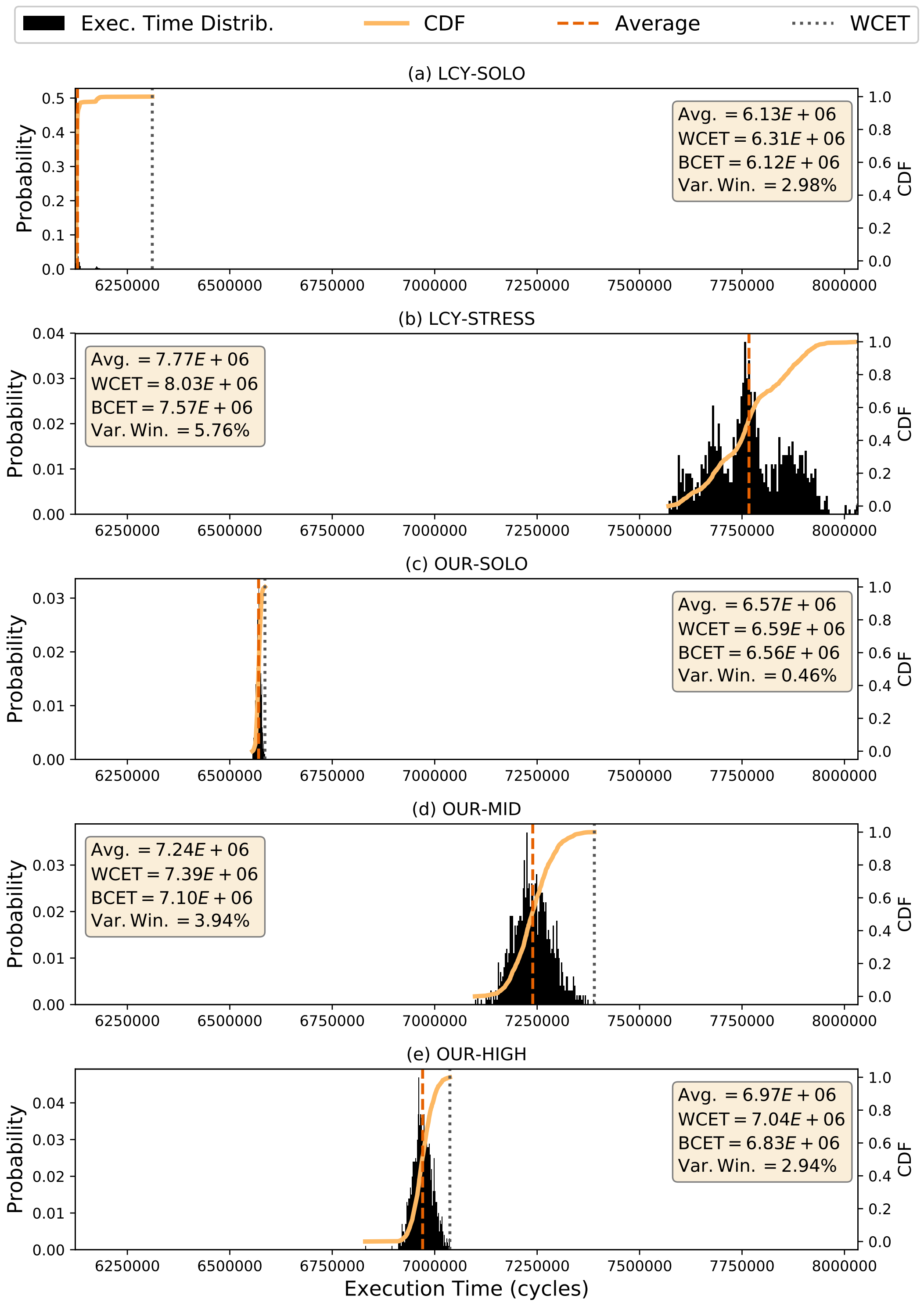


Figure 5 Results for `disparity` application. See the summary of the scenarios in Table 1.

Based on the evaluated scenarios, we can conclude that the hardware isolation provided by a dedicated memory interface, a dedicated SPM memory, together with cache coloring (Our-High case), is able to deliver better predictability and performance close to the ideal case (Our-Solo). We present and discuss the final hardware design in the next section.

6 Support for Mixed-Criticality Applications on MPSoCs

In this section we present a general overview of the implementation carried out in the ZCU102 platform to provide predictability for mixed-criticality applications. We start giving an overview of the implementation in Section 6.1, then we present the details for each implemented component (hypervisor, cache coloring, address translator, RTOS, and code/data relocation).

6.1 Overview of Implementation

Based on the experiments described in the previous section, our final hardware design is depicted in Figure 6. We assign one of the A53 cores to be a low-criticality core, two of them to be mid-criticality cores, and one of them to be a high-criticality core. The mid- and high-criticality cores run their own Real-Time Operating System (RTOS). A few noticeable features of our proposed design are: (i) the low-criticality domain is assigned direct access to DRAM because this domain features applications with sizable footprints; (ii) each mid- and high-criticality domain is assigned a private SPM; (iii) each of these SPMs is dual-ported and a controller is instantiated on each port to prevent contention between DMA and core at the SPM controller; and (iv) the high-criticality domain also occupies a dedicated PS-PL interface to access its private SPM. It should be noted that the SPM (BRAM) memories in the Xilinx FPGA are dual-ported and thus there is no extra overhead (which may not be the case when using dual-ported memories in Application Specific Integrated Circuits). Moreover, the size of each SPM can be defined according to the applications and RTOS requirements for each criticality domain. Since in our platform the maximum size of all SPMs is 3 MB, the size of the SPM used by the high-criticality domain was set to 2 MB, while the size of the other two SPMs used by mid-criticality domains was set to 512 KB each.

The low-criticality core is also responsible for booting a hypervisor (Jailhouse). Jailhouse allows us to partition shared memory resources, especially the LLC and DRAM by implementing cache coloring. We have two partitions in the DRAM; one dedicated to run Linux and another one to place the code/data of the tasks running on the A53 application cores (to support the three-phase model as will be discussed below).

We propose creating separate SPM in the PL for all the mid- and high-criticality cores. Thus, a dedicated or fast interface such that each core can access its own SPM without seeing a delay from another core is required. Unfortunately, there are only two high performance interfaces between PL and PS available in the platform and three A53 application cores. Therefore, in our design we assign one shared high performance interface to two A53 cores while the third core has a dedicated interface to its own SPM memory (see Figure 6).

Although there is another interface between PS and PL called low performance domain (LPD) that can be used for the third A53 core, we opt not to use it. We have used a latency benchmark [12] to measure the performance when one single core is accessing the LPD interface and when two cores access the same HPM interface under stress. The obtained latency for the HPM interface under stress was 202 ns, while for the LPD was 220 ns. Thus, the LPD interface is used to carry DMA transfers to/from the SPM/DRAM on the behalf of the A53 application cores, as part of the TDMA-based scheduling. The TDMA-based

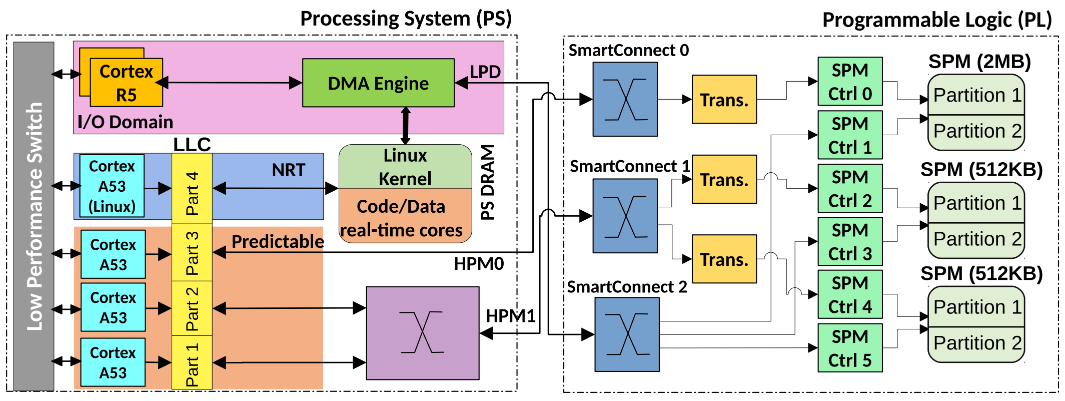


Figure 6 PS-PL interface and design.

scheduling of the DMA is handled by the R5 core. To pipeline the execution of a currently running task with the load of the next task, we divide the SPM into two halves. A dual ported SPM was used so that a DMA and an application core can both write/read to/from SPM at the same time.

In order to avoid the contention between A53 cores in different criticality domains, we partition the LLC via coloring. Coloring is used since no hardware support is available to partition the LLC. The use of coloring generally results in portions of physical memory being unusable to applications. This is generally acceptable for main memory, because its size is not constrained (few GBs). Conversely, SPMs in the PL are usually limited in size (few KBs or MBs). For instance, if coloring is used to define four equally sized LLC partitions, this would reduce the size of each SPM to 1/4. To avoid this side effect of coloring, we introduce an address translator between the A53 and the SPM. Since the cache is physically indexed, coloring both the PS DRAM and SPM is required to avoid interference (otherwise there would be a cache interference at every SPM access).

In the following subsections, we provide a detailed discussion on each of the main components including Jailhouse, page coloring, address translator, how the A53 cores in different criticality domains communicate using the hypervisor, RTOS support for the system model, and task relocation to support the three-phase model.

6.2 Jailhouse to Partition the Shared Resources

As the hypervisor we use Jailhouse. Jailhouse is a partitioning hypervisor which can be used to transform a symmetric multiprocessing (SMP) system into an asymmetric multiprocessing (AMP) system [25]. Jailhouse is bootstrapped via a Linux driver and favors simplicity and low-overhead over sophisticated (para-)virtualized techniques, which is ideal for real-time systems [25]. It requires at least one core to be assigned to Linux – the root cell. Once the driver is loaded, it takes control of the entire hardware and reassigns a partitioned view of the hardware resources back to Linux, based on a configuration file. Then, to create additional domains (called non-root cells), Jailhouse removes hardware resources assigned to Linux (such as a processor core or a specific I/O device) and reassigns them to the new cell [25]. The idea is to have non-critical tasks running on the Linux cell and critical tasks running on isolated partitions on top of an RTOS.

The A53 cores support a two-stage virtual memory translation. User-space applications in a guest-OS, such as Linux or an RTOS, are assigned virtual addresses (VA). The first stage of translation uses page tables maintained by the guest OS to translate VAs into intermediate physical addresses (IPA). The second stage of translation is in control of the hypervisor, and it is used to translate IPAs to physical addresses (PAs) via a second set of page tables.

The RTOS used for mid-/high-criticality domains is Erika Enterprise version 3, which is open-source and OSEK/VDX certified [9, 7]. Erika supports fixed-priority scheduling and a porting for Xilinx Ultrascale+ platform is available.

6.3 Page Coloring

To enforce strong inter-domain (inter-cell) and hence inter-core performance isolation, we leverage page coloring (see [18, 10] for a complete overview of the technique). We use the virtualization extensions of the processor to implement coloring by enforcing appropriate restrictions on the color of pages that Jailhouse maps to IPAs of virtualized cells. Specifically, we impose that physical pages with non-overlapping colors are assigned to cells activated on different cores. The advantage of this approach is twofold. On the one hand, it allows us to localize the changes required to implement coloring-based partitioning in a single software component (Jailhouse). On the other hand, it allows deploying unmodified and possibly closed-source OS inside our criticality domains. A similar technique was used in [22, 16, 18].

6.4 Address Translator to Overcome Limitations of Cache Coloring

To overcome the problem of memory waste imposed by coloring, we designed an address translation hardware IP. The component performs physical address translation for memory transactions originating from the PS towards the PL. To better understand how the component operates, let us consider our specific setup. To access an SPM with a size of 2 MB, 21 bits of the address are provided for requests originated from the PS. With cache coloring enabled (and four colors, one for each core), only one in four memory pages can be used, with addresses aligned at 16 KB boundaries (each page has a size of 4 KB). The adopted solution is the following. Instead of receiving 21 bits of an address, the translator IP receives 23 bits (8 MB) from the PS, removes the specific color bits from that, and passes it to the SPM controller.

Given the geometry of the LLC (1 MB, 16 ways) the color bits that can be used to perform partitioning are bits 12 to 15 of each physical address. To create 4 partitions, one could use bits 12 and 13. Pages with bits [12, 13] = 0b0 would be assigned to partition 1; pages with bits [12, 13] = 0b1 to partition 2; and so on. In this way, four sequential physical pages will be assigned to four different partitions. This is not ideal, however, because the L1-Data cache in this platform is *Physically Indexed, Physically Tagged* (PIPT) and fits 2 pages per way. If a CPU is only given access to one every four pages, only half of the L1-D cache will be utilized. To avoid this problem, we use bits 14 and 15 as the LLC color bits. In this configuration, each partition is given 4 consecutive pages.

Let us assume that the address of the translator in Figure 6 responds under the address range `0xA0000000` to `0xA07FFFFF` (8 MB). Following the discussion above, bits 14 and 15 are used as LLC coloring bits. Figure 7 shows an example where a request address of `0xA0023456` (offset `0x023456`) from a core arrives to the translator IP. Bits 14 and 15 of the offset are dropped by the translator and the resulting offset is `0x0B456` in a 2 MB non-colored space.

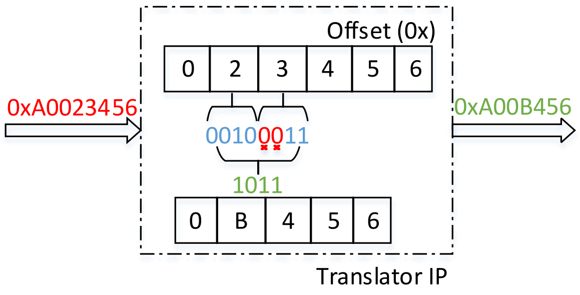


Figure 7 Translator IP operation. The two most significant bits from the fourth byte (in red) of the input address are dropped..

In our design (Figure 6), there are three translators to handle the requests coming from each core. With this mapping mechanism, the SPM capacity is not affected by the cache coloring (we do not lose space) and since the translator IP is burst-capable, we do not lose bandwidth nor increase latency in accessing the SPMs. Besides that, the area overhead of the module in terms of the numbers of Flip-Flops (FF) and Lookup tables (LUTs) compared with the design without any translation IP are 0.57% and 0.41% respectively, while the block RAM cell count remains the same.

6.5 Communication among Jailhouse Cells

Usually, communication among processors is done by sharing memory buffers and Inter-Processor Interrupts (IPIs). Jailhouse allows to issue direct IPIs only among processors that are assigned to the same cell, and hence the same criticality level. To avoid a low-criticality domain (*i.e.,* Linux) to interfere with higher-criticality domains (by sending an unlimited number of IPIs), currently we do not allow issuing IPIs among processors with different criticality levels. In the future, the communication among tasks from different criticality levels will be performed through the Jailhouse shared memory mechanism, based on the creation of virtual PCI devices and their legacy interrupts in the static configuration for each cell [25]. This means that the system designer could specify which criticality level (*i.e.,* Jailhouse cell) can communicate with other criticality levels. As future work, we will investigate how to enhance Jailhouse with a server-based scheduling mechanism for IPIs [8], such that real-time guarantees can be preserved in the event that a low-criticality domain tries to send an unlimited number of IPIs to a higher-criticality one. Also, we propose the use of FIFO buffers implemented in the shared memory (which can also be colored) to support the communication among OSes.

6.6 Erika RTOS Running on Real-Time Cores

All the A53 application cores run a partitioned fixed priority scheduler, provided by the Erika RTOS, and always execute from dedicated SPM memory assigned to them. Both the dedicated SPMs and the PS DRAM assigned to Linux are colored to avoid cache evictions in the shared cache. Erika does not support virtualization on ARMv8 CPUs, as such it would not use virtual addresses (VAs). By default, however, Jailhouse performs the setup of a flat 1:1 stage-one (virtual address to intermediate physical address – VA$\rightarrow$IPA) addressing space before booting any non-root cell. This is required to support cacheable memory. An application in the Erika RTOS is always statically compiled against VA/IPA addresses. As shown in Figure 8, the task running on the Erika core can be in one of the following states:

- **Running:** The tasks is executing from SPM.
- **Ready Loaded:** The task is loaded and is ready to execute from SPM.
- **Ready Unloaded:** The task is released but it is not yet loaded to SPM.
- **Completed**: A task has completed.
- **Waiting on Event (Unloaded)**: The task is waiting on a timer or on an event.

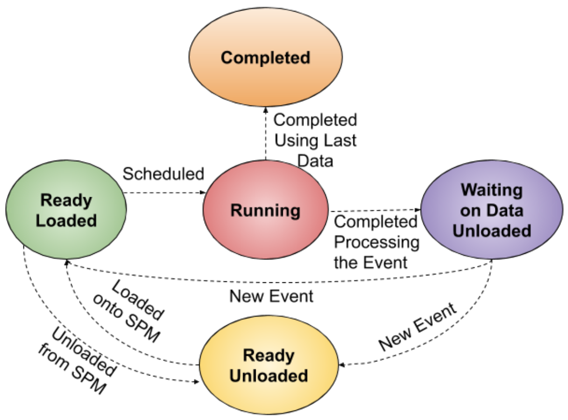


Figure 8 Overview of the different states of a task in Erika RTOS.

Note that the transition from Waiting on Event Unloaded to Ready Loaded is performed when there is no other ready task and the waiting task that was unloaded receives all the events it needs. The state transition also encompasses the loading phase. To allow the load and unload of code and data of Erika's tasks, we use the support for virtual memory implemented in Jailhouse, described in the next subsection.

6.7 Code/Data Relocation

Relocation is the process of assigning addresses to position-independent code and data. We use code/data relocation to support the loading and unloading of Erika tasks' code and data. Relocation is initiated by the Erika RTOS when its scheduler decides to load or unload a task as required. Recall, however, that applications in Erika are statically compiled against a set of virtual addresses (or intermediate physical addresses, since Erika does not support virtual memory). As such, relocation is performed by modifying the mapping from intermediate physical addresses to physical addresses (IPA→PA) managed by Jailhouse.

Erika first informs Jailhouse that a relocation must be performed. This is done via a hypercall (*i.e.,* `hvc` assembly instruction), which was added to Erika. Hypercalls in Jailhouse are services provided by the hypervisor to its cells. A Jailhouse hypercall receives three arguments; the hypercall code or ID and two arguments that are specific to the hypercall. We added to Jailhouse two new hypercall IDs, indicating either load or unload operations. The second argument is used to encode (i) the source/destination address in DRAM (page-aligned, least-significant 12 bits are zero); and (ii) the offset in pages from the beginning of the SPM where the task needs to be loaded to/unloaded from (the largest SPM is 2 MB, so the maximum offset is 512 - 1 pages, and it takes the 9 least-significant bits). The third argument encodes the size of the task that needs to be loaded/unloaded. As shown in [3], the overhead of a hypercall in Jailhouse on the ZCU102 platform is around 400 ns.

Once Jailhouse receives a request to relocate a task's code/data, it performs the following steps. First, it determines the absolute source (resp., destination) in DRAM and destination (resp., source) in SPM for a load (resp., unload) operation. Next, it modifies the IPA→PA mapping so that the range of intermediate physical addresses starting at the provided source address (resp., destination), and spanning for the number of pages specified by the size parameter, map to the destination address. After the remapping is completed, Jailhouse returns control to the calling environment (Erika RTOS). The effective copy of the task into/from SPM is performed by the DMA engine.

7 Evaluation

In this section, we present the evaluation of our system design. We start showing an evaluation of the DMA performance, including the time to transfer different data sizes from PS DRAM to the SPM and its programming overhead. We then demonstrate the limits in terms of hard real-time guarantees of our system through a case study on image processing.

7.1 DMA Evaluation

In order to move data between the PS DRAM and the SPM memory inside the PL, we use the PS-side DMA. Because only one DMA is used to move data on behalf of three A53 cores, we propose a fine granularity TDMA-based scheduling of the DMA. When activated, the DMA transfers data between PS DRAM and SPMs using the low-power domain (LPD) interface. In our design, a dedicated LPD port is used instead of a shared HPM port to avoid contending with the application cores when performing DMA transfers. The TDMA schedule is handled by one of the ARM Cortex-R5 cores. For this purpose, a bare-metal firmware was deployed on the R5, created using the Xilinx SDK 2018/02. The SDK uses the `armr5-none-eabi-gcc` compiler. The following compilation flags were used: `-DARMR5 -W -Wall -O0 -g3`.

We measured the DMA transfer time for different data sizes, extracting the average transfer time, standard deviation (STD), and the worst-case transfer time among 1000 samples. Table 2 shows the obtained results. Recall that 1 MB represents half the size of the largest SPM in our design. The results show that the standard deviation remains within the range [0.057, 0.1]. It can also be noted that the achievable bandwidth increases proportionally to the amount of contiguous memory transferred, peaking at around 870 MB/s with transfers of 1 MB in size.

We also measured the DMA programming overhead (*i.e.,* the time to program and start a DMA transfer). The worst-case DMA programming overhead obtained from all the experiments was 3.89 μs. For small data sizes (2 and 4 KB for instance), the relation between the programming overhead and the transfer time is significant. In this case, it may be beneficial to avoid small data transfer whenever possible or use the own task's core instead of the DMA. We plan to fully analyze the impact of the DMA programming overhead into the schedulability of real-time tasks in future work. However, based on insights provided by previous work on the three-phase model [28, 33], we would like to point out that the model behaves well as long as task execution times are longer than the time required to reload an SPM partition. As an example, if we consider a partition of 256 KB (half the size of a 512 KB scratchpad), and a TDMA slot with transfer size of 32 KB for each core, then based on Equation 1 we obtain $\sigma_j = 38.81 + 3.89 = 42.7$ us, $\Sigma = 3 \cdot 42.7 = 128.1$ us, and $k = 2 \cdot 256/32 = 16$ as the number of slots required to unload/load the partition. This results in a memory reload time $\Delta = 2092.3$ us, meaning that tasks should execute for at least 2 ms to hide the memory time.

Table 2 DMA transfer time (in us) and bandwidth for different data sizes.

Transfer Size	Transfer Time			Bandwidth (MB/s)
	Average (μs)	STD	Worst-case (μs)	
2 KB	4.92	0.057	5.11	397.0
4 KB	7.15	0.04	7.27	546.3
8 KB	11.63	0.01	12.01	671.8
9.1 KB	12.91	0.05	13.11	688.4
16 KB	20.62	0.08	20.96	757.8
22 KB	27.42	0.10	27.72	783.5
32 KB	38.52	0.05	38.81	811.3
1 MB	1149.44	0.05	1149.78	870.0

7.2 Case Study: Image Processing

To evaluate our system design we consider a system where video frames captured from a camera are processed in a high-criticality domain. Video frames are processed using the `disparity` benchmark from the SD-VBS suite [32]. `Disparity` computes depth information for objects represented in two input images, obtaining relative positions of objects in the scene. This kind of algorithm is useful in applications such as cruise control, pedestrian tracking, and collision control [32]. The objective of this evaluation is to demonstrate how the proposed system behaves in a realistic setup and to show its limits in terms of achievable hard real-time guarantees.

To this end, the `disparity` benchmark is executed as a periodic task. During each activation, it computes the `disparity` of two input images. At every new period, `disparity` reuses one image from the previous iteration and uses a new image transferred by the communication engine. We performed two experiments with two different image resolutions, *i.e.,* 64x48 and 128x96 (SQCIF). We only used these image resolutions due to limitations in the size of the SPM. Also, `disparity` requires input images to be in the bitmap image file (BMP) format, which is uncompressed. Thus, for a resolution of 64x48, an image has a size of around 9.1 KB, while for 128x96 an image has a size of 22 KB. We use a set of 20 images of a scene from the KITTI vision benchmark suite dataset [27]. In particular, we used 20 frames from the 2015 stereo multiview dataset. The original images had a resolution of 1241x376. We converted the frames to the lower resolutions described above. We move the I/O data of the tasks from/to DRAM to/from the SPM at load/unload phase of the task using the same approach as described in [29]. Table 2 shows the DMA transfer time for both image resolutions (9.1 KB and 22 KB). Erika RTOS consumes 294 KB of memory (including data and code) and it is fixed on the SPM (we do not load nor unload code/data of the RTOS). `Disparity` using image resolution of 64x48 consumes 349 KB, while for 128x96 it consumes 745 KB, also including data and code. Although not required in this case study, note that when input data is too large to fit into the SPM, it is possible to use compiler-level techniques to break the load/unload phases into small chunks [26].

We considered four out of the five scenarios described in Table 1. We run `disparity` alone in the system from the PS DRAM on top of Linux (LCY-SOLO), next `disparity` runs from the PS DRAM with three bandwidth (BW) benchmark instances (see Section 5) also executing and accessing the PS DRAM (LCY-STRESS). The `disparity` benchmark is then executed from SPM on top of Erika/Jailhouse with coloring and using our hardware design without (OUR-SOLO) or with (OUR-STRESS) interference from the rest of the system. Ideally, when `disparity` runs with contention from the SPM (OUR-STRESS) it should exhibit comparable performance with respect to the case when `disparity` runs without interference from the SPM (Our-Solo). The case when `disparity` runs solo from PS DRAM (Lcy-Solo) serves as a baseline, while the case when it runs from PS DRAM under contention (Lcy-Stress) provides an idea of what we gain in terms of isolation and performance thanks to the proposed set of software/hardware techniques. Periodic execution of the `disparity` task was achieved under Linux by using a `CLOCK_REALTIME` timer to invoke a handler at the desired frequency. The handler then releases the `disparity` thread using a semaphore. The `disparity` benchmark, Erika OS, and the BW benchmark instances were compiled using `gcc` version 5.4 for the ARM64 architecture with the `-O2` flag.

First, we present the execution time of `disparity` in each of the four cases using an image resolution of 64x48 in Table 3, and a resolution of 128x96 in Table 4. We measured the execution time of 1000 individual processing jobs and extracted the average execution time, standard deviation (STD), BCET, WCET, and variability window. The variability window is calculated as $(WCET_{stress} - BCET_{solo})/WCET_{stress}$. Time measurements were taken using the processor cycle counter and converted to ms. Note that when working at 64x48 resolution, the two input images (9 KB each) fit into the L1 cache (32 KB). Thus, the observed worst-case when `disparity` is running alone is similar for both memories (PS DRAM and SPM). However, when contention is introduced, the benchmark suffers visible interference in the Lcy-Stress setup. Note that there is still some contention when `disparity` uses the dedicated HPM interface and cache coloring in the Our-Stress setup. This may be due to contention over Miss Status Holding Registers (MSHRs) in the last level cache [30]. The results for 128x96 (SQCIF size) input images were presented in Section 5 and correspond to the cases analyzed in Figure 5(a), 5(b), 5(c), and 5(e).

Table 3 Average, standard deviation, BCET, and WCET obtained from 1000 executions for the considered four cases with input image size of 64x48. All values in ms. Highlighted values in bold are used to calculate the variability window.

	Lcy-Solo	Lcy-Stress	Our-Solo	Our-High
Average	15.89	17.86	15.94	16.49
STD	0.01	0.07	0.01	0.06
BCET	**15.88**	17.69	**15.92**	16.34
WCET	16.00	**18.18**	15.96	**16.73**
Var. Window	12.6%		4.8%	

Table 4 Average, standard deviation, BCET, and WCET obtained from 1000 executions for the considered four cases with input image size of 128x96. All values in ms. Highlighted values in bold are used to calculate the variability window.

	Lcy-Solo	Lcy-Stress	Our-Solo	Our-High
Average	61.50	75.09	66.04	69.80
STD	0.02	0.34	0.07	0.26
BCET	**61.45**	74.32	**65.79**	69.04
WCET	61.80	**77.09**	66.30	**70.59**
Var. Window	20.2%		6.8%	

Based on the observed WCET in the various experiments, we vary the image processing task period and study when `disparity` starts missing deadlines in each case. Table 5 presents the obtained results for image size of 64x48. We vary the frequency from 55 Hz (18.18 ms period) to 63 Hz (15.87 ms period). A tick mark in the table indicates that the desired image processing rate was sustainable. In other words, that no instance of `disparity` missed its relative deadline (equal to the period). Whereas a cross mark indicates that the desired rate was not sustainable. From the results in Table 5, we can see that by running `disparity` without any interference, the maximum sustainable rate is 62 Hz. However, when running under contention and no isolation enforcement (Lcy-Stress case) the sustainable image processing rate drops to 55 Hz. Conversely, a rate of 59 Hz is sustainable if `disparity` executes from within a high-criticality domain defined using the proposed software/hardware techniques. Note that in this setup each image processing job has to wait for an image to be transferred in input by the DMA before it can start execution. Because DMA accesses to DRAM can experience contention, a decrease in sustainable rate is visible between the Lcy-Solo and the Lcy-Stress cases. Nonetheless, this experiment shows that our design provides better predictability and enables higher processing rates when the system is under heavy load.

Table 5 Supported frequencies for image size of 64x48.

Freq. (Hz)	Period (ms)	Lcy-Solo	Lcy-Stress	Our-Solo	Our-High
55	**18.18**	✓	✓	✓	✓
56	**17.86**	✓	✗	✓	✓
57	**17.54**	✓	✗	✓	✓
58	**17.24**	✓	✗	✓	✓
59	**16.95**	✓	✗	✓	✓
60	**16.67**	✓	✗	✓	✗
62	**16.13**	✓	✗	✓	✗
63	**15.87**	✗	✗	✗	✗

Table 6 shows results for input images with resolution 128x96 when running the `disparity` benchmark. The average execution time for `disparity` with image resolution of 128x96 when running solo from PS DRAM is 61.5 ms – see Table 4, Lcy-Solo case. Thus, we vary the frequency from 10 Hz until 17 Hz and observe that the image processing task starts missing deadlines when activated at 17 Hz. With 128x96 input images, the `disparity` benchmark under contention can sustain a rate of 14 Hz in spite of heavy system load when isolated in a high-criticality container (Our-High case). Conversely, the sustainable rate decreases to 12 Hz when no isolation is enforced. In the Our-Solo case, `disparity` can run at a maximum frequency of 15 Hz, which is slightly lower than what can be achieved in the Lcy-Solo case (16 Hz). The drop arises from the fact that the SPM memory in PL is a bit slower than the PS DRAM [34]. We did not see the same behavior for an image resolution of 64x48 due to the cache. Importantly, however, the sustainable rate in the Our-Solo case is very close to the Our-Stress case. Thus, it can be concluded that our software/hardware co-design is able to deliver performance to highly critical applications that are close to the best-case. It is also important to highlight the low performance achieved by disparity for higher resolution images. We plan to investigate how to achieve better processing rates for image applications on top of the platform in future work.

8 Conclusion and Future Work

In this paper, we have shown how one can define multiple criticality domains by exploiting the rich hardware features provided by a modern heterogeneous SoC that incorporates multiple CPUs and PL. Following the proposed design, the PL is used to define dedicated portions of scratchpad memory for mid-/high-criticality applications. Additionally, we ensure that

Table 6 Supported frequencies for image size of 128x96.

Freq. (Hz)	Period (ms)	Lcy-Solo	Lcy-Stress	Our-Solo	Our-High
10	**100.00**	✓	✓	✓	✓
11	**90.91**	✓	✓	✓	✓
12	**83.33**	✓	✓	✓	✓
13	**76.92**	✓	✗	✓	✓
14	**71.43**	✓	✗	✓	✓
15	**66.67**	✓	✗	✓	✗
16	**62.50**	✓	✗	✗	✗
17	**58.82**	✗	✗	✗	✗

no contention exists for high-criticality applications by routing their memory transactions using a dedicated high-performance memory interface inside the PL. Similarly, mid-criticality applications access their SPM using a memory interface shared only with other mid-criticality applications. External I/O and communication data from the rest of the system is transferred to the mid-/high-criticality domains by a TDMA-scheduled DMA engine using a real-time R5 core. We described our full-stack implementation of the proposed techniques and evaluated the system using realistic SD-VBS benchmarks.

As a part of future work, we plan to investigate how to enhance Jailhouse with a server-based scheduling mechanism for limiting the number of IPIs between processors from different criticality levels, how to integrate compiler code generation to automatically generate load/unload requests for tasks following the three-phase model, to discuss the schedulability analysis of the proposed fine-granularity DMA transfer based on TDMA, how to guarantee that the communication engine executes safely, since if it fails (due to a security attack for instance) the entire system also fails, and how to achieve higher computational power for image processing applications without giving up on the relatively strong isolation achieved by the current design.

References

1 A. Agrawal, R. Mancuso, R. Pellizzoni, and G. Fohler. Analysis of Dynamic Memory Bandwidth Regulation in Multi-core Real-Time Systems. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 230–241, December 2018. `doi:10.1109/RTSS.2018.00040`.

2 Ali Awan, Konstantinos Bletsas, Pedro F. Souto, Benny Akesson, and Eduardo Tovar. Mixed-Criticality Scheduling with Dynamic Memory Bandwidth Regulation. In *IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 111–117, August 2018.

3 Maxim Baryshnikov. FPGA-based support for predictable execution model in multi-core CPU. Master's thesis, Czech Technical University in Prague, Prague, Czech Republic, May 2018.

4 Alessandro Biondi, Mauro Marinoni, Giorgio Buttazzo, Claudio Scordino, and Paolo Gai. Challenges in Virtualizing Safety-Critical Cyber-Physical Systems. In *Proceedings of Embedded World Conference 2018*, pages 1–5, February 2018.

5 Alan Burns and Robert I. Davis. A Survey of Research into Mixed Criticality Systems. *ACM Comput. Surv.*, 50(6):82:1–82:37, November 2017. `doi:10.1145/3131347`.

6 A. Crespo, P. Balbastre, J. Simó, J. Coronel, D. Gracia Pérez, and P. Bonnot. Hypervisor-Based Multicore Feedback Control of Mixed-Criticality Systems. *IEEE Access*, 6:50627–50640, 2018.

7 Evidence. Erika Enterprise RTOS v3, October 2018. Online; accessed 16 October 2018. URL: `http://www.erika-enterprise.com/`.

8 T. Facchinetti, G. Buttazzo, M. Marinoni, and G. Guidi. Non-preemptive interrupt scheduling for safe reuse of legacy drivers in real-time systems. In *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 98–105, July 2005. `doi:10.1109/ECRTS.2005.21`.

9 Paolo Gai, Enrico Bini, Giuseppe Lipari, Marco Di Natale, and Luca Abeni. Architecture For A Portable Open Source Real Time Kernel Environment. In *In Proceedings of the Second Real-Time Linux Workshop and Hand's on Real-Time Linux Tutorial*, 2000.

10 G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni. A Survey on Cache Management Mechanisms for Real-Time Embedded Systems. *ACM Comput. Surv.*, 48(2), November 2015.

11 G. Gracioli and A. A. Fröhlich. Two-phase colour-aware multicore real-time scheduler. *IET Computers Digital Techniques*, 11(4):133–139, 2017.

12 Heechul Yun. Latency and Bandwidth Utilities. https://github.com/heechul/misc, February 2019.

13 C. Kenna, J. Herman, B. Ward, and J. H. Anderson. Making shared caches more predictable on multicore platforms. In *ECRTS '13*, pages 157–167, 2013.

14 H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154, April 2014.

15 Hyoseung Kim, Arvind Kandhalu, and Ragunathan Rajkumar. A Coordinated Approach for Practical OS-Level Cache Management in Multi-core Real-Time Systems. In *Proc. of the ECRTS 2013*, pages 80–89, 2013.

16 Hyoseung Kim and Ragunathan (Raj) Rajkumar. Predictable Shared Cache Management for Multi-Core Real-Time Virtualization. *ACM Trans. Embed. Comput. Syst.*, 17(1):22:1–22:27, December 2017.

17 N. Kim, B. C. Ward, M. Chisholm, C. Fu, J. H. Anderson, and F. D. Smith. Attacking the One-Out-Of-m Multicore Problem by Combining Hardware Management with Mixed-Criticality Provisioning. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, April 2016.

18 T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna. Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Montreal, Canada, April 2019.

19 R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 45–54. IEEE, 2013.

20 R. Mancuso, R. Pellizzoni, M. Caccamo, Lui Sha, and Heechul Yun. WCET(m) estimation in multi-core systems using single core equivalence. In *2015 27th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 174–183, July 2015.

21 M. Mendez, J.L.G. Rivas, D.F. Garca-Valdecasas, and J. Diaz. Open platform for mixed-criticality applications. In *Proc. of the Conference on Design, Automation and Test in Europe, WICERT (DATE)*, pages 1–7, 2013.

22 P. Modica, A. Biondi, G. Buttazzo, and A. Patel. Supporting Temporal and Spatial Isolation in a Hypervisor for ARM Multicore Platforms. In *Proceedings of the IEEE International Conference on Industrial Technology (ICIT 2018)*, pages 1–7, February 2018.

23 T. Mück, A. A. Fröhlich, G. Gracioli, A. Rahmani, and N. Dutt. CHIPS-AHOy: A Predictable Holistic Cyber-Physical Hypervisor for MPSoCs. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 1–8, Samos Island, Greece, 2018.

24 A. Patel, M. Daftedar, M. Shalan, and M. W. El-Kharashi. Embedded Hypervisor Xvisor: A Comparative Analysis. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 682–691, March 2015.

25 R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer. Look Mum, no VM Exits! (Almost). In *Proc. of the 13th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2017)*, pages 13–18, 2017.

26 M. R. Soliman and R. Pellizzoni. PREM-based optimal task segmentation under fixed priority scheduling. In *2019 31st Euromicro Conference on Real-Time Systems (ECRTS)*, pages 1–24, July 2019.

27 The KITTI Vision Benchmark Suite. KITTI, October 2019. Online; accessed 20 April 2019. URL: `http://www.cvlibs.net/datasets/kitti/`.

28 R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo. A Real-Time Scratchpad-Centric OS for Multi-Core Embedded Systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11, April 2016.

29 R. Tabish, R. Mancuso, S. Wasly, R. Pellizzoni, and M. Caccamo. A Real-Time Scratchpad-centric OS with Predictable Inter/Intra-Core Communication for Multi-core Embedded Systems. *Real Time Systems*, 2019.

30 P. K. Valsan, H. Yun, and F. Farshchi. Taming Non-Blocking Caches to Improve Isolation in Multicore Real-Time Systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, April 2016. `doi:10.1109/RTAS.2016.7461361`.

31 Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016 IEEE*, pages 1–12. IEEE, 2016.

32 S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. SD-VBS: The San Diego Vision Benchmark Suite. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 55–64, October 2009.

33 S. Wasly and R. Pellizzoni. Hiding memory latency using fixed priority scheduling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 75–86. IEEE, 2014.

34 Xilinx. Zynq UltraScale+ Device - technical reference manual. URL: `https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf`.

35 Xilinx. Versal: The First Adaptive Compute Acceleration Platform (ACAP). URL: `https://www.xilinx.com/support/documentation/white_papers/wp505-versal-acap.pdf`, 2018. [Online; accessed 16-January-2019].

36 M. Xu, L. Thi, X. Phan, H. Y. Choi, and I. Lee. vCAT: Dynamic cache management using CAT virtualization. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 211–222, April 2017.

37 Y. Ye, R. West, J. Zhang, and Z. Cheng. MARACAS: A real-time multicore VCPU scheduling framework. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 179–190, November 2016.

38 H. Yun, R. Mancuso, Z.P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 155–166. IEEE, 2014.

39 H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64. IEEE, 2013.

40 Y. Zhou and D. Wentzlaff. MITTS: Memory Inter-arrival Time Traffic Shaping. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 532–544, June 2016. `doi:10.1109/ISCA.2016.53`.