32nd Euromicro Conference on Real-Time Systems

ECRTS 2020, July 7–10, 2020, Virtual Conference

Edited by

Marcus Völp

Editors

Marcus Völp
University of Luxembourg, Luxembourg
marcus.voelp@uni.lu

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 978-3-95977-152-8

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/978-3-95977-152-8.

Publication date
June, 2020

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.2020.0

ISBN 978-3-95977-152-8 **ISSN 1868-8969** **https://www.dagstuhl.de/lipics**

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

32nd Euromicro Conference on Real-Time Systems (ECRTS 2020).
Editor: Marcus Völp
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

Message from the Chairs

Welcome to the **32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)** in . . . , well this year we are everywhere on earth, in all our homes to fight SARS-COV-2. 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 2020, we received submissions from 20 countries, a majority of 67% of which are from or with contributions of authors from outside Europe. 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 60 external reviewers. The submissions were evaluated according to their contribution, originality, technical correctness and writing quality. The program committee selected in a virtual meeting 24 excellent works for publication in the proceedings and presentation at the conference. This accounts for an acceptance rate of 34%.

From the 24 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 2020, 6 papers (25% 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.

Unfortunately, due to the difficulties imposed by COVID 19, we had to postpone all workshops to ECRTS 2021. A special thanks goes to the organizers of the workshops, who already spent a lot of time organizing their respective events. We hope to see you all at ECRTS 2021 in Modena.

For the same reasons as described above, we had to pospone the ECRTS 2020 work-in-progress session, journal-to-conference presentations, the industrial challenge and, unfortunately also the keynote speaches to ECRTS 2021. Thanks a lot to all those starting to organize these sessions and a special thanks to our keynote speakers. Now people will be even more ancious to hear your talks in 2021.

Being a virtual conference, we had the unique opportunity to learn what defines our conference. Hopefully we found replacements for what is most precious to you about ECRTS, and maybe we will keep some parts also when we all can meet again, for example at ECRTS 2021 in Modena, Italy. Our thoughts are with all those suffering from the pandemic and our gratitude with those that keep our societies operational. We all felt humbled when we

32nd Euromicro Conference on Real-Time Systems (ECRTS 2020).
Editor: Marcus Völp

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

realized that in most parts its not the scientific and industrial elite, which brings us through such hard times, but the tellers in the supermarkets, the nurses and many others.

ECRTS 2020 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 2020; the **artifact evaluation chairs** and the **artifact evaluators** who help this conference pave the way for reproducible research; the **workshop chairs** most of which had to postpone their events after putting a lot of effort in preparation; and Dagstuhl Publishing for their support in publishing these proceedings. Many thanks to the organization committee for their help in turning ECRTS 2020 into a virtual conference. We also thank **Sophie Quinton** for sharing her experience as the ECRTS 2019 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 2020. This conference would not exist without you and we are proud of the high quality and scientific relevance of this year's program. Let us now enjoy ECRTS 2020!

Marko Bertogna
General Chairs, ECRTS 2020

Marcus Völp
Program Chair, ECRTS 2020

Committees

General Chairs

Marko Betogna, University of Modena, Italy

Local Organization Chair

Micaela Verucchi, University of Modena, Italy

Program Chair

Marcus Völp, SnT, University of Luxembourg

Real-Time Technical Committee Chair

Gerhard Fohler, TU Kaiserslautern, Germany

Artifact Evaluation Chairs

Alessandro Papadopoulos, Mälardalen University, Sweden
Alessandro Biondi, Scuola Superiore Sant'Anna, Pisa, Italy

Workshop Chairs

Workshops General Chair

Sebastian Altmeyer, Universität Augsburg, Germany

CERTS - Security and Dependability of Critical Embedded Real-Time Systems
António Casimiro, University of Lisboa, Portugal
Jeremie Decouchant, Snt, University of Luxembourg

OSPERT - Operating Systems Platforms for Embedded Real-Time Applications
Daniel Lohmann, Leibnit Universität Hannover, Germany
Renato Mancuso, Boston University, USA

RT-Cloud - Real-Time Cloud
Johan Eker, Lund University and Ericsson, Sweden
Luca Abeni, Scuola Superiore Sant'Anna, Italy

RTSOPS - Real-Time Scheduling Open Problems Seminar
Mitra Nasri, TUDelft and Eindhoven University of Technology, Netherlands
Alessandro Biondi, Scuola Superiore Sant'Anna, Italy

WATERS - Analysis Tools and Methodologies for Embedded and Real-time Systems
Selma Saidi, TU Dortmund, Germany
Paolo Burgio, University of Modena and Reggio Emilia, Italy

WCET - Worst-Case Execution Time Analysis
Clément Ballabriga, Lille University, France

32nd Euromicro Conference on Real-Time Systems (ECRTS 2020).
Editor: Marcus Völp
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Program Committee

Benny Akesson, ESI (TNO) / University of Amsterdam, The Netherlands
Sebastian Altmeyer, University of Augsburg, Germany
Sanjoy Baruah, Washington University in St. Louis, USA
Andrea Bastoni, SYSGO GmbH, Germany
Marko Bertogna, UNIMORE, Italy
Timothy Bourke, INRIA Paris, France
Björn Brandenburg, Max Planck Institute for Software Systems, Germany
Francisco J. Cazorla, Barcelona Supercomputing Center, Spain
Johan Eker, Lund University and Ericsson, Sweden
Rolf Ernst, TU Braunschweig, Germany
Nathan Fisher, Wayne State University, USA
Gerhard Fohler, Technische Universität Kaiserslautern, Germany
Steve Goddard, University of Iowa, USA
Joël Goossens, Université libre de Bruxelles, Belgium
Arne Hamann, Robert Bosch GmbH, Germany
Angeliki Kritikakou, University Rennes, IRISA, Inria, France
Adam Lackorzynski, Kernkonzept GmbH and TU Dresden, Germany
George Lima, Federal University of Bahia, Brasil
Martina Maggio, Lund University, Sweden
Geoffrey Nelissen, CISTER, ISEP, Portugal
Ramon Serna Oliver, TTTech, Austria
Claire Pagetti, ONERA/ENSEEIHT, France
Alessandro Vittorio Papadopoulos, Mälardalen University, Sweden
Isabelle Puaut, University of Rennes and INRIA, France
Rodolfo Pellizzoni, University of Waterloo, Canada
Sophie Quinton, INRIA Grenoble Rhône-Alpes, France
Gordana Rakic, University of Novi Sad, Serbia
Christine Rochange, University of Toulouse, France
Jean-Luc Scharbarg, University of Toulouse – IRIT – INPT/ENSEEIHT, France
Patrick Meumeu Yomsi, CISTER Research Centre, ISEP, Portugal
Dirk Ziegenbein, Robert Bosch GmbH, Germany

Artifact Evaluators

Leonidas Kosmidis, Barcelona Supercomputing Center, Spain
Daniel Casini, Scuola Superiore Sant'Anna, Italy
Paolo Pazzaglia, Scuola Superiore Sant'Anna, Italy
Georg von der Brüggen, Max Planck Institute for Software Systems, Germany
Syed Aftab Rashid, ISEP IPP, Portugal
Anway Mukherjee, Virginia Tech University, USA
Matthias Becker, KTH Royal Institute of Technology in Stockholm, Sweden
Micaela Verucchi, University of Modena, Italy
Corey Tessler, Wayne University, USA

Additional Reviewers

Aakash Soni
Aaron Willcock
Adam Kostrzewa
Antoine Bertout
Anway Mukherjee
Arthur Clavière
Borislav Nikolic
Carles Hernandez
Claudio Mandrioli
Corey Tessler
Dakshina Dasari
Damien Masson
David Trilla
Enrico Mezzetti
Florian Heilmann
Frédéric Boniol
Gautam Gala
Gautham Nayak Seetanadi
Hamid Tabani
Houssam Zahaf
Ignacio Sanudo
Ivan Rodriguez
James Orr
Javier Barrera
Johannes Schlatow
Jonas Peeck
Jordi Cardona
Jorge Luis Martinez Garcia
Jérôme Ermont
Konstantinos Bletsas
Kristin Krüger
Leonidas Kosmidis
Leonie Köhler
Liliana Cucu-Grosjean
Lukas Krupp
Maksym Planeta
Marco Maida
Marco Perronet
Marco Solieri
Marine Kadar
Micaela Verucchi
Michael Pressler
Michael Roitzsch
Miguel Alco
Mischa Möstl
Nathanael Sensfelder
Nicola Capodieci
Nils Vreman
Oana Hotescu
Reyhaneh Karimipour
Roberto Cavicchioli
Roger Pujol
Sebastian Schildt
Selma Saidi
Sergey Bozhko
Thawra Kadeed
Thomas Carle
Tobias Blass
Xavier Poczekajlo

Fixed-Priority Memory-Centric Scheduler for COTS-Based Multiprocessors

Gero Schwäricke
Technical University of Munich, Germany
gero.schwaericke@tum.de

Tomasz Kloda
Technical University of Munich, Germany
tomasz.kloda@tum.de

Giovani Gracioli
Federal University of Santa Catarina, Brazil
giovani@lisha.ufsc.br

Marko Bertogna
Università di Modena e Reggio Emilia, Italy
marko.bertogna@unimore.it

Marco Caccamo
Technical University of Munich, Germany
mcaccamo@tum.de

Abstract

Memory-centric scheduling attempts to guarantee temporal predictability on commercial-off-the-shelf (COTS) multiprocessor systems to exploit their high performance for real-time applications. Several solutions proposed in the real-time literature have hardware requirements that are not easily satisfied by modern COTS platforms, like hardware support for strict memory partitioning or the presence of scratchpads. However, even without said hardware support, it is possible to design an efficient memory-centric scheduler.

In this article, we design, implement, and analyze a memory-centric scheduler for deterministic memory management on COTS multiprocessor platforms without any hardware support. Our approach uses fixed-priority scheduling and proposes a global "memory preemption" scheme to boost real-time schedulability. The proposed scheduling protocol is implemented in the Jailhouse hypervisor and Erika real-time kernel. Measurements of the scheduler overhead demonstrate the applicability of the proposed approach, and schedulability experiments show a 20% gain in terms of schedulability when compared to contention-based and static fair-share approaches.

2012 ACM Subject Classification Computer systems organization → Embedded systems; Computer systems organization → Multicore architectures; Software and its engineering → Real-time schedulability; Security and privacy → Virtualization and security

Keywords and phrases Schedulability Analysis, Scheduler Implementation, memory-centric Scheduling, Virtualization, Multiprocessor

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.1

Funding *Marco Caccamo*: Marco Caccamo was supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research.

1 Introduction

In commercial-off-the-shelf (COTS) multiprocessor systems, a task's execution time can increase by an order of magnitude due to shared main memory interference, generated by tasks running simultaneously on other cores [25]. Bounding or even eliminating this interference is highly desirable in real-time systems. The *PRedictable Execution Model* (*PREM*) [40,41] and

its extensions [6, 8, 15, 57] attempt to mitigate the memory contention problem by splitting a task's execution into separate phases for memory transactions and pure computation. During a memory phase, task data is fetched from the shared main memory to a fast local memory (private cache partition or scratchpad). During a computation phase, a task performs the computation on the prefetched data without the need to access the shared main memory. A memory-centric scheduler ensures that memory phases of tasks running on different processors are scheduled in non-overlapping time slots.

Many successful implementations of memory-centric schedulers [19, 41, 52, 53, 57] have adopted time-division multiplexing (TDM) as the underlying principle. TDM-based arbitration distributes the resource main memory in a very predictable way. One known downside of this approach is the underutilization of the resource: if a processor has a reserved time slot but does not use it, the slot cannot be offered to another processor [23, 49]. A priority-based memory scheduler may permit to schedule task sets with higher processor utilization due to its work-conserving nature [5].

The prohibitively high preemption cost of memory transactions favors non-preemptive scheduling. This is particularly relevant when using Direct Memory Access (DMA) controllers, which can effectively increase the overall system performance by enabling an overlap of memory transactions and computation [3, 20, 53, 57]. Unfortunately, most COTS processors use cache memories without any architectural support for offloading memory operations (e.g., cache stashing). Nevertheless, the processor can still temporarily suspend and resume its own prefetch operation. The ability to suspend a processors prefetch operation can be exploited to enable global memory preemptions in memory-centric scheduling and thus remove blocking caused by low priority tasks running on other cores.

State-of-the-art approaches for the implementation of memory-centric scheduling rely on specific hardware support (e.g., cache locking [3, 46, 60] or the presence of scratchpads [19, 53, 57]). Such features may not be present on all modern COTS multiprocessor systems. For instance, the ARM Cortex A-57 (used in Nvidia Tegra TX1/TX2 and Exynos 7 Octa) and its more energy-efficient alternative the ARM Cortex A-53 (used in Raspberry Pi 3) have no scratchpads and no explicit locking support for caches.

This paper. To address these shortcomings and enable a contentionless memory scheduling, we make the following contributions:

- design of a fixed-priority preemptive memory-centric scheduler for modern COTS multiprocessor systems that do not feature hardware support for predictable memory management,
- implementation of the memory-centric scheduler within a hypervisor and a real-time operating system kernel that are based entirely on open-source software components,
- evaluation of the effectiveness and the limitations of the proposed approach with a set of microbenchmarks and real-world workloads,
- fixed-priority partitioned multiprocessor schedulability analysis for PREM-compliant task sets backed by schedulability experiments that incorporate the actual memory arbitration overheads in the analytical model and identify a heuristic for task partitioning.

Paper structure. The remainder of the paper is organized as follows. In Section 2, we discuss the design decisions, design alternatives, and rationale behind the proposed memory-centric scheduler. The model of the scheduler, and the system as a whole, are formalized in Section 3, followed by the schedulability analysis in Section 4. The implementation details are given in Section 5, and the evaluation is presented in Section 6. Finally, Section 7 reviews the related work, and Section 8 concludes the paper.

2 System Design and Assumptions

Figure 1 shows an overview of our system design. A hypervisor provides memory-centric scheduling and resource partitioning for real-time operating system (RTOS) guests and controls the communication among RTOSs through interprocessor interrupts (IPIs). The objective is to rule out interference among the RTOSs due to concurrent main memory accesses. In the next subsections, we describe the assumptions required by this work considering the system architecture and briefly discuss our design choices.

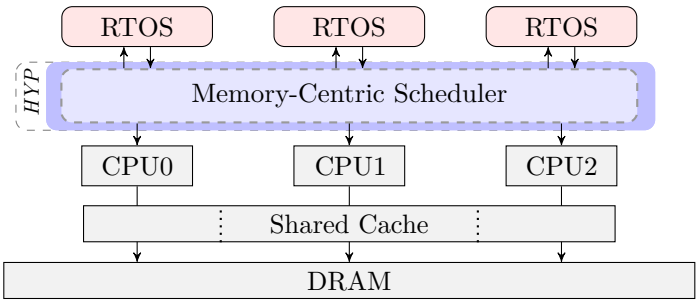


Figure 1 System architecture example for $N = 3$ real-time operating systems.

2.1 Predictable Execution Model

To minimize the amount of main memory interference, the processors service their tasks according to the *PREM*. Such tasks are split into a memory phase and a computation phase. The memory phase prefetches all required task data into a local memory section and is therefore recognizable by a large number of main memory accesses. The computation phase uses only data that was prefetched in a prior memory phase and shows almost no main memory accesses.

2.2 Local Memory

Data that was prefetched during memory phases must be kept in a fast local memory. Two types of local memories are typically used: last-level caches and scratchpads.

When using scratchpads, memory blocks are moved to/from the main memory explicitly in software (usually with the help of a DMA controller). The deterministic nature of scratchpads makes them a suitable choice in real-time systems [43]. However, support for scratchpad memories is rare in commercially available platforms [19, 36].

Last level caches (LLC), as available in most COTS platforms, can store considerable amounts of data copies of frequently used main memory sections. LLCs are self-managed and generally perform well on generic workloads, but introduce some degree of non-determinism because their memory is shared among the processors of a multiprocessor system. The cached data from one core can be evicted by another core if they are using data in memory spaces that are mapped to the same cache location (index). A cache partitioning technique can reduce this interference (see the next subsection). In this work, we consider processor architectures that have a shared LLC and no scratchpads.

2.3 Cache Partitioning

To protect cached data from eviction, cache partitioning assigns a portion of the cache to a given task or core for its exclusive use [19]. It can be done using either a hardware feature (e.g., Intel's Cache Allocation Technology [51], ARM's Lockdown by master [4]) or a software method (e.g., cache coloring [24, 31]). Many embedded platforms do not feature hardware cache partitioning (e.g., platforms based on ARM Cortex-A57 or A72). Thus, in this work, we use software cache partitioning based on cache coloring.

Cache coloring uses virtual memory to take control over the placement of virtually addressed memory in a physically-indexed cache. Memory pages mapped to the same cache sets are said to share the same color. Thus pages with different colors have different locations within the cache. The cache is partitioned by assigning distinct colors to different tasks or cores. The maximal number of distinct colors depends on the cache geometry (size, number of ways, cache line size) and the size of physical pages.

We assume that the hypervisor provides cache coloring to guest OSs, which overcomes many practical problems related to the modification of memory allocators in the OSs. Hardware-assisted virtualization features an additional address translation stage at which intermediate physical addresses (IPA), used by the guests, are translated into actual physical addresses (PA). The page tables at this translation stage are controlled by the hypervisor and can be configured to assign only pages of specific colors to a guest OS [20].

All tasks on a processor can inherit the same set of assigned colors [26], or each task can be assigned a subset of the processor's color set [60]. The latter approach requires a coloring-aware OS to distribute the colors among the tasks [26]. Apart from several academic implementations [19, 31, 62, 64], such a technique is not common. We assume a coloring-unaware OS. The model presented in this paper is also valid for hardware cache partitioning.

Figure 2 illustrates cache coloring implemented in a virtualized environment hosting two guests such that the first guest (CPU0) is allocated to the upper two and the second guest (CPU1) to the lower two sets of a 4-way set-associative L2 cache.

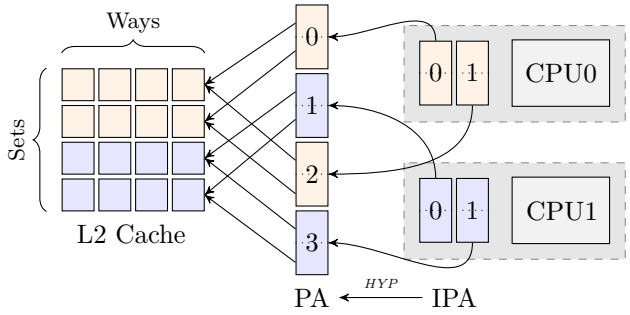


Figure 2 Cache coloring at the hypervisor level for a 4-way set-associative L2 cache.

2.4 Memory Prefetching

Several scheduling frameworks [20, 48, 53] take advantage of DMA offloading to parallelize memory prefetching of newly released tasks alongside ongoing computation. Writing directly to cache with a DMA controller (e.g., ARM's cache stashing) is featured on certain embedded platforms (e.g., Freescale P4080, ARM Cortex-A9, new Cortex-A75, and A55). However,

this mechanism is not widely available [60] (as is the case for Cortex-A57). Without this "passive prefetching", the cache partition must be loaded actively by the processor executing prefetch instructions (such as the `prfm` assembly instruction in ARM architectures).

When prefetching data into cache-based local memory, the cache replacement policy must be taken into account to avoid the problem of self-eviction [40, 41]. Since congruent addresses contend for the cache lines of the same set, the cache replacement policy decides in which cache way of the set the data is placed.

A popular replacement policy, pseudo-random (as in Cortex-A57), works as follows: if there is at least one way in the target set whose cache line is empty (invalid) then the new data is simply copied into that empty cache line; otherwise, the controller randomly selects a way and unloads its cache line to make space for the new data. To prevent self-evictions during prefetching, a sufficient number of unused cache lines must be invalidated before [28, 36].

2.5 Multiprocessor Scheduler and Preemptions

Global non-preemptive multiprocessor scheduling has recently received increasing attention [3, 33, 39, 58]. On top of that, cache-aware scheduling algorithms were designed, guaranteeing that two running tasks' cache spaces do not overlap at any time [21]. These works assume that either all tasks can fit into the cache or cache partitions can be reassigned arbitrarily at runtime. Yet, efficiently implementing dynamic cache partition reallocation can have very high overheads [42, 63]. Also, in direct-mapped and set-associative physically indexed caches, the memory of two different tasks can be mapped to the same cache sets. Without full control of way-placement (e.g., ARM's Lockdown by way, as in [34]), the concurrent execution of such tasks can cause mutual interference. Because of the mentioned limitations, lower runtime overheads, static isolation, and easier extension to heterogeneous platforms [7, 13], we opt for partitioned multiprocessor scheduling.

Tasks assigned to the same processor use the same cache partition. In non-preemptive task scheduling, each task can use the entire partition. In preemptive task scheduling, the partition must be divided among the tasks, leading to less local memory for each task. To maximize the task data footprint, we consider a non-preemptive scheduler. If some degree of preemption is required, limited preemption [9], stack-sharing techniques, or cache-cognizant scheduling [54] can be used to increase memory efficiency.

Since the processors have non-overlapping cache partitions, tasks running on one processor cannot evict the prefetched data of the other processors. Processors can also stop their ongoing cache prefetching operation at any time. Therefore, a processor can preempt another processor during its memory phases using interprocessor interrupts (as will be shown in Section 6.1).

2.6 Priority-based Scheduling

We consider fixed-priority scheduling due to its widespread usage in real-time operating systems (e.g., VxWorks, FreeRTOS, Erika). As stated before, tasks follow the PREM. However, we performed a slight modification to the original model: we allow global memory preemptions during the memory phases (a task can be preempted during its memory phase by tasks running on another processor, but cannot be preempted by tasks running on the same processor). This modification helps to reduce the blocking time caused by low priority tasks [9].

Global memory preemptions during memory phases may lead to uncontrolled *priority inversion* [44, 50]. For instance, consider the case illustrated in Figure 3. A high priority task τ_H is waiting for a low priority task τ_L scheduled on the same processor P_1 due to local

non-preemptive scheduling. Task τ_M, with medium priority, running on processor P_2, can preempt the low priority task τ_L during its memory phase and increase the blocking time of task τ_H. Priority inversion can be countered by adopting a synchronization protocol [44, 50], where each protocol comes with a different implementation complexity and inversion bound.

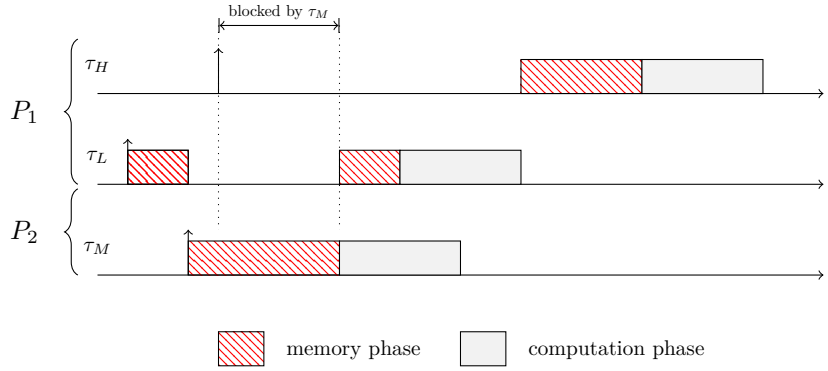


Figure 3 Priority inversion in fixed-priority inter-processor preemptive and intra-processor non-preemptive memory-centric scheduling.

To avoid this problem, our design enforces a strict priority order between the tasks allocated to two different processors. Each processor has a statically assigned unique memory access priority that is inherited by all tasks allocated to it. Our approach comes with the advantage of low runtime overhead compared to a priority inheritance protocol: a registered memory access request does not have to be updated if a new task is released on the same processor. The main drawback lies in additional complexity in the task-to-processor allocation, which we will discuss in Section 6.3.

3 System Model

We consider a multiprocessor platform with shared last level cache and shared main memory on which a partitioned scheduler executes a set of sporadic tasks on each processor. The tasks are composed of a single memory and a single computation phase.

Locally, the scheduler uses a fixed-priority non-preemptive policy. As an exception, memory phases can be preempted by tasks running on processors of higher priority, as described in Section 2.6.

3.1 Processors

The main memory is shared among N processors. Each processor is assigned a unique static priority. The priorities govern the contentionless access to the main memory. Processors are indexed in priority order with P_1 having the highest priority and P_N the lowest priority. We say that $P_h > P_l$ if processor P_h has a higher priority than processor P_l.

3.2 Tasks

We consider a partitioned system in which each task is statically assigned to a single processor. We say that $\tau_i \in P$ if τ_i is allocated to the processor P, and we denote by $P(i)$ the processor to which task τ_i is allocated. Each task τ_i gives rise to a potentially infinite sequence of jobs. Task τ_i releases jobs sporadically after the minimum inter-arrival time T_i (period), and each job of τ_i must be completed within a fixed time interval from its release given by a relative deadline D_i. We assume that tasks have constrained deadlines, i.e., $D_i \leq T_i$.

Each task is composed of two phases: a memory phase and a computation phase. During its memory phase, task τ_i fetches data from the main memory to the local cache partition. The prefetch operation is supposed to fully occupy the processor that is unavailable to perform other work during that time. In its computation phase, the task performs the computation on the prefetched data. Thus the task does not access the main memory. The maximal time to complete the memory phase, with exclusive memory access, is m_i (no memory interference from the other processors), and the maximal time to complete the computation phase, executed in isolation from the other tasks, is c_i (no other tasks running on the same processor). The total worst-case execution time of task τ_i is given as $e_i = m_i + c_i$. All the above parameters are positive integers. The total memory utilization of all the tasks allocated to processor P is given as $U^m(P) = \sum_{\tau_j \in P} \frac{m_j}{T_j}$ and the total utilization as $U(P) = \sum_{\tau_j \in P} \frac{e_j}{T_j}$.

The tasks are assumed to be independent and do not share resources other than the processor and the local memory partition. The worst-case execution time includes preemption, context switch, and scheduler overheads. Once a task starts, it will not voluntarily suspend its execution. Tasks are scheduled on each processor by a fixed-priority non-preemptive scheduler and are indexed in priority order with τ_1 having the highest priority and τ_n the lowest priority where n is the number of tasks allocated to the processor under study. Each task has a unique priority. We introduce the notation $hp(i)$ and $lp(i)$ for the set of tasks with priorities, respectively, higher than and lower than the priority of task τ_i, which are running on the same processor ($hp(i) \subseteq \{\tau_j \,|\, \tau_j \in P(i)\}$ and $lp(i) \subseteq \{\tau_j \,|\, \tau_j \in P(i)\}$). The global fixed-priority preemptive scheduler schedules the accesses to the main memory: a task running in its memory phase can be preempted by the memory phase of a task running on a processor with higher priority. Only one task can access the main memory at a time. If $P_h > P_l$, then all tasks allocated to processor P_h have higher memory access priority than all tasks allocated to processor P_l.

3.3 Memory

Memory is a globally shared resource and can be accessed by all processors incurring the same memory access latencies for each processor. We assume that the processor waits synchronously for every prefetch instruction caused either by a cache miss or an explicit prefetch instruction. We assume the order in which the memory controller serves memory requests issued simultaneously from different processors to be unknown. We assume that the local cache memory has much higher bandwidth and lower latency than the main memory. Each processor has a dedicated cache partition of a fixed size. The largest memory footprint of any real-time task allocated to a processor can fit entirely into the processor's local memory partition. The partitions are non-overlapping.

4 Schedulability Analysis

In the following, we give details of the schedulability analysis for two-phase *PREM*-compliant tasks running under the fixed-priority memory-centric scheduler described in the previous sections. We first identify the different sources of interference and show how the interference can be upper bounded. We then integrate the obtained factors into the response time analysis.

Figure 4 shows a sample execution of a task set on two processors. Processor P_1 has higher priority than processor P_2. Tasks are indexed in decreasing priority order and execute non-preemptively. Tasks may experience blocking from low priority tasks allocated to the

same processor (e.g., task τ_2 blocked by task τ_4 at $t = 0$). However, their memory phases can be preempted by the memory phases of tasks running on the other processors (e.g., task τ_2 preempted by task τ_1 at $t = 4$).

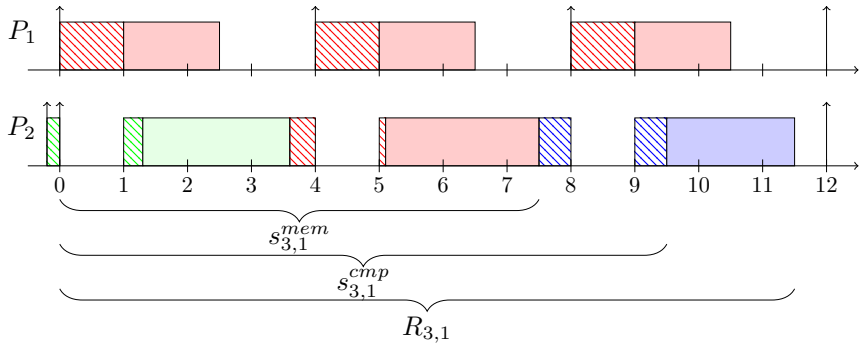


Figure 4 Sample schedule for two processors. P_1 has a higher priority than P_2. Task running on P_1: $\tau_1 : (m_1 = 1.0, c_1 = 1.5, T_1 = 4)$. Tasks running on P_2: $\tau_2 : (m_2 = 0.5, c_2 = 2.4, T_2 = 12)$, $\tau_3 : (m_3 = 1.0, c_3 = 2.0, T_3 = 12)$, and $\tau_4 : (m_4 = 0.5, c_4 = 2.3, T_4 = 24)$.

Sources of interference. The worst-case response time R_i of task τ_i is composed of the following factors:

- *Blocking* due to a lower priority task on the same processor that starts its execution just before the release of task τ_i.
- *Intra-processor interference* due to the tasks with higher priority than task τ_i scheduled on the same processor. This interference can delay the start of a task running on $P(i)$.
- *(Inter-processor) memory interference* due to the memory phases of tasks scheduled on higher priority processors. This interference can only delay the execution of memory phases running on the processor under analysis.

4.1 Interference Calculation

We characterize the interference that task τ_i running on processor $P(i)$ can experience during an interval of length $\Delta > 0$ due to blocking, intra-processor interference, and inter-processor memory interference.

Formulas for blocking and intra-processor interference can be obtained straightforwardly. The blocking factor B_i is the longest worst-case execution time among all tasks with priority lower than τ_i allocated to the same processor. The maximum of an empty set is assumed to be zero.

$$B_i = \max_{\tau_j \in lp(i)} e_j \tag{1}$$

Intra-processor interference $I_i(\Delta)$ is the sum of the worst-case execution times of all higher priority task instances that can be released on the same processor within the time interval Δ:

$$I_i(\Delta) = \sum_{\tau_j \in hp(i)} \left\lceil \frac{\Delta}{T_j} \right\rceil e_j \tag{2}$$

The problem of estimating the inter-processor memory interference cannot be solved with the above formula and requires additional analysis. This is due to the fact that i) the

memory interference of the two-phase model can exhibit a jitter behavior, and ii) within the processor busy period preemption from the other processors can occur only during memory phases, which are interleaved with the non-preemptive computation phases.

The jitter behavior is evidenced by the fact that the time between the start of the memory phases of two consecutive instances of the same task can be shorter than its period. Consider the example from Figure 5. We try to characterize the memory interference generated by the memory phases of task τ_1. The first instance of task τ_1 is blocked by the lower priority task τ_2. The first memory phase of τ_1 starts at time instant 2. Every following memory phase of τ_1 does not start earlier than at its next release, that is, at time instants 4, 8, 12, ... The time distance between the start times of the first and the second memory phase is shorter than between the second and the third memory phase.

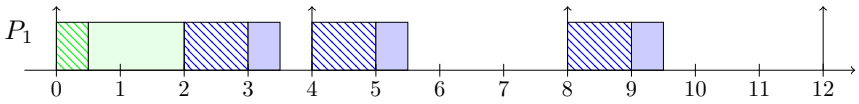


Figure 5 Inter-processor memory interference from a single processor. Tasks running on processor P_1: $\tau_1 : (m_1 = 1.0, c_1 = 0.5, T_1 = 4)$ and $\tau_2 : (m_2 = 0.5, c_2 = 1.5, T_2 = 12)$.

This is equivalent to the problem of *inherited release jitter* [55]. When estimating memory interference generated by a task from another processor, we may assume that its release jitter is equal to its latest start time. The latter can be safely upper bounded by $R_i - e_i$ as proposed in [11] for remote blocking. According to the above approach, the inter-processor memory interference from tasks running on processors with a higher priority than processor $P(i)$ on which τ_i is running can be expressed as:

$$\alpha_i(\Delta) = \sum_{P_h > P(i)} \sum_{\tau_j \in P_h} \left\lceil \frac{\Delta + R_j - e_j}{T_j} \right\rceil m_j \tag{3}$$

Not every external memory phase within the busy period of the processor under analysis can interfere with the memory phases that this particular processor runs. Consider the example shown in Figure 6. In time interval $[0, 9]$, task τ_1 running on processor P_1 can release up to 3 memory phases. However, not all of these memory phases can interfere with the execution of a task running on processor P_2. Indeed, processor P_2 is running a computation for a long time (from time instant 2 to 9), which is not affected by the memory interference from processor P_1.

Evaluating the number of times the processor is exposed to the memory interference from the higher priority processors can reduce the pessimism stemming from the assumption that every external memory phase is interfering. For instance, in the example from Figure 6, processor P_2 executes only one memory phase in the interval $[0, 9]$. Therefore it can be assumed that it does not experience the memory interference from processor P_1 more than once.

The number of memory phases released on processor $P(i)$ during the busy-period of task τ_i with length $\Delta > 0$, such that there is a pending instance of τ_i, can be calculated as (we consider higher priority tasks, previous instances of task τ_i and one lower priority task that may block τ_i if τ_i is not the lowest priority task):

$$\mathcal{N}_i(\Delta) = \sum_{\tau_j \in hp(i)} \left\lceil \frac{\Delta}{T_j} \right\rceil + \left\lfloor \frac{\Delta}{T_i} \right\rfloor + [lp(i) \neq \emptyset] \tag{4}$$

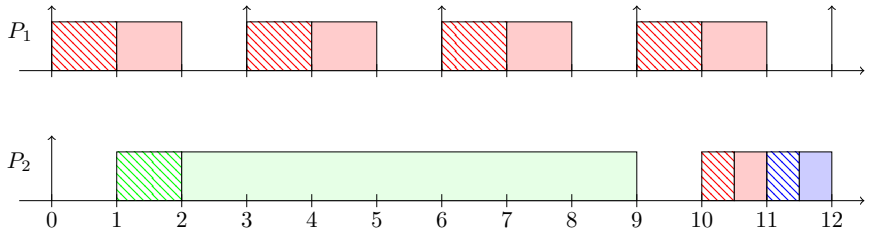


Figure 6 Exposure to inter-processor memory interference in a two processor system. Processor P_1 has a higher priority than processor P_2. Task running on P_1: $\tau_1 : (m_1 = 1.0, c_1 = 1.0, T_1 = 3)$. Tasks running on P_2: $\tau_2 : (m_2 = 0.5, c_2 = 0.5, T_2 = 24)$, $\tau_3 : (m_3 = 0.5, c_3 = 0.5, T_3 = 24)$, and $\tau_4 : (m_4 = 1.0, c_4 = 8.0, T_4 = 24)$.

where $[\cdot]$ is the Iverson bracket evaluating to one for a true expression and to zero otherwise. The maximum interference that can affect one of these memory phases released on processor $P(i)$ is not greater than the least positive fixed-point of the following recurrent equation:

$$\varepsilon_{P(i)} = \alpha_i(\varepsilon_{P(i)} + \widehat{m}_{P(i)}) \quad \text{where} \quad \widehat{m}_{P(i)} = \max\{m_j \,|\, \tau_j \in P(i)\} \tag{5}$$

The term $\widehat{m}_{P(i)}$ designates the longest memory phase among all the memory phases of tasks running on processor $P(i)$. The fixed-point iteration converges if and only if $\sum_{P_h > P(i)} U^m(P_h) < 1$. All in all, the total memory interference from the higher priority processors within interval $\Delta > 0$ is not greater than the number of memory phases released during the busy period times the maximum interference from the higher priority processors that can affect one of these memory phases:

$$\beta_i(\Delta) = \mathcal{N}_i(\Delta) \cdot \varepsilon_{P(i)} \tag{6}$$

This approach can overapproximate the actual memory interference as well. It can happen when memory phases on the processor under analysis are scheduled more frequently than the memory phases of the higher priority processors (e.g., in Figure 6 from time instant 10 to 11.5). If that is the case, an approach in which each external memory phase is considered to be interfering can give a less pessimistic estimation.

Based on these two approaches, we can now formulate a safe upper bound on the inter-processor memory interference from the tasks scheduled on higher priority processors:

$$\min\{\alpha_i(\Delta), \beta_i(\Delta)\} \tag{7}$$

4.2 Response Time Analysis

The worst-case response time of task τ_i running on processor $P(i)$ can be determined by combining the expressions for interference presented above with the analysis for single processor non-preemptive scheduling [9, 14].

The latest start time $s^{mem}_{i,k}$ of the memory phase of the k-th instance of task τ_i is bounded by the least positive fixed-point of the following recurrent equation:

$$s^{mem}_{i,k} = B_i + I_i(s^{mem}_{i,k}) + (k-1) \cdot e_i + \min\left\{\alpha_i(s^{mem}_{i,k}), \beta_i(s^{mem}_{i,k})\right\} \tag{8}$$

Now we compute the latest start time of its computation phase. At $s^{mem}_{i,k}$, task τ_i starts its memory phase. It cannot be delayed by the tasks from the same processor (non-preemptive

local scheduling). From that point on, a further delay can only be caused by the memory interference from tasks running on processors with higher priority (preemption in memory phase):

$$s_{i,k}^{cmp} = B_i + I_i(s_{i,k}^{mem}) + m_i + (k-1)\cdot e_i + \min\left\{\alpha_i(s_{i,k}^{cmp}), \beta_i(s_{i,k}^{mem}) + \alpha_i(s_{i,k}^{cmp} - s_{i,k}^{mem})\right\} \quad (9)$$

Proof. Let $s_{i,k}^{mem}$ and $s_{i,k}^{cmp}$ be respectively, the start and the end of the memory phase of the k-th instance of task τ_i. To ease the notation, we replace $s_{i,k}^{mem}$ with t_s and $s_{i,k}^{cmp}$ with t_e. At t_s, the memory phase of task τ_i starts and cannot be delayed by the tasks running on the same processor due to the non-preemptive scheduling. The interference from higher priority processors within the interval $[0, t_e]$ is upper bounded by $\alpha_i(t_e)$.

First, suppose that $\alpha_i(t_s) > \beta_i(t_s)$. The interference from the higher priority processors within the interval $[0, t_s]$ is upper bounded by $\beta_i(t_s)$ and within the interval $[t_s, t_e]$ by $\alpha_i(t_e - t_s)$. Consequently, within the interval $[0, t_e]$, it cannot be greater than $\min\{\alpha_i(t_e), \beta_i(t_s) + \alpha_i(t_e - t_s)\}$. Now suppose that $\alpha_i(t_s) \leq \beta_i(t_s)$. It must be that $\alpha_i(t_e) \leq \beta_i(t_s) + \alpha_i(t_e - t_s)$ since $\alpha_i(t_e) \leq \alpha_i(t_s) + \alpha_i(t_e - t_s) \leq \beta_i(t_s) + \alpha_i(t_e - t_s)$. Consequently, $\min\{\alpha_i(t_e), \beta_i(t_s) + \alpha_i(t_e - t_s)\} = \alpha_i(t_e)$. ◀

Once started, the computation phase executes non-preemptively and is not subject to any type of interference. The worst-case response time of the k-th instance of task τ_i is:

$$R_{i,k} = s_{i,k}^{cmp} + c_i - (k-1)\cdot T_i \quad (10)$$

To determine the worst-case response time of task τ_i it is necessary to calculate the response time of each instance within the i-level busy period, i.e., the longest time processor $P(i)$ can be busy executing the jobs of tasks with a priority higher than or equal to the priority of task τ_i [30]:

$$L_i = B_i + I_{i-1}(L_i) + \min\left\{\alpha_i(L_i), \beta_i(L_i) + \widehat{m}_{P(i)}\right\} \quad (11)$$

Note that $I_{i-1}(L_i)$ includes the jobs $hp(i)$ as well as the jobs of τ_i. Moreover, $\widehat{m}_{P(i)}$ must be added to $\beta_i(L_i)$ to account for the last instance of τ_i. The fixed-point iteration can be solved for all the tasks of processor $P(i)$ if and only if:

$$U(P(i)) + \min\left(\sum_{P_h > P(i)} U^m(P_h), \sum_{\tau_j \in P(i)} \frac{\varepsilon_{P(i)}}{T_j}\right) < 1 \quad (12)$$

The largest $R_{i,k}$ calculated for the instances of τ_i released within interval L_i gives the worst-case response time:

$$R_i = \max_k R_{i,k} \quad \text{where} \quad k \in \left[1, \left\lceil \frac{L_i}{T_i} \right\rceil\right] \quad (13)$$

Figure 4 shows the scenario corresponding to the worst-case response time of task τ_3.

4.3 Schedulability Test

The task set is said schedulable if all jobs of every task meet their deadlines, i.e., $\forall i : R_i \leq D_i$. Before computing the worst-case response time of task τ_i, the worst-case response times of all higher priority tasks running on the other processors must be calculated first. This is necessary to account for their memory interference in Equation (3). Also note that the factor $\varepsilon_{P(i)}$ from Equation (5) can be computed only once, before starting the response time analysis of the first task on processor $P(i)$. Moreover, the values of the external memory interference given by Equation (7) can be stored and reused to speed up the computation for all the tasks allocated to the same processor.

5 System Implementation

In this section, we describe the implementation of the proposed fixed-priority memory-centric scheduling approach. According to the system design presented in Section 2, the full system has three parts: application tasks, one or more RTOSs, and a hypervisor containing the memory access scheduler. The structure of this section follows these three parts, beginning with a description of requirements for application tasks, followed by implementation details for the support in an RTOS (i.e., Erika), and the implementation of the memory-centric scheduling in a hypervisor (i.e., Jailhouse). At last, we depict the interaction between an RTOS and the hypervisor and describe the concrete implementation that we used for the evaluation.

5.1 Application Task Requirements

Tasks must follow the two-phase PREM as specified in Section 3. The code structure of a suitable PREM task is typically formed by i) a call to a function to prefetch data (memory phase), ii) a call to a function signaling the end of the memory phase (here: `end_memory_phase()`), and iii) the code using the prefetched data (computation phase). All tasks on all RTOSs in the system need to follow this structure to ensure proper guarding of all memory accesses.

Data prefetching is performed using assembly instructions (`prfm` for the ARM architecture). On platforms with random replacement policy, a sufficient number of data cache lines need to be cleaned and invalidated before the data prefetching (`cisw` for the ARM architecture).

5.2 Real-Time Operating System Support

The RTOS provides typical task states (ready, running, waiting) with the addition of a state for tasks that were suspended by the memory arbitration (suspended). When a task enters the ready state, and no prior task is in the suspended state, the RTOS performs a hypercall (trap to hypervisor mode, here: `enable_request`) to request memory access permission from the memory-centric scheduler implemented in the hypervisor. If the memory access is not granted or suspended tasks exist, the task is suspended. Otherwise, it runs the task (begin of the memory phase). At the end of the memory phase (call to `end_memory_phase()`), the RTOS revokes its memory access permission through a hypercall (here: `disable_request`) and disables all interrupts to run the subsequent computation phase without interference. When a task's computation phase ends, the RTOS enables the interrupts again and performs an `enable_request` hypercall for the highest priority task in the suspended state (if any).

The hypervisor can pause and resume a task's memory phase at any point in time due to the preemptive property of the memory-centric scheduling. Pausing and resuming is signaled to the RTOS by interprocessor interrupts (IPIs). When receiving an IPI to pause the memory access, the RTOS suspends the running task. Since tasks in the suspended state are also non-preemptive, the prefetched data will still be in the cache partition when the task is resumed by another IPI.

Figure 7 shows the task states and their transitions in the RTOS: the memory phase can only be accessed once the permission to access the main memory was acquired, either through a hypercall (from the ready state) or an interrupt (from the suspended state). The hypervisor is notified through a hypercall to revoke the access request once the memory phase is completed.

To reclaim execution time when the memory arbitration suspends a task, the RTOS can optionally implement an aperiodic server running preemptively at the lowest priority.

The server's tasks should access only very little memory or work entirely on non-cacheable memory. The effect of the server's memory accesses should be added in the inter-processor memory interference computation for the other processors.

5.3 Hypervisor Support

The hypervisor implements the memory-centric scheduler. The implementation is based on a kind of token passing algorithm. There exists one token in the system, which permits the RTOS possessing it to freely access the main memory. Each RTOS can register (and deregister) one request for the token. Requests have a priority, which is equal to the requesting processor's priority.

The scheduling occurs when one of the hypercalls `enable_request` and `disable_request` is executed. The first registers and the second revokes a token request. The first hypercall's return value informs the calling RTOS if the token was acquired. When registering a request, the hypervisor reclaims the token if it is assigned to a core of lower priority than the request priority by sending an IPI (see Listing 1). When revoking a request, the hypervisor finds the highest priority pending request and passes the token to the associated processor by sending an IPI (see Listing 2).

Listing 1 Enable memory request hypercall.

```
bool enable_request(int request_prio,
                    int request_proc):
  spin_lock(&mem_arbiter)
  memory_requests[request_proc] ←
      request_prio
  if request_prio > token_prio:
    if token_owner is not undefined:
      notify_pause(token_owner)
    endif
    token_owner ← request_proc
    token_prio ← request_prio
  endif
  bool token_acquired = (token_owner ==
                         request_proc)
  spin_unlock(&mem_arbiter)
return token_acquired
```

Listing 2 Disable memory request hypercall.

```
void disable_request(int request_proc):
  spin_lock(&mem_arbiter)
  memory_requests[request_proc] ←
      undefined
  token_owner ← undefined
  token_prio ← undefined
  foreach pending_prio in memory_requests:
    if pending_prio is not undefined and
        (token_prio is undefined or
         pending_prio > token_prio):
      token_owner ← index(pending_prio)
      token_prio ← pending_prio
    endif
  endforeach
  if token_owner is not undefined:
    notify_resume(token_owner)
  endif
  spin_unlock(&mem_arbiter)
return
```

A spinlock is used to control the access to shared scheduler data structures used from the different cores in the arbitration hypercalls. Consequentially, a processor calling the hypercall `enable_request` may have to wait. The proposed implementation assumes Linux's *ticket lock* (first in first out queue-based mechanism). Thus, the spinning processor cannot be bypassed twice by another processor. The worst-case time to register the memory request can therefore be bounded by $wcet(hypercall) + N \cdot wcet(arbitration)$. After winning the arbitration, the token may need to be reclaimed from a different processor. This event may induce a delay Δ_{IPI} needed for the interprocessor interrupt propagation. The maximal time to acquire exclusive memory access is therefore upper bounded by:

$$wcet(hypercall) + N \cdot wcet(arbitration) + \Delta_{IPI} \tag{14}$$

To enable the usage of an OS that does not provide the required support for the model, the hypervisor can optionally suspend all processors of the said OS when the token is assigned and resume the processors otherwise.

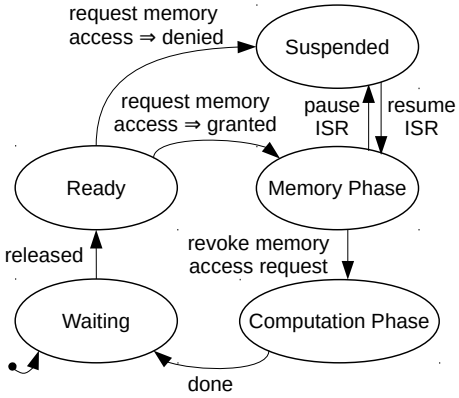


Figure 7 The states and transitions of a PREM task in the RTOS.

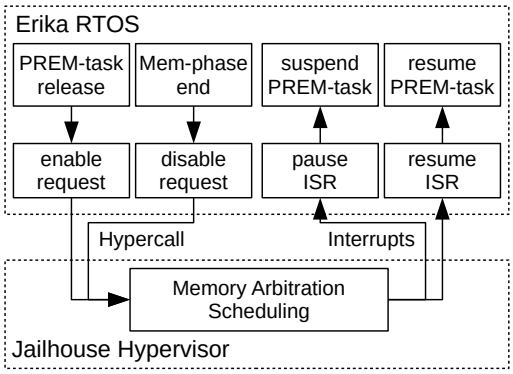


Figure 8 Communication of Erika and Jailhouse in the states of a PREM-task.

5.4 Our Implementation

As RTOS, we use Erika Enterprise version 3, which is an OSEK/VDX certified open-source/commercial RTOS [17]. Erika supports fixed-priority scheduling and runs on several embedded multicore platforms, such as Xilinx Ultrascale+ and NVIDIA Jetson TX2.

We restrict our implementation to PREM-tasks. We use function calls at task release and task completion instead of including the memory access arbitration into the scheduler to reduce implementation complexity. This simplification introduces only a small overhead in case a task is scheduled and then immediately suspended at the `enable_request` hypercall. We added a new task queue to the Erika kernel for tasks suspended by the memory-centric scheduler, since Erika does not support suspending tasks from an interrupt.

As hypervisor, we use *Jailhouse*, which is a partitioning hypervisor based on Linux [27] (required for bootstrapping) that runs on Intel and ARM 64-bit processors featuring hardware-assisted virtualization. Jailhouse statically splits the system into isolated partitions called cells. Each cell runs one guest OS and has full control over its statically assigned resources (e.g., CPUs, memory regions, and PCI devices). We have chosen Jailhouse due to its simplicity and low overhead that favors real-time applications [20, 45].

We implemented the memory-centric scheduler in Jailhouse. Local memory partitions for the RTOSs are created using cache coloring in Jailhouse [20]. We restrict our implementation by not coloring the Jailhouse code. The detrimental effect on the cores' cache partitions can be assumed as low. Figure 8 summarizes the functions in Erika and Jailhouse and the flow of calls.

6 Evaluation

In this section, we provide experimental results showing the overheads incurred by the memory-centric scheduling. We first measure different factors of overheads in microbenchmarks. Then, with a set of benchmarks, we evaluate the overall performance of the implemented system under stress. Finally, we conduct experiments with randomly generated workloads to measure the effectiveness of the schedulability analysis.

The evaluations in Subsection 6.1 and 6.2 are carried out on an Nvidia Tegra TX2 with 4 Cortex-A57 cores, 8GB 128-bit LPDDR4 RAM @ 1866Mhz, 2MB 16-way set-associative shared L2 cache with a pseudo-random replacement policy. For all measurements, Linux was

halted to remove interference from Linux itself or its DMAs on the memory subsystem. The remaining three cores each run Erika RTOS, allowing measurements for low, medium, and high processor priorities. The L2 cache is partitioned into four equal sizes by Jailhouse.

6.1 Microbenchmarks

We measured the duration of hypercalls, IPIs, and the memory arbitration in the memory-centric scheduler using a set of benchmarks that we crafted specifically for this purpose. The benchmarks are implemented as bare-metal applications and *Jailhouse* patches to measure the cost of particular operations using the ARM physical counter-timer. The following benchmarks were executed:

Hypercall The transition from the OS to the hypervisor and back to the OS without any computation in the hypervisor. It measures the cost of a single hypercall (`hvc` instruction).

IPI Issuing an interprocessor interrupt from one processor to another at the hypervisor level. It measures the cost of a single IPI.

Arbitration Arbitration within the hypervisor to find the processor with the highest memory access priority. Four processors ($N = 4$) were assumed.

Table 1 presents the results of *Hypercall*, *IPI*, and *Arbitration* microbenchmarks. Similar evaluations for *KVM* and *Xen* are available in [12]. We collected 1 000 000 samples for each benchmark and extracted the average execution time (AVG), standard deviation (STD), worst-case execution time (WCET), and best-case execution time (BCET) from these samples. Given the times from Table 1, the overhead incurred by the memory-centric scheduler (see Formula (14) with $N = 4$) is less than 6.028 μs. Compared to the worst-case execution times of the memory phases in the real-world application benchmarks (in the following section), the overhead represents less than 6.5%. Moreover, the hypervisor-related overheads have a small impact on the schedulability, as will be shown in Section 6.3.

Table 1 Jailhouse hypercall, Interprocessor Interrupt (IPI), and arbitration overheads (in ns).

	AVG	STD	WCET	BCET
Hypercall	1265.83	191.50	3129	709
IPI	979.65	80.95	1999	935
Arbitration	127.79	8.94	225	64

6.2 Real-World Benchmarks

We performed benchmarks for the full system model in two stages. First, we analyze the maximum interference of RTOSs on the highest priority RTOS with and without our approach. After that, we apply our system model and perform response time benchmarks for two application tasks on each of the cores.

To analyze the interference of parallel prefetching operations, we let two cores prefetch 448 KB of memory (equivalent to the memory size of the benchmarks used below and small enough to fit into a cache partition, which has 512 KB). The prefetching frequency for the high priority core is 300 Hz. The low priority core runs an interfering task whose main loop consists only of prefetching to generate the highest possible interference from a PREM-ized task set running on an RTOS. The memory prefetching time is measured on the high priority core using the ARM physical counter-timer. The experiment is run for 90 000 task activations.

Figure 9 shows the worst-case prefetching time on the high priority core. The results are shown broken down by the enabled (*ours*) or disabled (*legacy*) memory-centric scheduler, and

with the low priority core being idle (*solo*) or generating worst-case interference (*stress*). With our approach preventing unpredictable interference, the prefetching time is bounded, and the performance is not significantly impacted by the memory arbitration related overheads.

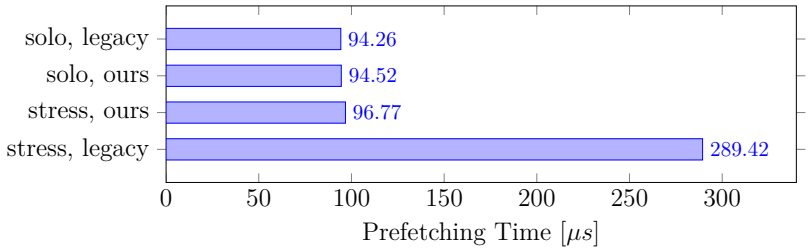
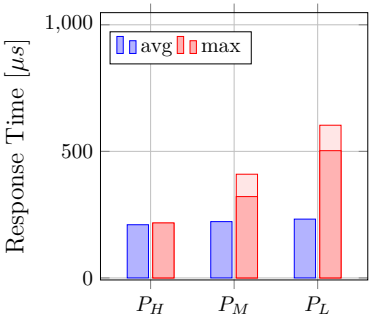


Figure 9 Comparison of prefetching times for a task running on the highest priority core.

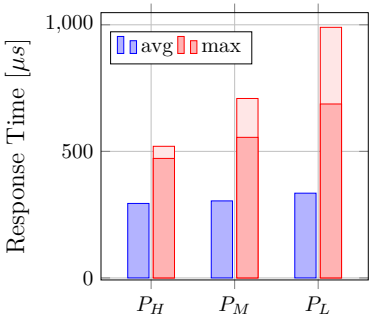
For the second stage of our experiments, we run a task set consisting of two application benchmarks from *TACLeBench* [18], which are compliant with the two-phase PREM. The first task computes an average of integer values and has a high priority (in the task set). It is released with a frequency of 300 Hz. The second task, with a low priority (in the task set), computes a SHA-1 hash and is activated with a frequency of 100 Hz. Both tasks are released synchronously and use 448 kB of data. We run the benchmark on three cores, each with a different processor priority. The measurement is taken identically to the first experiment over 90 000 task activations for the high priority task and 30 000 task activations for the low priority task.

When the response time is measured on the high priority core, the medium priority core runs the interference task described in the first experiment while the low priority core is idle. When the measurement is done on the medium priority core, the high priority core runs the benchmark task set, while the low priority core runs the interference task. The high and medium priority cores both run the benchmark task set when the response time is measured on the low priority core.

In Figure 10, the average and worst-case response time for the high and low priority task is shown. The analytical worst-case response time for the high priority task takes the synchronous release of the tasks into account and thus does not incorporate the blocking from the low priority task scheduled on the same processor.



(a) High priority task.



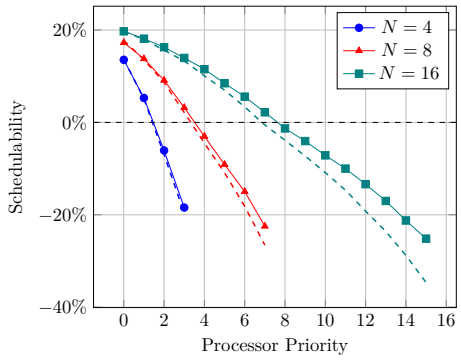
(b) Low priority task.

Figure 10 Response times for the *TACLeBench* benchmark task set executed on the high (P_H), medium (P_M), and low priority (P_L) core. The figures show the average (blue), measured (red), and analytical (light red) worst-case response time.

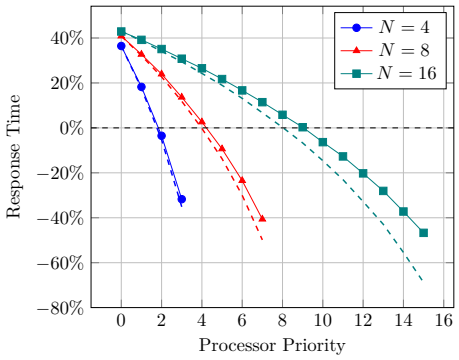
6.3 Schedulability Analysis Evaluation

We generate a sporadic task set with a given number of tasks and a given utilization. We randomly generate period values in the range $[10\,000, 100\,000]$ microseconds with log-uniform distribution, as suggested in [16]. We also use the Emberson et al. unbiased method [16] to generate random task-utilization values that sum to the requested total utilization. Based on the application profiles provided in [61], the ratio between memory and computation phases is generated randomly with a uniform distribution in the range $[0.05, 0.20]$. Tasks are assumed to have implicit deadlines and are assigned priorities based on the Rate-Monotonic priority assignment policy. We consider platforms with $N \in \{4, 8, 16\}$ processors. We compare fixed-priority scheduling to contention-based scheduling [59] (i.e., each processor is assigned $1/N$ of the memory bandwidth) and round-robin scheduling with the quantum size 1 for each processor (bandwidth is shared equally among all processors having pending memory requests).

Schedulability and response time. We investigate the impact of the processor priority on the task's worst-case response time and the improvement of the task schedulability. Each processor is assigned 8 tasks with a total utilization of 0.6. The memory utilization is scaled down proportionally to the processor count (i.e., the bandwidth utilization in the system is constant). 10 000 task sets were randomly generated. If a task was not schedulable under a given policy, its worst-case response time was considered double of the worst-case response time under any other scheduling policy. The results are shown in Figure 11. The baseline represents contention-based scheduling [59]. Round-robin is omitted in Figure 11: for schedulability, its improvement is 1%, and for the reduction of worst-case response times 2%. Additionally, we integrated the scheduler overheads (see Formula 14 and Section 6.1) in the response time analysis by inflating the generated WCET of memory phases with the overheads shown in Table 1. The dashed and the plain lines represent the results, respectively, with and without overheads. As expected, the fixed-priority policy improves the schedulability and reduces the response times on the high priority processors at the expense of the low priority processors. The tasks running on the low priority processors suffer more from the scheduling overheads as their memory phases are preempted more frequently.



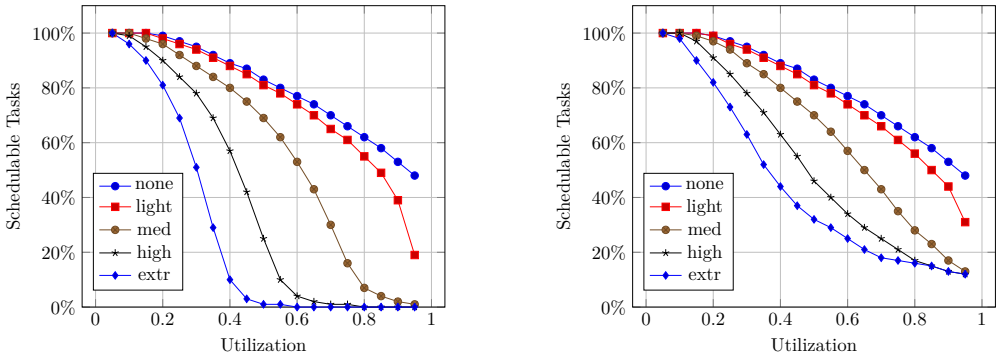
(a) Schedulability improvement.



(b) Response time reduction.

Figure 11 Percentage improvement in task schedulability and response time with respect to contention-based scheduling as a function of processor priority. The dashed lines take into account the measured memory arbitration overheads shown in Table 1.

Memory interference. We measure the impact of the task memory stall (i.e., the ratio between a task's memory phase duration and WCET - m_i/e_i) on the task schedulability for an $N = 4$ processor system. For each utilization within the range $[0.05, 1]$, we generate 5 types of sets with 4 tasks each. The types differ in the range of random stall distributions [61]: $[0.00, 0.00]$ (none), $[0.00, 0.05]$ (light), $[0.05, 0.20]$ (med), $[0.20, 0.40]$ (high) and $[0.40, 0.65]$ (extr). For instance, a task with a memory stall of 0.2 spends 20% of its WCET in the memory phase and 80% in the computation phase. Then, we take 4 sets with the same utilization and the same stall distribution, assign each set to one processor and test the schedulability with contention-based and fixed-priority memory-centric scheduling. Our test covers more than 550 randomly generated task sets per utilization and memory stall type. The percentage of schedulable tasks for contention-based scheduling and for fixed-priority scheduling are shown respectively in Figure 12(a) and Figure 12(b). Both algorithms can schedule only a very limited number of memory-intensive tasks. In fixed-priority scheduling, the tasks running on the highest priority processor are isolated from the interference generated by tasks running on the low priority processors. For this reason, the curves on Figure 12(b) converge to a higher value than the curves on Figure 12(a). In the next experiment, we show how grouping and prioritizing of high-rate memory-bound tasks can improve the overall schedulability of the system.



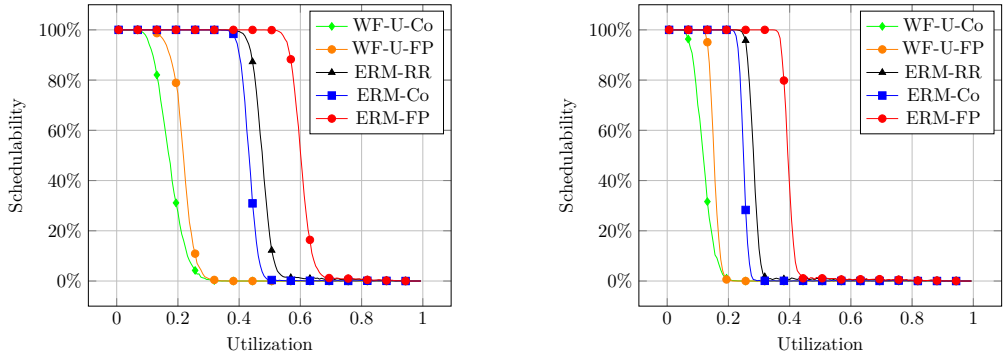
(a) Contention-based scheduling.

(b) Fixed-priority memory-centric scheduling.

Figure 12 Schedulability ratios for contention-based and fixed-priority memory-centric scheduling.

Heuristics for task assignment. We run our schedulability test for randomly generated task sets partitioned among the processors using various heuristics. We assume a multiprocessor system of $N = 16$ processors and a set of $N \cdot 8 = 128$ tasks. More than 5000 task sets are generated for each utilization factor within $[0, N]$ with a step size of 0.1. To partition the tasks among the processors, we consider many widely used heuristics and propose a new heuristic matching to our scheduling model. The standard heuristics include *First Fit (FF)*, *Next Fit (NF)*, and *Worst Fit (WF)*, each combined with an initialization step sorting the tasks in decreasing or increasing order by utilization factor or period, or without sorting. The best results among the standard heuristics are obtained for *WF* with tasks sorted in decreasing order of utilization. The heuristic that we propose first sorts the tasks in increasing order of their periods and then allocates the tasks using *FF* with the processors capacity reduced to U/N to ensure the proper load balancing (similarly to [63]). This heuristic clusters the tasks with shorter periods on the processors with higher priorities: the tasks do not suffer interference in memory phases from the low priority processors, and their blocking factor (due to the non-preemptive execution on the same processor) is not increased by the tasks that may have longer periods and longer worst-case execution times. The latter are

placed on the processors with lower priority since they can potentially accommodate more interference in their memory phases within the longer deadlines. The intuition is analogous to *Rate Monotonic* priority assignment.



(a) Normal task memory stall ($[0.05, 0.20]$). **(b)** High task memory stall ($[0.20, 0.40]$).

Figure 13 Schedulability ratio for different task partitioning heuristics.

In our evaluation, the fixed-priority policy performs best in every case, with the exception of the case where all policies fail for *FF* at $1/N$ due to the overcharge on the first processors. The results with the normalized utilization are shown in Figure 13. Due to space limitations, we report only *WF* with the utilization sorted in decreasing order (*WF-U*), which is the best among all the standard heuristics, and our Rate Monotonic like heuristic with equally distributed load (*ERM*) for round-robin (*RR*), contention-based (*Co*) and Fixed-Priority (*FP*) scheduling. The latter shows a maximal gain of 20% in the schedulability compared to the former two policies.

7 Related Work

Scratchpad-based memory-centric scheduling. Scratchpad-based systems allow overlapping of DMA memory transactions and CPU execution, resulting in better overall schedulability compared to cache-based systems. This approach was combined with partitioned [20, 48, 53, 57] and global scheduling [3]. Melani et al. [37] proposed exact response time analysis for fixed-priority scheduling on a single core with a fully preemptive DMA engine.

Cache-based memory-centric scheduling. The *PREM* was first introduced in [40, 41] to co-schedule memory accesses from CPU and I/O on a single core with multi-level caches. In [59], the *PREM* was applied for partitioned multiprocessor systems under TDM main memory accesses. To better utilize the assigned TDM-slot, the scheduling policy of each processor raises the priority of its pending memory phases above the priority of the computation phases. The concept of prioritizing the memory phases was also applied in global scheduling [60].

Schedulability analysis of global fixed-priority non-preemptive scheduling was extended for PREM compliant tasks [2, 33]. Non-preemptive execution permits to share the same cache partition among different tasks, but the scheduling algorithms proposed in the cited works are unaware of the tasks' cache footprints. Gaun et al. [21] integrated this constraint into the schedulability analysis for global scheduling. Nonetheless, without explicit cache locking, it is not sufficient to check that the tasks' data footprints are smaller than the available cache space to guarantee that their cache regions are not overlapping. In the partitioned scheduling considered in our work, the inter-core cache interference can be avoided by design with the proposed assignment of tasks to processors.

Matějka et al. [36] generate a static offline schedule for PREM-compliant tasks. The authors target COTS platforms with very similar characteristics to ours but recognize to disregard the inter-core eviction in L2 shared cache. The inter-core cache interference can be accounted for in the analysis proposed by Xiao et al. [58].

The closest related work was presented recently in [46]. The key differences to our work are that we do not consider hardware cache partitioning, we implement memory arbitration at the hypervisor-level, and our implementation, as well as schedulability analysis, are not limited to a single task per processor.

Cache coloring. Cache-coloring-based cache management frameworks for multiprocessor systems leverage virtualization [26, 32, 35, 38]. Our approach for cache partitioning is based on cache coloring implemented in *Jailhouse* hypervisor [20, 28]. Coloring can also be used for controlling DRAM bank allocation [64]. However, on platforms with address randomization (e.g., Nvidia TX1), this solution may be impracticable [36].

Time division multiplexing. Wandeler and Thiele [56] find an optimal TDM time slot and cycle assignment for systems characterized by arrival and service curves. Our model could be plugged into their framework by providing such curves. Hamann and Ernst [22] apply an evolutionary algorithm to optimize the time slot assignments of tasks mapped to TDM-scheduled resources. Worst-case execution time analysis can be extended to take the TDM parameters into account [47]. Several dynamic arbitration schemes based on TDM were recently proposed to avoid or reduce the resource contention in multiprocessor [1, 23] and network-on-chip [29] systems.

8 Conclusions, Limitations, Future Work

Conclusions. In this work, we proposed a memory-centric fixed-priority scheduler. Our solution does not rely on hardware features for predictable memory management of any kind, which is the case for many COTS multiprocessor platforms today. Furthermore, we were able to go beyond the static allocation of bus mastership as in the classical TDM approach, reducing the worst-case response time for high priority tasks. Our implementation shows that it is possible to achieve dynamic memory arbitration at the hypervisor level with relatively small overheads. Moreover, to the best of our knowledge, we are the first to propose a task-to-processor assignment algorithm integrated with the PREM.

Limitations. We have ignored any interference effects due to I/O devices using the shared memory hierarchy. This may not be of concern on COTS-based platforms that feature a dedicated I/O bus (e.g., MPC5777M, MPC5746M). Otherwise, the I/O interference should be taken into account into the worst-case memory phase time or other approaches for isolating I/O (e.g., [41]) or accelerators (e.g., GPU [10]) should be integrated. Since the OS and its tasks share the same cache partition, a minimal degree of cache pollution can occur. Additionally, the memory region of the hypervisor is not colored. We made the assumption that task data cannot exceed the size of the cache partition. Prior works [33,37,53,57,60] made a similar assumption. While the proposed implementation can already support multi-segment tasks, the schedulability analysis should be revisited.

Future work. As part of our ongoing work, we are investigating limited preemption for tasks assigned to the same processor and extending our model for three-phase PREM tasks.

References

1 A. Agrawal, G. Fohler, J. Freitag, J. Nowotsch, S. Uhrig, and M. Paulitsch. Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, pages 2:1–2:22, 2017.

2 A. Alhammad and R. Pellizzoni. Schedulability Analysis of Global Memory-predictable Scheduling. In *2014 Inter. Conference on Embedded Software (EMSOFT)*, pages 1–10, October 2014.

3 A. Alhammad, S. Wasly, and R. Pellizzoni. Memory Efficient Global Scheduling of Real-Time Tasks. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 285–296, April 2015.

4 ARM. Primecell Level 2 Cache Controller (PL310) - Technical Reference Manual, Revision: r2p0. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0246c/index.html`. Accessed: 2019-10-07.

5 S. Bak, G. Yao, R. Pellizzoni, and M. 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, August 2012.

6 M. Becker, D. Dasari, B. Nicolic, B. Akesson, V. Nélis, and T. Nolte. Contention-Free Execution of Automotive Applications on a Clustered Many-Core Platform. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 14–24, July 2016.

7 B. B. Brandenburg and M. Gül. Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 99–110, November 2016.

8 P. Burgio, A. Marongiu, P. Valente, and M. 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)*, pages 1–8, October 2015.

9 G. C. Buttazzo, M. Bertogna, and G. Yao. Limited Preemptive Scheduling for Real-Time Systems. A Survey. *IEEE Transactions on Industrial Informatics*, 9(1):3–15, February 2013.

10 N. Capodieci, R. Cavicchioli, P. Valente, and M. 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)*, pages 48–57, 2017.

11 J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. Audsley, R. Rajkumar, D. de Niz, and G. von der Brüggen. Many Suspensions, Many Problems: A Review of Self-Suspending Tasks in Real-Time Systems. *Real-Time Systems*, 55(1):144–207, January 2019.

12 C. Dall, S.W. Li, J. T. Lim, and J. Nieh. ARM Virtualization: Performance and Architectural Implications. *SIGOPS Oper. Syst. Rev.*, 52(1):45–56, August 2018.

13 R. I. Davis and A. Burns. A Survey of Hard Real-time Scheduling for Multiprocessor Systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, October 2011.

14 R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien. Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, April 2007.

15 G. Durrieu, M. Faugère, S. Girbal, D. Gracia Pérez, C. Pagetti, and W. Puffitsch. Predictable Flight Management System Implementation on a Multicore Processor. In *Embedded Real Time Software (ERTS)*, February 2014.

16 P. Emberson, R. Stafford, and R.I. Davis. Techniques For The Synthesis Of Multiprocessor Tasksets. In *WATERS at the Euromicro Conference on Real-Time Systems*, pages 6–11, July 2010.

17 Evidence. Erika Enterprise RTOS v3, October 2018. Accessed: 2019-10-16. URL: `http://www.erika-enterprise.com/`.

18 H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R.B. Sørensen, P. Wägemann, and S. Wegener. TACLeBench: A Benchmark Collection to

Support Worst-Case Execution Time Research. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET)*, pages 2:1–2:10, 2016.

19 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):32:1–32:36, November 2015.

20 G. Gracioli, R. Tabish, R. Mancuso, R. Mirosanlou, R. Pellizzoni, and M. Caccamo. Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. In *31st Euromicro Conference on Real-Time Systems (ECRTS)*, pages 27:1–27:25, 2019.

21 N. Guan, M. Stigge, W. Yi, and G. Yu. Cache-aware Scheduling and Analysis for Multicores. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, pages 245–254, 2009.

22 A. Hamann and R. Ernst. TDMA Time Slot and Turn Optimization with Evolutionary Search Techniques. In *Design, Automation and Test in Europe*, pages 312–317, March 2005.

23 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.

24 R. E. Kessler and M. D. Hill. Page Placement Algorithms for Large Real-indexed Caches. *ACM Trans. Comput. Syst.*, 10(4):338–359, November 1992.

25 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.

26 H. Kim and R. Rajkumar. Real-Time Cache Management for Multi-Core Virtualization. In *2016 International Conference on Embedded Software (EMSOFT)*, pages 1–10, October 2016.

27 J. Kiszka, V. Sinitsyn, H. Schild, and contributors. Jailhouse Hypervisor. Siemens AG on GitHub, `https://github.com/siemens/jailhouse`, 2018. Accessed: 2019-10-10.

28 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 *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–14, April 2019.

29 A. Kostrzewa, S. Saidi, L. Ecco, and R. Ernst. Flexible TDM-Based Resource Management in on-Chip Networks. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems (RTNS)*, page 151–160, 2015.

30 J. P. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 201–209, December 1990.

31 J. Liedtke, H. Haertig, and M. Hohmuth. OS-Controlled Cache Predictability for Real-Time Systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS)*, pages 213–, 1997.

32 R. Ma, W. Ye, A. Liang, H. Guan, and J. Li. Cache Isolation for Virtualization of Mixed General-purpose and Real-time Systems. *J. Syst. Archit.*, 59(10):1405–1413, November 2013.

33 C. Maia, G. Nelissen, L. Nogueira, L. M. Pinho, and D. G. Pérez. Schedulability Analysis for Global Fixed-Priority Scheduling of the 3-Phase Task Model. In *IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10, August 2017.

34 R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. 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, April 2013.

35 J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto. Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems. In *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*, pages 3:1–3:14, 2020.

36 J. Matějka, B. Forsberg, M. Sojka, Z. Hanzálek, L. Benini, and A. Marongiu. Combining PREM Compilation and ILP Scheduling for High-performance and Predictable MPSoC Execution. In

Proc. of the 9th Inter. Workshop on Programming Models and Applications for Multicores and Manycores (PMAM), pages 11–20, 2018.

37 A. Melani, M. Bertogna, R. I. Davis, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo. Exact Response Time Analysis for Fixed Priority Memory-Processor Co-Scheduling. *IEEE Transactions on Computers*, 66(4):631–646, April 2017.

38 P. Modica, A. Biondi, G. Buttazzo, and A. Patel. Supporting Temporal and Spatial Isolation in a Hypervisor for ARM Multicore Platforms. In *2018 IEEE International Conference on Industrial Technology (ICIT)*, pages 1651–1657, February 2018.

39 M. Nasri, G. Nelissen, and B. B. Brandenburg. A Response-Time Analysis for Non-Preemptive Job Sets under Global Scheduling. In *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, pages 9:1–9:23, 2018.

40 R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, and M. Caccamo. Predictable Execution Model: Concept and Implementation. Technical report, University of Illinois at Urbana-Champaign, June 2010. URL: `http://hdl.handle.net/2142/16605`.

41 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 *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279, April 2011.

42 L. T. X. Phan, M. Xu, and I. Lee. Cache-aware Interfaces for Compositional Real-time Systems: Invited Paper. *SIGBED Rev.*, 13(3):52–55, August 2016.

43 I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Design, Automation and Test in Europe Conference and Exposition (DATE)*, pages 1484–1489, 2007.

44 R. Rajkumar. Real-Time Synchronization Protocols for Shared Memory Multiprocessors. In *Proceedings.,10th International Conference on Distributed Computing Systems*, pages 116–123, May 1990.

45 R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer. Look Mum, no VM Exits! (Almost). *Proceedings of the 13th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, May 2017.

46 J. M. Rivas, J. Goossens, X. Poczekajlo, and A. Paolillo. Implementation of Memory Centric Scheduling for COTS Multi-Core Real-Time Systems. In *31st Euromicro Conference on Real-Time Systems (ECRTS)*, pages 7:1–7:23, 2019.

47 J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In *28th IEEE International Real-Time Systems Symposium (RTSS)*, pages 49–60, December 2007.

48 B. Rouxel, S. Skalistis, S. Derrien, and I. Puaut. Hiding Communication Delays in Contention-Free Execution for SPM-Based Multi-Core Architectures. In *31st Euromicro Conference on Real-Time Systems (ECRTS)*, pages 25:1–25:24, 2019.

49 M. Schoeberl, L. Pezzarossa, and J. Sparsø. A Multicore Processor for Time-Critical Applications. *IEEE Design Test*, 35(2):38–47, April 2018.

50 L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

51 V. Shivappa. x86: Intel Cache Allocation Technology support. `https://lwn.net/Articles/622893/`. Accessed: 2019-10-07.

52 M. R. Soliman, G. Gracioli, R. Tabish, R. Pellizzoni, and M. Caccamo. Segment Streaming for the Three-Phase Execution Model: Design and Implementation. In *IEEE Real-Time Systems Symposium (RTSS)*, December 2019.

53 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.

54 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.

55 K. Tindell and J. Clark. Holistic Schedulability Analysis for Distributed Hard Real-time Systems. *Microprocess. Microprogram.*, 40(2-3):117–134, April 1994.

56 E. Wandeler and L. Thiele. Optimal TDMA Time Slot and Cycle Length Allocation for Hard Real-Time Systems. In *Asia and South Pacific Conference on Design Automation, 2006.*, pages 6 pp.–, January 2006.

57 S. Wasly and R. Pellizzoni. Hiding Memory Latency Using Fixed Priority Scheduling. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 75–86, April 2014.

58 J. Xiao, S. Altmeyer, and A. Pimentel. Schedulability Analysis of Non-preemptive Real-Time Scheduling for Multicore Processors with Shared Caches. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 199–208, December 2017.

59 G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems*, 48(6):681–715, November 2012.

60 G. Yao, R. Pellizzoni, S. Bak, H. Yun, and M. Caccamo. Global Real-Time Memory-Centric Scheduling for Multicore Systems. *IEEE Trans. on Computers*, 65(9):2739–2751, September 2016.

61 G. Yao, H. Yun, Z. P. Wu, R. Pellizzoni, M. Caccamo, and L. Sha. Schedulability Analysis for Memory Bandwidth Regulated Multicore Real-Time Systems. *IEEE Transactions on Computers*, 65(2):601–614, February 2016.

62 Y. Ye, R. West, Z. Cheng, and Y. Li. COLORIS: A Dynamic Cache Partitioning System Using Page Coloring. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 381–392, August 2014.

63 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.

64 H. Yun, R. Mancuso, Z.P. Wu, and R. Pellizzoni. PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, April 2014.

CPU Energy-Aware Parallel Real-Time Scheduling

Abusayeed Saifullah
Wayne State University, Detroit, MI, USA
saifullah@wayne.edu

Sezana Fahmida
Wayne State University, Detroit, MI, USA
fahmida.sezana@wayne.edu

Venkata P. Modekurthy
Wayne State University, Detroit, MI, USA
modekurthy@wayne.edu

Nathan Fisher
Wayne State University, Detroit, MI, USA
fishern@wayne.edu

Zhishan Guo
University of Central Florida, Orlando, FL, USA
zsguo@ucf.edu

Abstract

Both energy-efficiency and real-time performance are critical requirements in many embedded systems applications such as self-driving car, robotic system, disaster response, and security/safety control. These systems entail a myriad of real-time tasks, where each task itself is a parallel task that can utilize multiple computing units at the same time. Driven by the increasing demand for parallel tasks, multi-core embedded processors are inevitably evolving to many-core. Existing work on real-time parallel tasks mostly focused on real-time scheduling without addressing energy consumption. In this paper, we address hard real-time scheduling of parallel tasks while minimizing their CPU energy consumption on multicore embedded systems. Each task is represented as a directed acyclic graph (DAG) with nodes indicating different threads of execution and edges indicating their dependencies. Our technique is to determine the execution speeds of the nodes of the DAGs to minimize the overall energy consumption while meeting all task deadlines. It incorporates a frequency optimization engine and the dynamic voltage and frequency scaling (DVFS) scheme into the classical real-time scheduling policies (both federated and global) and makes them energy-aware. The contributions of this paper thus include the first energy-aware online federated scheduling and also the first energy-aware global scheduling of DAGs. Evaluation using synthetic workload through simulation shows that our energy-aware real-time scheduling policies can achieve up to 68% energy-saving compared to classical (energy-unaware) policies. We have also performed a proof of concept system evaluation using physical hardware demonstrating the energy efficiency through our proposed approach.

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

Keywords and phrases Real-time scheduling, multicore, energy-efficiency, embedded systems

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.2

Funding This work was supported by National Science Foundation through grants CAREER-1846126, CNS-1618185, IIS-1724227, and CNS-1850851, by Wayne State University through Rumble Fellowship, and by University of Central Florida through a startup grant.

1 Introduction

Energy-efficiency is an important requirement in embedded systems (e.g., mobile phones, tablets, cars, robots, and computerized numerical controls) as they rely on limited or unreliable sources of energy such as batteries or energy harvesters. Embedded systems are used in all aspects of human life and industries and are now being sparked by billions of

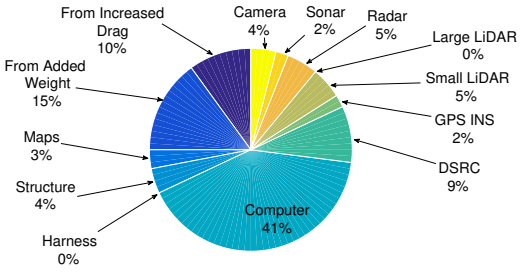


Figure 1 Energy consumption of the autonomy system in a self-driving car (Ford Fusion)[51].

devices through the evolution of Internet of Things. Energy consumption by these billions of devices can be significant. It is projected that computers and cell phones will consume 14% of worldwide power by 2020 [1]. A recent study using the Ford Fusion autonomy system has revealed that 41% energy is consumed by the computing platform of a self-driving car (Figure 1) [51, 66, 29]. It is estimated that up to 80% of the total energy consumption of an embedded system is due to software-related activities [50]. In such systems, a processor consumes a significant share of their power consumption. For example, in spacecraft, the RAD750 processor draws almost 33%-50% share of the power consumption [2]. Real-time performance is another critical requirement in many embedded systems applications such as self-driving car, advanced robotic system, disaster response, and surveillance systems. Many involve mission critical applications that require a predictable real-time system behavior but battery re-charging during the mission may not be possible.

With the evolution of various computation-intensive systems (e.g., cloud computing, self-driving car), today's real-time systems evolve in the form of many parallel tasks. For example, a self-driving car [39] entails a myriad of real-time tasks such as motion planning, sensor fusion, computer vision, and decision making system that exhibit *intra-task parallelism*, where each task itself can utilize multiple computing units simultaneously. For example, the decision making system collects massive amounts of data from various types of sensors and processes them in parallel. Driven by the increasing demand for parallel tasks, multicore embedded processors are inevitably evolving to many-core (e.g., Intel's 48-core SCC chip [3], TILERA's 100-core TILE-Gx100 [7], 248-core PC205 of picoChip [5]). Multicore offers opportunities for energy minimization. For example, the energy consumption of executing a certain workload equally distributed in two cores is significantly less than that of executing it in one core at double speed [55]. As the energy-related benefits resulting from Moore's law are leveling off, software-level techniques need to be exploited to reduce the power consumption on multicores. While real-time scheduling for parallel tasks on multicores has been widely studied recently, existing work mostly focused on scheduling *without* addressing energy consumption [13, 14, 61, 28, 63, 62, 12, 44, 43, 19, 38, 8, 40, 53].

In this paper, we address hard real-time scheduling of parallel tasks while minimizing their CPU energy consumption on multicore embedded systems. We consider periodic parallel tasks with deadlines, where each task is represented by a directed acyclic graph (DAG). In a DAG, the *nodes* stand for different threads of execution while the *edges* represent their dependencies. DAG is the most general model of deterministic parallel tasks [62]. Although our proposed techniques can be applied to any multicore platform for reduced energy consumption, we focus on embedded systems as their energy efficiency is of more general need. Energy-aware real-time scheduling is challenging in general due to the complicated (nonlinear) relationship between frequency, energy consumption, and execution time of a task.

Although much work exists on energy-aware real-time scheduling on multiprocessor [56, 55, 52, 18, 37, 22, 20, 60, 10, 72, 21, 24, 46, 47, 42, 65, 68], it has considered only sequential task models. There is a recent study on energy-aware parallel real-time scheduling that considered overly simplified federated scheduling of DAGs where at any time *only one task* can run and the number of cores *cannot be pre-fixed* as an input [34, 17]. It adopted a table-driven scheduling where the entire schedule (up to the hyper-period which is of *exponential size* in task periods) needs to be created in-advance. In contrast, we consider scheduling many recurrent tasks exploiting both intra- and inter-task parallelism on a finite set of cores. We adopt *online* scheduling so that tasks can be scheduled as they arrive. We consider both *global* and *federated* scheduling. For global scheduling, we consider *both* dynamic priority and fixed priority scheduling.

In this paper, we focus on minimizing CPU energy consumption by means of *dynamic voltage and frequency scaling (DVFS)*. DVFS is a commonly-used power-management technique where the clock frequency of a processor is decreased to allow a reduction in the supply voltage. This reduces power consumption, which leads to reduction in the energy required for a computation. Many AMD and Intel processors support per-core DVFS for flexible power control (e.g., AMD family 10h processors, Intel Haswell processors) [31]. Our approach is to regulate the frequencies across the cores and across different execution parts of the same task for minimizing overall energy consumption. It incorporates an optimization engine and DVFS into the traditional real-time scheduling policies (e.g., earliest deadline first, deadline monotonic, and federated scheduling) and makes them energy-aware. Specifically, once the optimal execution speeds of the nodes of all DAGs are determined, we adopt these classical real-time scheduling policies. Note that these classical policies are online, highly efficient in real-time performance, but are not energy-aware. In our approach, when the processor frequencies are adjusted based on the nodes' speed assignments, the adopted real-time scheduling policy leads to energy minimization.

Specifically, we make the following new contributions.

- For federated scheduling, the objective becomes to assign a certain number of cores to each task and execution speeds to different nodes so that overall energy consumption is minimized. We first formulate this as a constrained non-linear optimization problem whose solution becomes challenging due to the discrete nature of solution space. We propose a continuous convexification approach through a new core allocation scheme in federated scheduling while retaining its theoretical real-time performance bound. This is the **first** energy-aware federated scheduling of multiple tasks which does not rely on infinite number of cores or table-driven scheduling.
- We formulate energy-aware real-time global scheduling of DAGs as a convex optimization problem for both dynamic and fixed priority. The objective is to determine the execution speeds of the DAG nodes to minimize overall energy consumption while meeting all deadlines. An optimal solution is achieved in polynomial time. Upon adjusting the processor frequencies online, the overall energy consumption is minimized under classical real-time scheduling policies. This is the **first** energy-aware global scheduling of DAGs.
- We have evaluated our energy-aware real-time scheduling policies using synthetic workloads through simulations. The results show that our approach can achieve up to 68% energy-saving compared to classical (energy-unaware) policies. We have also performed a proof of concept system evaluation using multiple ODROID XU-4 boards [54] demonstrating the energy efficiency through our proposed approach.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 describes our parallel tasks, multicore, and energy consumption model and some background. Section 4 presents our energy-aware federated scheduling. Section 5 presents our energy-aware

global scheduling. Section 6 presents simulation results. Section 7 presents the proof of concept system evaluation. Section 8 concludes our paper with future research directions.

2 Related Work

While the works proposed in [56, 55, 52, 18, 37, 22, 20, 60, 10, 72, 21, 24, 46, 47, 42, 65, 68] studied energy-aware real-time scheduling on multiprocessor, they considered sequential tasks. A detailed review of these works can be found in a recent survey in [11]. While a parallel task can execute on multiple cores simultaneously, a *sequential task* can execute only on one core at a time. Therefore, parallel tasks scheduling is significantly different from sequential tasks. Parallel scheduling of DAGs is highly challenging as the dependencies among the nodes need to be considered while these are absent in sequential tasks.

Existing work on parallel real-time scheduling concentrates mostly on scheduling policies or analysis and has not focused on energy-consumption [13, 14, 61, 28, 63, 62, 12, 44, 43, 19, 38, 8, 40, 53, 67, 16]. Energy-aware scheduling of parallel tasks on multicore was studied in [45] for a very simplified model where a task has a fixed number of parallel threads, and the tasks are not recurrent or real-time [45]. The energy benefit of multicore scheduling was studied in [58, 41, 71, 32, 58] by running parallel threads. Energy-aware gang scheduling was studied in [57]. Gang scheduling involves a very special form of parallelism where all parallel threads of the same task use processors in the same window (i.e., they start and stop using the processors at the exact same time). A slack stealing based scheduling was studied in [74, 75] for energy minimization of an application consisting of inter-dependent sequential tasks. While those dependencies among the tasks were represented by a DAG, the model consists of a single DAG and does not consider recurrent tasks. A similar model was studied in [70, 69] for non-real-time power aware cluster computing. We consider energy-aware real-time scheduling of multiple/many recurrent DAGs.

An energy-efficient clustering of parallel tasks was studied in [33, 15]. It is only suitable for clustered multicore platform where all tasks assigned to the same cluster of cores have to run at the same speed (as all cores in the same cluster run at the same speed). Recently, a simplified model of energy-aware federated real-time scheduling of DAGs was studied where at any time **only one task can run** and the number of cores **cannot be pre-fixed** as an input [34, 17]. It adopted a **table-driven scheduling**, where the entire schedule (up to the hyper-period which is of **exponential size** in task periods) is created in-advance. *In contrast, this paper* **(1)** *considers scheduling multiple/many recurrent DAGs exploiting both intra- and inter-task parallelism on a given finite set of cores;* **(2)** *adopts online scheduling so that tasks can be scheduled as they arrive;* **(3)** *considers both global and federated scheduling with guaranteed performance bounds of the algorithms;* **(4)** *considers both dynamic priority and fixed priority under global scheduling.*

3 System Model and Background

3.1 Parallel Task Model

We consider a task set $\tau = \{\tau_1, \tau_2, \cdots, \tau_n\}$ of n periodic parallel tasks to be scheduled on a multicore platform consisting of m cores. Each task $\tau_i, 1 \leq i \leq n$, is represented as a DAG, where a *node* is a thread of execution (execution requirement), and the *edges* between the nodes indicate the dependencies between the associated threads of execution.

A node in τ_i is denoted by $W_i^j, 1 \leq j \leq n_i$, where n_i is the total number of nodes in τ_i. The *worst-case execution requirement (WCER)* (i.e. *the total number of CPU cycles needed in the worst-case*) of node W_i^j is denoted by c_i^j. This means that the node has *worst-case*

execution time (WCET) of c_i^j on a unit-speed processor. Note that the actual execution requirement (actual number of execution cycles) of a node W_i^j can vary in practice due to many complex factors such as data inputs, hardware features, and execution contexts but it will never exceed its WCER c_i^j. A directed edge from node W_i^j to node W_i^k, denoted as $W_i^j \rightarrow W_i^k$, implies that, for each DAG job released by task τ_i, the execution of W_i^k cannot start until W_i^j finishes. W_i^j, in this case, is called a *parent* of W_i^k, while W_i^k is its *child*. A node may have 0 or more parents or children, and can start execution only after all of its parents have finished execution. A node is *ready* to be executed as soon as all of its parents have been executed. A node having no parent is called a *source* while a node having no child is a *sink*. A DAG may have multiple sources or sinks.

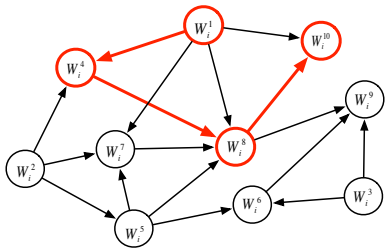


Figure 2 A parallel task τ_i represented as a DAG of 10 nodes: nodes W_i^1, W_i^2, and W_i^3 are the sources while W_i^9 and W_i^{10} are the sinks. Suppose for example $c_i^1 = 4$, $c_i^2 = 2$, $c_i^3 = 4$, $c_i^4 = 5$, $c_i^5 = 3$, $c_i^6 = 4$, $c_i^7 = 2$, $c_i^8 = 4$, $c_i^9 = 1$, $c_i^{10} = 1$. Then, $C_i = 30$. $L_i = 14$, and the $W_i^1 \rightarrow W_i^4 \rightarrow W_i^8 \rightarrow W_i^{10}$ (thick red-colored) is a critical path on unit-speed cores.

The WCER (i.e., *worst-case work*) C_i of task τ_i is the sum of the WCERs of all nodes in τ_i; that is, $C_i = \sum_{j=1}^{n_i} c_i^j$. Thus, C_i is the WCET of τ_i if it was executing on a single processor of speed 1. For task τ_i, the *critical path length*, denoted by L_i, is the sum of execution times of the nodes on a critical path. A *critical path* is a directed path that has the maximum execution time among all other paths in DAG τ_i. Thus, L_i is the *minimum execution time* of τ_i even when it is assigned an infinite number of unit-speed cores. Figure 2 illustrates a parallel task τ_i represented as a DAG with $n_i = 10$ nodes.

The *period* of task τ_i is denoted by T_i. Every instance of a task is called a *job*. Each task τ_i thus releases an infinite sequence of jobs, with T_i being the time separation between two consecutive jobs. The (relative) deadline D_i of each task τ_i is considered *implicit*, i.e., $D_i = T_i$. That is, each job of a task must finish before its next job releases. Since L_i is the minimum execution time of task τ_i even on a machine with an infinite number of cores, the condition $T_i \geq L_i$ must hold for τ_i to be schedulable (i.e. to meet its deadline). A task set is *schedulable* when all tasks in the set meet their deadlines.

The *utilization* u_i of a task τ_i, and the *total utilization* $u_{\text{sum}}(\tau)$ for task set τ of n tasks are defined as

$$u_i = \frac{C_i}{T_i}; \qquad u_{\text{sum}}(\tau) = \sum_{i=1}^{n} u_i = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

If the total utilization u_{sum} is greater than m, then no algorithm can schedule τ on m identical unit-speed processor cores.

3.2 Power/Energy Model

We consider the following widely used energy consumption model of processor [10, 73, 72, 27, 55, 56, 34]. Assuming continuous frequency scheme, let $s(t)$ denote the main frequency (speed) of a processor at time t. Then its power consumption $P(s)$ can be modeled as:

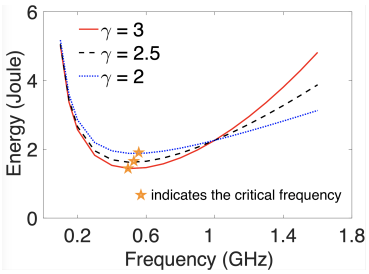
$$P(s) = P_s + P_d(s) = \beta + \alpha s^\gamma, \tag{1}$$

where P_s denotes the *static power consumption* which is introduced due to the leakage current and $P_d(s)$ is the *active power consumption*. $P_d(s)$ is introduced due to capacitor charging and discharging during processor activity, and it depends on the processor frequency. $P_d(s)$ can be represented as αs^γ where the constant $\alpha > 0$ depends on the effective switching capacitance [55], $\gamma \in [2, 3]$ is a fixed parameter determined by the hardware, and $\beta > 0$ represents the leakage power (i.e., the static part of power consumption whenever a processor remains on or idle). Power consumption is a convex-increasing function of the processor frequency. It is possible to reduce $P_d(s)$ by reducing the processor frequency through DVFS.

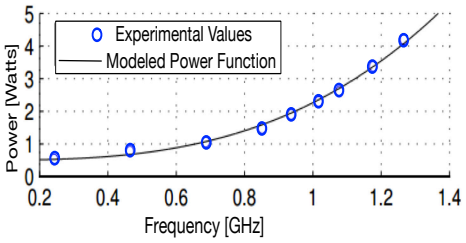
The energy consumption in any given period $[t_1, t_2]$ can be calculated as $E = \int_{t_1}^{t_2} P(s)\, dt$, which is almost close to the actual CPU energy consumption of many known systems. Specifically, given a fixed amount of workload C to be executed on a speed-s processor, the total energy consumption is the integral of power over the period of length C/s; i.e.,

$$E(C, s) = (\beta + \alpha s^\gamma)(C/s) = \beta C/s + \alpha C s^{\gamma-1} \tag{2}$$

Figure 3a illustrates how different values of γ and processor speed s may affect the total energy consumption to complete a certain amount of computation. In most modern processors, execution at a frequency much lower than the *critical frequencies* [55] (the highlighted most energy efficient speed in the figure) is energy inefficient as leakage power becomes the major "contribution". This model was shown to be quite realistic [56]. Figure 3b compares the original power consumption and the power model in Equation (1) as studied in [35].



(a) Energy consumption for executing a job with 10^9 computation cycles for various γ, where $\alpha = 1.76\, Watts/GHz^\gamma$, $\beta = 0.5$ $Watts$.



(b) Comparison of the power model (Eq. (2)) with experimental results in [35]. Here, $\alpha = 1.76\, Watts/GHz^3$, $\gamma = 3$, and $\beta = 0.5$ Watts.

Figure 3 CPU energy consumption model.

While we consider a continuous frequency scaling of the processors, our approach is applicable to the systems with discrete frequency levels as well. Specifically, any frequency in our continuous frequency scheme can be rounded up to achieve the discrete frequency level. In fact, most current processors have quite fine-grained steps to scale frequency. For example, the ODROID XU board (used in our system experiment) has a frequency range of 0.2 – 1.4GHz for LITTLE cores and 0.2 – 2GHz for big cores, with a scale step of 0.1GHz. With such fine-grained steps, the energy consumption modeled for continuous frequency scheme becomes very close to actual scenario for modern discrete frequency processors.

3.3 Global and Federated Scheduling Overview

We consider *preemptive scheduling*, where a task with higher priority can preempt an executing task that has lower priority. Non-preemptive scheduling is less preferred for time critical systems (partially) because of its poor responsiveness due to blocking and will not be explored in this paper. We shall develop both federated and global preemptive real-time scheduling of parallel tasks for energy minimization. Under global scheduling, we shall consider classical real-time scheduling policies such as *Earliest Deadline First (EDF)* and *Deadline Monotonic (DM)*, and make them energy-aware. For federated scheduling, we shall consider the policy introduced in [44] which is described below.

Federated scheduling was first introduced in [44] as a generalized approach of partitioned scheduling for parallel tasks which has later been widely used in the literature for real-time parallel scheduling. In ***federated scheduling***, a task is classified as a ***high-utilization task*** if its utilization $\geq$1, or as a ***low-utilization task*** otherwise. Each high-utilization task is allocated a dedicated cluster (set) of cores. A multiprocessor scheduling algorithm is used to schedule all low-utilization tasks (each having utilization <1), each of which is run sequentially, on a shared cluster composed of the remaining cores. Given a task set τ, the federated scheduling algorithm works as follows: First, tasks are divided into two disjoint sets: τ_{high} contains all high-utilization tasks – tasks with worst-case utilization at least one ($u_i \geq 1$), and τ_{low} contains all the remaining low-utilization tasks. Consider a high-utilization task τ_i with WCER C_i, worst-case critical-path length L_i, and deadline D_i ($= T_i$). We assign m_i dedicated cores to τ_i where

$$m_i = \left\lceil \frac{C_i - L_i}{T_i - L_i} \right\rceil .$$

We use $m_{\text{high}} = \sum_{\tau_i \in \tau_{\text{high}}} m_i$ to denote the total number of cores assigned to high-utilization tasks τ_{high}. We assign the remaining cores to all low-utilization tasks τ_{low}, denoted as $m_{\text{low}} = m - m_{\text{high}}$ The scheduling algorithm admits the task set τ, if $m_{\text{low}} \geq 2 \sum_{\tau_i \in \tau_{\text{low}}} u_i$.

After a valid core allocation, runtime scheduling proceeds as follows: (i) Any greedy (work-conserving) parallel scheduler can be used to schedule a high-utilization task τ_i on its assigned m_i cores. Informally, a **greedy scheduler** is one that never keeps a core idle if some node is ready to execute. (ii) Low-utilization tasks are treated and executed as though they are sequential tasks and any multiprocessor scheduling algorithm (such as partitioned EDF [48], or various rate-monotonic schedulers [9]) with a utilization bound of at most $\frac{1}{2}$ can be used to schedule them on m_{low} cores.

4 Energy-Aware Federated Scheduling

We first describe our approach for federated scheduling of real-time parallel tasks for minimizing their energy consumption on multicore. Since energy consumption of the cores is a function of their frequencies (speeds), our approach is to regulate the frequencies (speed) across the cores and across different nodes of the same task for minimizing overall energy consumption. For federated scheduling, we also have to determine a cluster (set) of cores on which a task will execute. We design our algorithms considering WCER of the nodes considering hard real-time systems that need both real-time guarantee and energy efficiency. Considering WCER, a node is executed at a uniform speed but different nodes can run at different speeds in our approach. Our approach incorporates an optimization engine and DVFS into the federated scheduling. Specifically, we first determine a cluster (set) of cores on which a task will execute and assign execution speeds to different nodes of all tasks. We

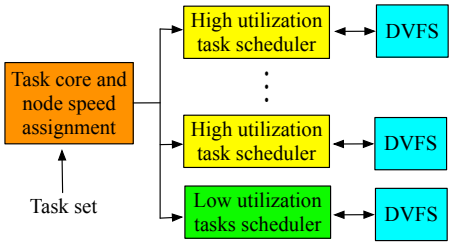


Figure 4 Energy-aware federated scheduling framework.

then incorporate DVFS into each cluster that will adjust the processor frequency during scheduling according to the assigned speeds of the nodes. Once the execution speeds of the nodes and the number of cores of all DAGs are determined, and the processor frequencies are adjusted based on the nodes' speed assignments, the adopted scheduling policy leads to energy minimization. Our federated scheduling uses the policy presented in Section 3.3 and makes it energy-aware. The scheduling is done after assigning the node execution speeds (i.e. the frequencies at which the cores will run when executing the nodes) and the number of cores of each high utilization task and for the low-utilization ones. Figure 4 shows our energy-aware federated scheduling framework.

4.1 Frequency and Core Assignment Formulation

Our objective is to assign cores to each DAG and determine execution speeds for different nodes of the DAG to minimize overall energy consumption. Such a speed and core assignment must guarantee that all DAGs remain schedulable when the federated policy is applied for their scheduling. Hence, we apply existing highly efficient schedulability conditions as constraints based on processor **capacity augmentation** analysis [44] for DAGs on multicore. Considering overall energy consumption as the objective and using existing schedulability conditions based on processor capacity augmentation analysis, we formulate energy-aware real-time federated scheduling of DAGs as a constrained non-linear optimization problem.

▶ **Definition 1.** *[44] A scheduling algorithm* $\mathbb{S}$ *with* ***capacity augmentation bound*** b*,* $b \geq 1$*, can always schedule a task set* τ *with total utilization of* u_{sum} *on* m *cores of speed* b *as long as* τ *satisfies two conditions on unit speed cores: (1)* $u_{sum} \leq m$*; and (2)* $L_i \leq D_i, \forall \tau_i$*.*

To formulate the problem, we first derive an expression to represent our objective (overall energy consumption) based on the energy model. By Equation (2), the total energy consumption for running node W_i^j at speed s_i^j becomes

$$E(c_i^j, s_i^j) = \left(\beta + \alpha (s_i^j)^\gamma\right)(c_i^j / s_i^j) = \beta c_i^j / s_i^j + \alpha c_i^j (s_i^j)^{\gamma - 1}$$

Thus total energy consumption by one job of task τ_i becomes $\sum_{j=1}^{n_i} E(c_i^j, s_i^j)$. To consider overall energy-consumption, we consider a complete schedule up to the hyper-period, denoted by H, of the tasks. Thus the overall energy consumption by task set τ is given by

$$\sum_{i=1}^{n} w_i \sum_{j=1}^{n_i} E(c_i^j, s_i^j), \quad \text{where } w_i = \frac{H}{T_i}$$

which becomes our objective (4). The actual energy consumption is also affected by a number of other issues such as frequency switching, turn on/off overhead (see Section 4.4) that are not considered in the above objective. As shown before in Figure 3b, this objective can

still represent a close approximation of the actual energy consumption of CPU. We want to minimize this objective while ensuring the schedulability under federated scheduling policy. Now we determine the constraints that must guarantee the schedulability under federated scheduling. The following remark plays a key role in establishing our constraints.

▶ Remark 2. A scheduler $\mathbb{S}$ has a **capacity augmentation bound** of b if it can schedule any task set τ on m cores which satisfies the following two conditions [44]:

- **Condition 1:** The total utilization of τ is at most $\frac{m}{b}$.
- **Condition 2:** For each task $\tau_i \in \tau$ the worst-case critical-path length L_i (execution time of τ_i on an infinite number of cores) is at most $\frac{1}{b}$ fraction of its deadline.

The federated scheduling that we consider (described in Section 3.3) has a capacity augmentation bound of 2 [44]. Using this bound, we have to incorporate the two conditions of Remark (2) as constraints. When node W_i^j of task τ_i is assigned speed s_i^j, where $1 \leq j \leq n_i, 1 \leq i \leq n$, the total utilization can be expressed as $\sum_{i=1}^{n}(\sum_{j=1}^{n_i} \frac{c_i^j}{s_i^j})/T_i$. Thus, the above **Condition 1** for schedulability is expressed as constraint (5) in our formulation.

Assigning different speeds to different nodes may change a task's critical path. Let $\Phi_i(s_i)$ be the set of nodes on a critical path if the nodes of τ_i are assigned speeds $s_i = \{s_i^1, s_i^2, \cdots, s_i^j, \cdots, s_i^{n_i}\}$. The critical path length $L_i(s_i)$ under this speed is given by

$$L_i(s_i) = \sum_{W_i^j \in \Phi_i(s_i)} \frac{c_i^j}{s_i^j} \tag{3}$$

Hence, **Condition 2** for schedulability is expressed as constraint (6) in our formulation.

Since assigning various speeds affects WCET and L_i, it affects the number of cores m_i assigned to each high utilization task τ_i and the total cores m_{low} assigned to all low utilization tasks. Hence, in addition to the constraints defined above, we have to maintain the following constraint for each speed assignment in federated scheduling as given in Section 3.3.

$$\begin{aligned}
& m_{\text{low}} = m - m_{\text{high}} \geq 2 \sum_{\tau_i \in \tau_{\text{low}}} u_i \\
\Leftrightarrow\ & m_{\text{high}} + 2 \sum_{\tau_i \in \tau_{\text{low}}} u_i \leq m \ \Leftrightarrow \sum_{\tau_i \in \tau_{\text{high}}} m_i + 2 \sum_{\tau_i \in \tau_{\text{low}}} u_i \leq m \\
\Leftrightarrow\ & \sum_{\tau_i \in \tau_{\text{high}}} \left\lceil \frac{C_i - L_i}{T_i - L_i} \right\rceil + 2 \sum_{\tau_i \in \tau_{\text{low}}} \frac{C_i}{T_i} \leq m \\
\Leftrightarrow\ & \sum_{\forall \tau_i \text{ with } \frac{C_i}{T_i} \geq 1} \left\lceil \frac{C_i - L_i}{T_i - L_i} \right\rceil + 2 \sum_{\forall \tau_i \text{ with } \frac{C_i}{T_i} < 1} \frac{C_i}{T_i} \leq m
\end{aligned}$$

Considering node W_i^j of task τ_i is assigned speed s_i^j, where $1 \leq j \leq n_i, 1 \leq i \leq n$, we write the above condition as constraint (7) in our formulation. Thus our objective is to determine speeds $s_i = \{s_i^1, s_i^2, \cdots, s_i^j, \cdots, s_i^{n_i}\}, \forall \tau_i$ to

$$\text{Minimize} \quad \sum_{i=1}^{n} w_i \sum_{j=1}^{n_i} E(c_i^j, s_i^j) \tag{4}$$

$$\text{subject to} \quad \sum_{i=1}^{n} \frac{\sum_{j=1}^{n_i} \frac{c_i^j}{s_i^j}}{T_i} \leq \frac{m}{2} \tag{5}$$

$$L_i(s_i) \leq \frac{T_i}{2}, \forall \tau_i \tag{6}$$

$$\sum_{\forall \tau_i \text{ with } \sum_{j=1}^{n_i} \frac{c_i^j}{T_i s_i^j} \geq 1} \left\lceil \frac{\sum_{j=1}^{n_i} \frac{c_i^j}{s_i^j} - L_i(s_i)}{T_i - L_i(s_i)} \right\rceil +$$

$$2 * \sum_{\forall \tau_i \text{ with } \sum_{j=1}^{n_i} \frac{c_i^j}{T_i s_i^j} < 1} \sum_{j=1}^{n_i} \frac{c_i^j}{T_i s_i^j} \leq m \tag{7}$$

It is worth noting that an optimal solution of the above formulation will not select any speed lower than critical speed as the solution then is non-optimal (a scenario where the leakage power dominates). As can be seen from Figure 3a, an optimization technique may choose a node's speed above the critical speed of the core to meet deadline i.e. to meet constraints (5), (6), and (7). But it will never choose a speed lower than the critical speed as it will neither help in meeting deadline nor will help in decreasing energy consumption.

Abstracting away Non-Uniformity. We use the result of speed-up bound (capacity augmentation bound) derived in [44]. We change processor speeds while maintaining the speed-up bound. While the bound in [44] was derived considering uniform-speed, in using it, as long as no core violates the speed-up bound it does not matter if the cores are running at different speeds or not. We can think this in terms of choosing an execution time for each node. In terms of scheduling it is not important how the execution speed-ups are obtained. Once our optimization finds the "best" feasible assignment of execution times for each node that satisfy the bounds of [44], we can use these execution times to do the scheduling upon (logically) identical multiprocessor platform. Thus, we abstracted away any non-uniformity in speed of the processors by considering that each node can change its execution time.

4.2 Steps for Solving Frequency and Core Assignment Problem

In the problem formulated in (4)–(7), the objective (4) and constraint (5) are convex. As we described before, selecting a different speed for a node can change the critical path in a DAG. That is, whenever we choose a new value of our decision variables (speeds), we have to run an algorithm to detect a critical path in the DAG. Every change in decision variable may invoke that algorithm. Thus constraint (6) raises a key challenge as the invocation of critical path finding algorithm may affect the characteristics of the optimization problem. Another key challenge in the above problem is that it is not differentiable as Constraint (7) is non-differentiable. Non-differentiability raises a significant challenge in optimization problem and restrict the applicability of many efficient approaches for solving. Our techniques to handle these challenges are described as follows.

4.2.1 Critical Path Formulation Using Convex Constraints

To address the challenge stemmed from constraint (6), we determine critical path of a DAG through a topological sorting of the nodes and represent the method through a set of convex constraints. For this, a DAG has to have a unique source and a unique sink. If a DAG has multiple sources, a new dummy node with WCER of zero is created as the parent of all source nodes, and this dummy node is considered as the unique source. Similarly, if a DAG has multiple sinks, a new dummy node with WCER of zero is created as the child of all sink nodes, and this dummy node becomes the unique sink. Hence, without loss of generality, from now onward we consider that each τ_i is a DAG with one source W_i^1 and one sink $W_i^{n_i}$, and that $W_i^1, W_i^2, \cdots, W_i^{n_i}$ is a topological ordering of the nodes of τ_i.

It is well-known that the shortest path problem for graphs with non-negative edge weights can be formulated as a linear programming maximization problem (e.g., see Chapter 29 in Cormen et al. [26]). Therefore, we can reverse the constraints of the shortest path formulation and use node weights to obtain quantification of the longest path from the source node W_i^1 to any other node W_i^k. Thus, based on topological ordering, we can express constraint (6) of the above formulation for critical path function in terms of convex constraints (8), (9), and (10) using a new variable d as follows.

$$d_i[W_i^1] = \frac{c_i^1}{s_i^1}, \forall \tau_i \tag{8}$$

$$d_i[W_i^\ell] \geq d_i[W_i^k] + \frac{c_i^\ell}{s_i^\ell}, \quad \forall \tau_i, (W_i^k \rightarrow W_i^\ell) \tag{9}$$

$$d_i[W_i^{n_i}] \leq \frac{T_i}{2}, \forall \tau_i \tag{10}$$

Constraint (8) fixes the weight of the source node. Constraint (9) enforces that the longest path from source to node W_i^ℓ must be no less than the longest path to any adjacent predecessor node plus the weight of W_i^ℓ. Finally, constraint (10) checks to ensure that the longest path to the sink node $W_i^{n_i}$ (i.e., the critical path) is bounded according to constraint (6).

4.2.2 Handling Non-Differentiability by Refining Core Allocation

Upon expressing constraint (6) of the problem given in (4) – (7) in terms of convex constraints (8), (9), and (10), the problem still remains non-differentiable due to constraint (7). A potential approach to solving the problem is to adopt a penalty based simulated annealing. Simulated annealing is a global optimization framework that employs stochastic global exploration to escape from local minima and is suitable for problems where gradient information is not available. A penalty-based approach (such as ℓ_1- penalty method [23, 64]) makes it adoptable to constrained problem. While such a method can achieve global optimality or high-quality solution under certain theoretical conditions and parameter setups, its running time can be very very long, making it impractical for real-time task scheduling. Instead, we propose to formulate a convex optimization problem by modifying the federated scheduling policy while retaining the theoretical performance bound of the scheduling technique compared to the original one. Therefore, an optimal solution of the proposed convex problem should be quite close to that of the original problem defined in (4) – (7).

Since non-differentiability exists in constraint (7), we aim to make it continuous and convex. Its non-differentiability stems from the assignment of integer number of processor cores to the tasks. Hence, we slightly modify the federated scheduling policy by assigning a continuous value to each task based on which the task is actually scheduled on an integer

number of cores (as the number of cores is always integer). This provably retains the theoretical schedulability performance of the federated scheduling. Instead of assigning $\lceil \frac{C_i - L_i}{T_i - L_i} \rceil$ cores to a high utilization task τ_i, we assign the value $(\frac{C_i - L_i}{T_i - L_i} + 1)$ to it (in our optimization) while the task is scheduled on $\lfloor \frac{C_i - L_i}{T_i - L_i} + 1 \rfloor$ cores. The remaining cores are assigned to low-utilization tasks. Theorem 3 proves that the federated scheduling retains its capacity augmentation bound of 2 upon the above modification.

▶ **Theorem 3.** *Assigning the value* $(\frac{C_i - L_i}{T_i - L_i} + 1)$ *to each high utilization task* τ_i *and scheduling it on* $\lfloor \frac{C_i - L_i}{T_i - L_i} + 1 \rfloor$ *cores, and assigning the remaining cores to all low-utilization tasks does not change the capacity augmentation bound of* 2 *of the federated scheduling.*

Proof. We consider a task set τ that satisfies Conditions 1 and 2 from Definition 1 for $b = 2$. As proved in [44], the capacity augmentation bound is 2 if every high-utilization task gets at least $\lceil \frac{C_i - L_i}{T_i - L_i} \rceil$ cores and all low-utilizations tasks together get at least $2 \sum_{\tau_i \in \tau_{\text{low}}} u_i$ cores. Hence, it is sufficient to prove that these two conditions hold upon our modification.
Our assigned value to a high utilization task τ_i,

$$\frac{C_i - L_i}{T_i - L_i} + 1 > \left\lceil \frac{C_i - L_i}{T_i - L_i} \right\rceil.$$

The actual number of cores on which task τ_i is scheduled in the modified federated scheduling is

$$m_i = \left\lfloor \frac{C_i - L_i}{T_i - L_i} + 1 \right\rfloor \geq \left\lceil \frac{C_i - L_i}{T_i - L_i} \right\rceil.$$

Now we have to prove that the total number of cores assigned to low utilization tasks is at least $2 \sum_{\tau_i \in \tau_{\text{low}}} u_i$. For simplicity let $\sigma_i = \frac{T_i}{L_i}$. Hence, $T_i = \sigma_i L_i, C_i = u_i T_i = \sigma_i u_i L_i$. Since each task satisfies Condition 2 from Definition 1 for $b = 2$, $L_i \leq \frac{T_i}{b} \Rightarrow \sigma_i \geq b = 2$. (Note $T_i = D_i$). By the definition of high-utilization task τ_i, $u_i \geq 1$. Together with $\sigma_i \geq 2$, we can state $\frac{(u_i - 1)(\sigma_i - 2)}{\sigma_i - 1} \geq 0$. Now

$$\begin{aligned} &\frac{C_i - L_i}{T_i - L_i} + 1 = \frac{\sigma_i u_i L_i - L_i}{\sigma_i L_i - L_i} + 1 = \frac{\sigma_i u_i + \sigma_i - 2}{\sigma_i - 1} \\ &\leq \frac{\sigma_i u_i + \sigma_i - 2}{\sigma_i - 1} + \frac{(u_i - 1)(\sigma_i - 2)}{\sigma_i - 1} = \frac{2u_i(\sigma_i - 1)}{\sigma_i - 1} = 2u_i. \end{aligned}$$

Now total cores assigned to low-utilization tasks,

$$\begin{aligned} m_{\text{low}} &= m - \sum_{\tau_i \in \tau_{\text{high}}} \left(\frac{C_i - L_i}{T_i - L_i} + 1 \right) \geq m - \sum_{\tau_i \in \tau_{\text{high}}} 2u_i \\ &\geq 2u_{\text{sum}} - \sum_{\tau_i \in \tau_{\text{high}}} 2u_i = 2 \sum_{\tau_i \in \tau_{\text{low}}} u_i \end{aligned}$$

◀

The above modification in federated scheduling makes constraint (7) continuous and convex.

4.3 Energy-Aware Scheduling upon Convex Optimization

By incorporating the results from Sections 4.2.1 and 4.2.2, we can express the new formulation for determining node speeds as follows.

$$\text{Minimize} \quad \sum_{i=1}^{n} w_i \sum_{j=1}^{n_i} E(c_i^j, s_i^j)$$

$$\text{subject to} \quad \sum_{i=1}^{n} \frac{\sum_{j=1}^{n_i} \frac{c_i^j}{s_i^j}}{T_i} \leq \frac{m}{2};$$

$$d_i[W_i^1] = \frac{c_i^1}{s_i^1}, \forall \tau_i;$$

$$d_i[W_i^\ell] \geq d_i[W_i^k] + \frac{c_i^\ell}{s_i^\ell}, \quad \forall \tau_i, (W_i^k \rightarrow W_i^\ell);$$

$$d_i[W_i^{n_i}] \leq \frac{T_i}{2}, \forall \tau_i;$$

$$\sum_{\forall \tau_i \text{ with } \sum_{j=1}^{n_i} \frac{c_i^j}{T_i s_i^j} \geq 1} \left(\frac{\sum_{j=1}^{n_i} \frac{c_i^j}{s_i^j} - L_i(s_i)}{T_i - L_i(s_i)} + 1 \right)$$

$$+ 2 * \sum_{\forall \tau_i \text{ with } \sum_{j=1}^{n_i} \frac{c_i^j}{T_i s_i^j} < 1} \sum_{j=1}^{n_i} \frac{c_i^j}{T_i s_i^j} \leq m$$

The objective and all the constraints in the above problem are now convex, making it a convex optimization problem. Therefore, we can find its **optimal** solution in **polynomial time** through gradient based or Interior Point method using any convex problem solver. In practice, it is very efficient to use any standard convex optimization tool such as CVX [30], IPOPT [36], or MATLAB's `fmincon` function [4] for an optimal solution.

Note that by solving the above formulation, we jointly determine speed of nodes and the number of cores for each DAG as well as for all low-utilization tasks. After the core and speed assignment, every cluster of cores for high-utilization task boils down to scheduling a single DAG on m_i cores. Note that we need to run the above optimization only once. After determining the running speeds of all nodes, we schedule the tasks on their assigned cores. Through incorporating DVFS with federated scheduling, the speeds of the cores are adjusted according to what nodes they are executing. Such speed switching is quite feasible and common in practice. Modern microprocessors tend to change their DVFS setting rather frequently in response to rapid changes in the application behavior [59]. Note that at anytime if a core remains idle, it can be shutdown for energy saving and turned on later when needed.

Low-utilization tasks are treated and executed as if they were sequential tasks. Thus, the cluster of cores assigned to low-utilization task boils down to multiprocessor scheduling of sequential tasks. Our speed assignment technique hence considered an entire low-utilization task as a single node and there was one speed assignment for every low-utilization task. As we mentioned before, any multiprocessor scheduling algorithm such as partitioned EDF [48], or various rate-monotonic schedulers [9] with a utilization bound of at most $\frac{1}{2}$ can be used to schedule all the low-utilization tasks on the allocated m_{low} cores. Since there is much work on energy-aware real-time scheduling of sequential tasks on multiprocessor, we can also adopt one of that policy for further reducing energy consumption. One potential approach

is to adopt the Adaptive DVFS technique proposed in [49] that adaptively determines the frequency of each task for better performance on both schedulability and energy consumption. It schedules the tasks with prolonged execution while ensuring that all meet deadlines.

4.4 Other Factors of Energy Consumption

In federated scheduling, there is no preemption across tasks and hence we have not considered preemption cost. Our proposed solution is optimal only for the formulated problem. Our formulation has ignored several factors of energy consumption. For example, it has ignored the time and energy overhead in speed switching as well as the overhead associated with processor turn on/off. Also, the speed assignment is based on the WCER of the tasks while the actual execution requirement of a task may vary. Our result provides a strong fundamental basis to address these issues in the future. In the future, we shall also address energy minimization considering other key components, i.e., GPU, system bus, and memory/cache.

5 Energy-Aware Global Scheduling

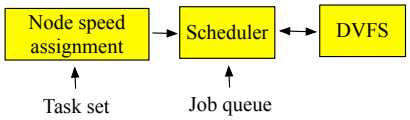


Figure 5 Energy-aware global scheduling framework.

We now describe our proposed scheduling approach for energy-aware global real-time scheduling. Here also, our approach is to regulate the frequencies (speed) across the cores and across different nodes of the same task for minimizing overall energy consumption. Hence, our approach incorporates an optimization engine and DVFS into the classical real-time scheduling policies such as EDF and DM and makes them energy-aware. Specifically, once the execution speeds (and hence the processor frequencies) of the nodes of all DAGs are determined through optimization, we can adopt an existing global real-time scheduling policy. Figure 5 shows our energy-aware global scheduling framework. When the processor frequencies are adjusted based on the nodes' speed assignments, the adopted scheduling policy leads to energy minimization. We describe our techniques for global EDF and DM.

5.1 Energy-Aware Global EDF

EDF is a classic and widely adopted dynamic-priority scheduling policy where tasks are prioritized according to their absolute deadlines. In our approach, the EDF policy is adopted after assigning the node execution speeds. Once the execution speeds of the nodes of all DAGs are determined, and the processor frequencies are adjusted accordingly through DVFS, running EDF policy leads to energy minimization. A node is ready to execute when all its predecessors have executed. EDF for parallel tasks works as follows: at each time step, the scheduler first tries to schedule as many ready nodes from all jobs with the earliest deadline as it can; then it schedules ready nodes from the jobs with the next earliest deadline, and so on, until either all cores are busy or no nodes are ready.

Formulation. We use the overall energy consumption used in our formulation in Section 4.1 as our objective here also. Similar to that formulation, here also we want to minimize this objective while ensuring the schedulability under the global EDF scheduling policy. For global

EDF scheduling of DAG tasks on multicore, the work in [44] derived a capacity augmentation bound of 2.618. When node W_i^j of task τ_i is assigned speed s_i^j, where $1 \leq j \leq n_i, 1 \leq i \leq n$, using a capacity augmentation bound of 2.618, we define **Condition 1** and **Condition 2** for schedulability in Remark (2) as Constraints (12) and (13), respectively, in our formulation. Thus, for global EDF scheduling for energy minimization, we first formulate our problem as follows. Our objective is to determine speeds $s_i = \{s_i^1, s_i^2, \cdots, s_i^j, \cdots, s_i^{n_i}\}, \forall \tau_i$ to

$$\text{Minimize} \quad \sum_{i=1}^{n} w_i \sum_{j=1}^{n_i} E(c_i^j, s_i^j) \tag{11}$$

$$\text{subject to} \quad \sum_{i=1}^{n} \frac{\sum_{j=1}^{n_i} \frac{c_i^j}{s_i^j}}{T_i} \leq \frac{m}{2.618} \tag{12}$$

$$L_i(s_i) \leq \frac{T_i}{2.618}, \forall \tau_i \tag{13}$$

Convexification. The objective (11) and constraint (12) are convex. By incorporating the results from Section 4.2.1, constraint (13) can be replaced by three convex constraints using a new variable d considering $W_i^1, W_i^2, \cdots, W_i^{n_i}$ as a topological ordering of the nodes of DAG τ_i with one source and one sink. Thus, we convert the above problem formulation as the following convex optimization problem where our objective is to determine speeds $s_i = \{s_i^1, s_i^2, \cdots, s_i^j, \cdots, s_i^{n_i}\}, \forall \tau_i$ to

$$\text{Minimize} \quad \sum_{i=1}^{n} w_i \sum_{j=1}^{n_i} E(c_i^j, s_i^j) \tag{14}$$

$$\text{subject to} \quad \sum_{i=1}^{n} \frac{\sum_{j=1}^{n_i} \frac{c_i^j}{s_i^j}}{T_i} \leq \frac{m}{2.618} \tag{15}$$

$$d_i[W_i^1] = \frac{c_i^1}{s_i^1}, \forall \tau_i \tag{16}$$

$$d_i[W_i^\ell] \geq d_i[W_i^k] + \frac{c_i^\ell}{s_i^\ell}, \quad \forall \tau_i, (W_i^k \rightarrow W_i^\ell) \tag{17}$$

$$d_i[W_i^{n_i}] \leq \frac{T_i}{2.618}, \forall \tau_i \tag{18}$$

The objective and all the constraints in the above problem are now convex, making it a convex optimization problem. Therefore, we can find its **optimal** solution in **polynomial time** through any convex problem solver. Note that we need to run the above optimization only once. After determining the running speeds of all nodes, we execute the tasks based on EDF scheduling. During execution, a node runs at its assigned speed. Through incorporating DVFS with EDF scheduling, the speeds of the cores are adjusted according to what nodes they are executing. Note that at anytime if a core remains idle, it can be shutdown for energy saving and turned on later when needed.

For the same reasons we explained for the problem in Section 4, the above said solution is optimal only for the problem formulated in (14)–(18), and is not an optimal solution for actual energy minimization for a number of issues not addressed in the problem formulation such as frequency switching overhead, preemption cost in global scheduling, task execution time uncertainty, and contributions from other components including GPU, system bus, and memory/cache. As stated before, we shall address these issues in the future.

5.2 Energy-Aware Global DM

DM is an efficient and widely used fixed-priority scheduling policy for real-time systems where tasks are assigned priorities according to their (relative) deadlines; the task with the shortest deadline being assigned the highest priority. Our approach for energy-aware DM scheduling is similar to that for energy-aware EDF scheduling described above. We first determine the execution speeds of the nodes. We then incorporate DVFS with DM so that every node runs at the assigned speed during execution.

For global DM scheduling of DAG tasks on multicore, the work in [44] derived a capacity augmentation bound of 3.732. Using this bound, we formulate the problem in the same way as follows considering $W_i^1, W_i^2, \cdots, W_i^{n_i}$ as a topological ordering of the nodes of DAG τ_i. Our objective is to determine speeds $s_i = \{s_i^1, s_i^2, \cdots, s_i^j, \cdots, s_i^{n_i}\}, \forall \tau_i$ to

$$\text{Minimize} \quad \sum_{i=1}^{n} w_i \sum_{j=1}^{n_i} E(c_i^j, s_i^j)$$

$$\text{subject to} \quad \sum_{i=1}^{n} \frac{\sum_{j=1}^{n_i} \frac{c_i^j}{s_i^j}}{T_i} \leq \frac{m}{3.732}$$

$$d_i[W_i^1] = \frac{c_i^1}{s_i^1}, \forall \tau_i$$

$$d_i[W_i^\ell] \geq d_i[W_i^k] + \frac{c_i^\ell}{s_i^\ell}, \quad \forall \tau_i, (W_i^k \rightarrow W_i^\ell)$$

$$d_i[W_i^{n_i}] \leq \frac{T_i}{3.732}, \forall \tau_i$$

The characteristics of the above problem are the same as those of the energy-aware EDF scheduling. Specifically, this problem is also convex and its optimal solution can be achieved in polynomial time through any convex problem solver.

6 Evaluation

In this section, we present the evaluation of our energy-minimization policies using synthetic workloads through simulations. We used a custom-built simulator developed in MATLAB. We conduct simulation for both federated scheduling and global scheduling.

6.1 Simulation Setup and Evaluation Metrics

We generated DAG task sets using the Erdös-Rènyi method $G(n_i, p)$ where n_i is the number of nodes and p is the probability of adding edges between nodes [25]. The value of n_i was chosen randomly from range $[5, 10]$. If a DAG was disconnected, we added the minimum number of edges needed to make it connected. Each node's execution requirement was randomly chosen from range $[5, 10]$. We assign harmonic period T_i to a task τ_i by finding the smallest value x such that critical path, $L_i <= 2^x$ and set T_i to be either 2^x or 2^{x+1}. T_i value was chosen randomly to get a fair distribution of high and low utilization tasks in a task set. Harmonic periods were used to reduce the time needed for collecting the results. We added tasks to a set until its total utilization was achieved. We show the simulation results for an average over 900 task sets using 20 cores (m). To compute the node speeds, we use MATLAB's `fmincon` which provides an optimal solution of our convex problem.

The energy consumption of a task set is computed based on our objective function. We then compute the ratio of the energy consumption and the hyper-period to get the *average power consumption* and use it as our key metric for evaluation. In our proposed approach, the optimization problem uses a schedulability test as a constraint that guarantees that a task is always schedulable (assuming no extra system overhead). Thus, evaluation in terms of schedulability is redundant and is not shown when our approaches are compared against traditional (energy-unaware) real-time scheduling policies under similar schedulability conditions. However, under federated scheduling, we compare our approach against an additional baseline proposed in [17] that trades energy consumption for schedulablity but cannot pre-set the number of cores as an input. For this particular case, schedulability becomes a differentiating factor and will be used as a metric for evaluation. Specifically, our approach always considers schedulable tasks and outperforms the baseline (Section 6.2).

6.2 Results for Energy-Aware Federated Scheduler

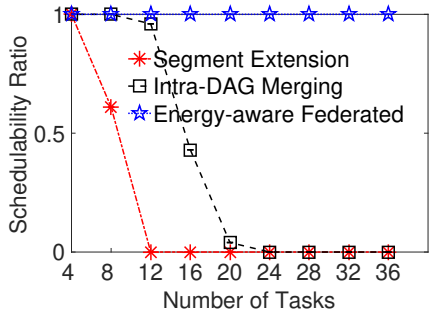
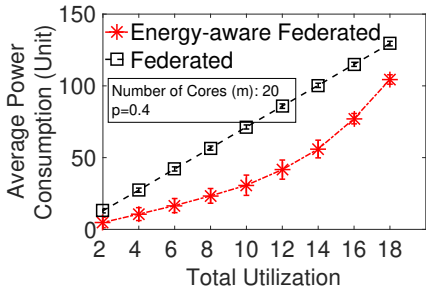


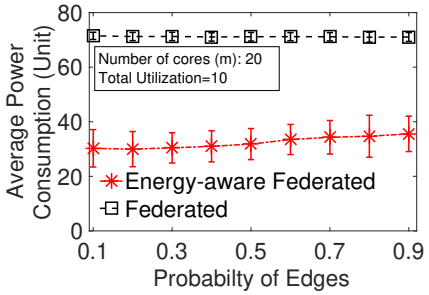
Figure 6 Schedulability under energy-aware federated and existing approach.

We first compare the performance of the energy-aware federated scheduler against the energy-aware simplified federated scheduling, called *intra-DAG merging*, proposed in [17]. As discussed before in Section 2, in this intra-DAG merging approach, the limitation on number of concurrent tasks degrades the schedulability of tasks while our approach ensures schedulability of any feasible task set. To demonstrate their schedulability difference, we generate 900 random DAG task sets with utilization between $0.1m$ and m. We evaluate the performance under varying number of tasks. Figure 6 shows that intra-DAG merging performs poorly in schedulability ratio (i.e., the fraction of schedulable cases) compared to our energy-aware federated scheduler. When the number of tasks reaches 20, the schedulabilty ratio of the intra-DAG merging approach is 0, while our scheduler can always schedule the generated task set. Since the performance of Intra-DAG merging is very poor in terms of schedulability, we do not consider it in comparing energy-consumption.

Figure 7a shows the results for our energy-aware federated scheduler for 900 task sets under varying total utilization of each set in the range $[2, 18]$. All other parameters are fixed as before. We compare the performance with (energy-unaware) federated scheduler. We considered the speed of tasks running in federated scheduler to be 2.0. The figure shows that our energy-aware federated scheduler saves up to 62.95% of the power when compared to federated scheduler. Figure 7b shows the energy performance under varying p. We see that our energy-aware federated scheduler consistently outperforms federated scheduler and can achieve up to 57.95% energy saving compared to the latter.



(a) Varying task set utilization.



(b) Varying probability of edges.

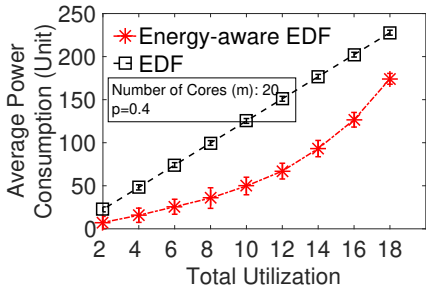
Figure 7 Performance of Energy-Aware Federated Scheduler.

6.3 Results for Energy-Aware Global Scheduling

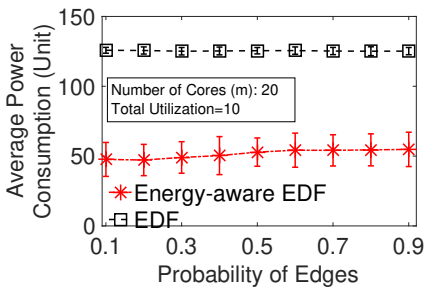
We now present the results under global EDF and global DM scheduling.

6.3.1 Energy Savings under Energy-Aware Global EDF

In Figure 8a, we show the average power consumption over a hyper-period for 900 DAG task sets and vary the total utilization from 2 to 18. For a moderately dense DAG, we set $p = 0.4$. We compare our energy-aware global EDF policy with traditional energy-unaware global EDF (named "EDF" subsequently) scheduling as, to our knowledge, no approach has yet been proposed for global energy-aware scheduling of parallel tasks for multicore. Under EDF, all tasks run at the speed-up bound, 2.618, since a speed of 2.618 always ensures schedulability of a feasible task set. Figure 8a shows that the average power consumption of our approach increases with an increase in total utilization. This is due to the reduction in available slack with the increase in total utilization. From this result, we can see that our energy-aware EDF consumes up to 68.17% less power compared to EDF. Figure 8b shows the performance under different DAG structures by varying edge probability p uniformly in range $[0.1, 0.9]$ setting total utilization at 10. Our approach is consistently better than EDF and can achieve up to 62.4% power saving compared to EDF.



(a) Varying task set utilization.

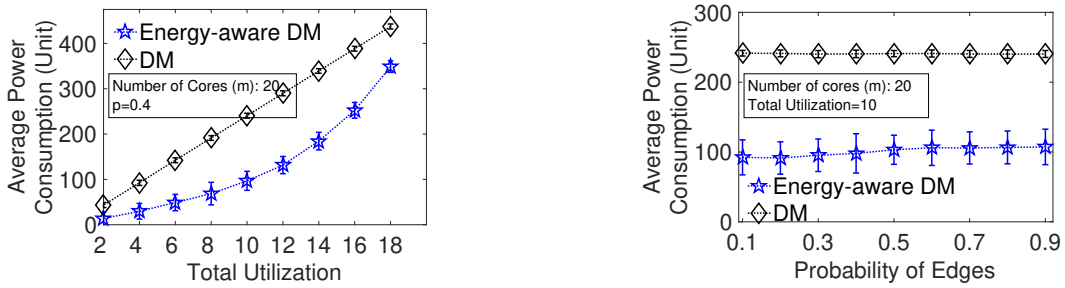


(b) Varying probability of edges.

Figure 8 Performance of Energy-Aware Global EDF.

6.3.2 Energy Savings under Energy-Aware Global DM

In Figure 9a, we show the average power consumption in our energy-aware DM against traditional energy-unaware DM (named "DM" subsequently) under varying total utilization. Here our parameters are chosen similar to the EDF case. We consider the speeds of all task running under DM to be 3.732 which is the corresponding speed up bound for ensuring schedulability under DM. As the speed-up bound for DM is higher than EDF, we see an increase in average power consumption for both energy-aware DM and DM. However, our energy-aware DM consumes up to 64.10% less power than DM. Figure 9b shows the performance under different DAG structures by varying p uniformly in range $[0.1, 0.9]$ setting total utilization at 10. As the figure shows our approach is consistently better than DM and can achieve up to 62.1% power saving compared to DM. For both schedulers, the DAG structure does not show observable effect in performance.



(a) Varying task set utilization.

(b) Varying probability of edges.

Figure 9 Performance of Energy-Aware Global DM.

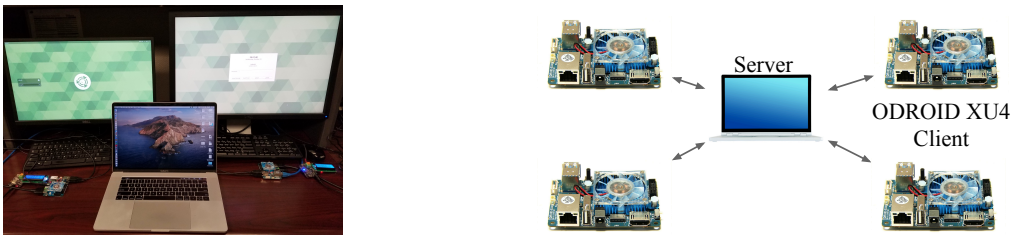
7 Proof of Concept

Although many AMD and Intel processors support per-core DVFS, we could not use those in PC/Laptop for our experiment as we found it quite challenging to accurately measure the CPU power consumption due to interference from numerous sources such as peripheral devices and other programs. Hence, we have developed a proof of concept system for our results using multiple ODROID boards [54] where this issue is less severe.

7.1 Proof of Concept System Setup

ODROID board equips an Exynos5 Octa 5422 SoC consisting of two quad-core clusters – a "LITTLE" cluster with four ARM Cortex-A7 and a "big" cluster with four ARM Cortex-A15. We have used Ubuntu 16.04.1 LTS MATE operating system with Linux kernel 4.14. Cores of a cluster operate at the same frequency and have the same voltage. Since a single ODROID board does not support per-core DVFS, we created our proof of concept system as a multiprocessor platform by connecting four boards where we used one core from each board. Thus our setup uses four cores from four different boards to enable per-core DVFS. Such a proof of concept platform was made up only to enable per-core DVFS-based experiment.

Figure 10a shows our experimental setup and Figure 10b shows its architecture as a multi-processor environment. We connected each ODROID board to a central server that resides on a MacBook Pro via a wired LAN. The central server schedules nodes from all jobs, based on the proposed energy-aware federated scheduler, on the clients. Each node is



(a) Experimental Setup. **(b)** Experimental Architecture.

Figure 10 Experiment Setup and Architecture.

emulated by a C program that is executing an empty `for` loop for a specified time. Since our focus is to measure energy consumption, we rely only on an empty `for` loop to expend processor cycles which consumes energy similar to a regular instruction execution while avoiding any cache/memory overheads. The scheduler located at the server maintains the dependencies among nodes. Thus, a node was executed only when all of its predecessor nodes had finished execution. Upon receiving a request from the server, the ODROID client executes the C program as a process on a dedicated core (i.e., without interference from OS or other programs). In our implementation, we used CPUFREQ-SET program from CPUFREQUTILS package for setting the frequency of each CPU. The overhead from the client program and driver is very low and can be included in the context switch cost.

We set the frequency of both the big and LITTLE cluster at $600Mhz$ by default. Upon receiving a request from the server, the client changes the frequency of the LITTLE cluster to the requested value and executes the program for a specified duration. Upon its completion, the frequency is scaled back to $600Mhz$. The choice of $600Mhz$ stems from the observations that below $600Mhz$ the operating system struggles to schedule processes on time. Note that we do not put the core on the ODROID board that is executing the client to sleep as it incurs additional time overhead which can affect the schedulability of a task set.

We used ODROID's smartpower device [6] for measuring power consumption during our experiments. The smartpower device uses a discrete sampling of voltage and current to compute the instantaneous power and total energy consumption for the entire board. It transmits these values over WiFi, which is collected manually for the hyper-period of the task set. Note that the energy measurement over the hyper-period for one board includes the total energy consumed by context switches, OS interferences, and peripheral chips/devices connected to the ODROID board. Since we use four ODROID boards, the number of peripheral devices and their static energy consumption scales accordingly.

7.2 Implementation and Experimental Methodology

We generated 4 DAG task sets using the approach described in Section 6.1. The worst-case execution time of each node was chosen randomly from $[27s, 54s, 81s, 108s, 135s]$, which corresponds to $[2, 4, 6, 8, 10] \times 10^9$ empty for loop iterations on 0.7Ghz . The task sets for experiments had total utilization of $0.3m$, $0.4m$, $0.5m$ and $.6m$ ($m = 4$). The value of p was fixed at 0.6. All other parameters were kept the same as simulation. The speed of each node in a task was calculated using MATLAB's `fmincon`. Here also we evaluate in terms of *average power consumption* which is the average power in Watts consumed by the platform during the execution of one task set up to its hyper-period. We measured the total energy consumed over the hyper-period and used it to compute the average power.

7.3 Proof of Concept Results

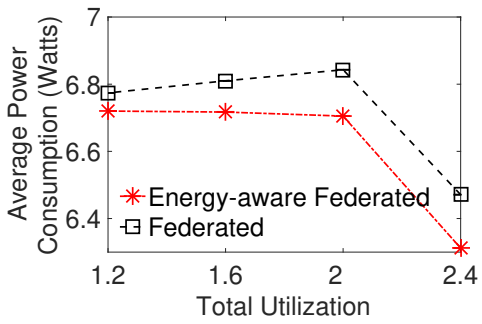


Figure 11 Performance of energy-aware federated scheduling.

Figure 11 shows that the average power consumption of the energy-aware federated scheduler is lower than the existing energy-unaware federated scheduler (named "federated scheduler" subsequently) at all values of total utilization. There is up to 0.159 Watts of power savings in our approach. Due to the randomness of the task set generation policy, the task set with utilization value 2 has tasks with smaller critical path length compared to the task set with utilization 2.4. (This is due to our task generation policy that connected two nodes in the graph based on whether a randomly generated number is less than 0.6 or not. For task at utilization 2.4, the random task generator did not add as many edges as the task at utilization 2 which resulted in smaller critical path length.) This resulted in higher frequency assignment for all nodes in the former. Thus, the average power consumption for task set with utilization 2.4 was lower than that of task set with utilization 2.

Figure 11 shows the average power consumption for both the baseline and the proposed approach is in the order of $6 - 7$ Watts. This high power consumption is due to the static power consumed by input devices (like keyboard and mouse), communication chips, RAM/memory modules, and the big cluster operating 600Mhz on each ODROID board. In a processor with per-core DVFS, the static power consumption would be significantly lower than that exhibited by the proof of concept platform, and hence the average power savings as a percentage of the baseline power consumption will be significantly higher. Due to the high static power consumption of our experimental platform, the proof of concept results only show energy savings but are not indicative of the percentage of energy savings that can be obtained through the proposed approach.

8 Conclusion and Future Directions

In this paper, we have proposed energy-aware real-time scheduling for parallel tasks. We have demonstrated its performance through simulations as well as a proof of concept system evaluation on a physical platform. The adopted resource augmentation bounds only provide a sufficient condition for schedulability. Any improvement over the resource augmentation bounds in the future would lead to more energy efficient scheduler under our framework. We have limited the scope of our results to CPU energy consumption ignoring several practical factors. For example, we have ignored the overhead in processor speed switching as well as turn on/off. Also, the speed assignment is based on the worst-case execution requirement of the tasks while the actual execution requirement of a task may vary across runs/instances.

Yet, our results provide a strong fundamental basis to incorporate these considerations into future research. In the future, we shall also take into account all key components, i.e., CPU, GPU, system bus, and memory/cache, that contribute to energy consumption.

References

1 URL: https://www.greentechmedia.com.

2 URL: https://web.archive.org/web/20090326020946/http://www.aero.org/conferences/mrqw/2002-papers/A_Burcin.pdf.

3 Intel SCC. URL: https://www.intel.cn/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-single-chip-cloud-overview-paper.pdf.

4 Matlab fmincon. URL: https://www.mathworks.com/help/optim/ug/fmincon.html.

5 Pc 205. URL: https://en.wikipedia.org/wiki/PicoChip.

6 Smartpower. URL: hardkernel.com/shop/smartpower2-with-15v-4a/.

7 TILE-Gx™. URL: http://www.tilera.com/products/processors/TILE-Gx_Family.

8 Björn Andersson and Dionisio de Niz. Analyzing global-edf for multiprocessor scheduling of parallel tasks. In *OPODIS*, pages 16–30. Springer, 2012.

9 Björn Andersson and Jan Jonsson. The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. In *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings.*, pages 33–40, 2003.

10 Hakan Aydin and Qi Yang. Energy-aware partitioning for multiprocessor real-time systems. In *Parallel and Distributed Processing Symposium. Proceedings. International*, pages 9–pp. IEEE, 2003.

11 Mario Bambagini, Mauro Marinoni, Hakan Aydin, and Giorgio Buttazzo. Energy-aware scheduling for real-time systems: A survey. *ACM Trans. Embed. Comput. Syst.*, 15(1):7:1–7:34, January 2016.

12 Sanjoy Baruah. Improved multiprocessor global schedulability analysis of sporadic DAG task systems. In *26th Euromicro Conference on Real-Time Systems*, pages 97–105, 2014.

13 Sanjoy Baruah, Vincenzo Bonifaci, and Alberto Marchetti-Spaccamela. The global EDF scheduling of systems of conditional sporadic DAG tasks. In *27th Euromicro Conference on Real-Time Systems*, pages 222–231. IEEE, 2015.

14 Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. A generalized parallel task model for recurrent real-time processes. In *Real-Time Systems Symposium (RTSS), IEEE 33rd*, pages 63–72. IEEE, 2012.

15 A. Bhuiyan, D. Liu, A. Khan, A. Saifullah, N. Guan, and Z. Guo. Energy-efficient parallel real-time scheduling on clustered multi-core. *IEEE Transactions on Parallel and Distributed Systems*, 31(9):2097–2111, 2020.

16 A. A. Bhuiyan, K. Yang, S. Arefin, A. Saifullah, N. Guan, and Z. Guo. Mixed-criticality multicore scheduling of real-time gang task systems. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 469–480, 2019.

17 Ashikahmed Bhuiyan, Zhishan Guo, Abusayeed Saifullah, Nan Guan, and Haoyi Xiong. Energy-efficient real-time scheduling of dag tasks. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(5):84, 2018.

18 Enrico Bini, Giorgio Buttazzo, and Giuseppe Lipari. Minimizing CPU energy in real-time systems with discrete speed management. *ACM Transactions on Embedded Computing Systems (TECS)*, 8(4):31, 2009.

19 Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Sebastian Stiller, and Andreas Wiese. Feasibility analysis in the sporadic DAG task model. In *25th Euromicro Conference on Real-Time Systems*, pages 225–233. IEEE, 2013.

20 Gang Chen, Kai Huang, and Alois Knoll. Energy optimization for real-time multiprocessor system-on-chip with optimal DVFS and DPM combination. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(3s):111, 2014.

21 Jian-Jia Chen, Heng-Ruey Hsu, and Tei-Wei Kuo. Leakage-aware energy-efficient scheduling of real-time tasks in multiprocessor systems. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, pages 408–417. IEEE, 2006.

22 Jian-Jia Chen, Andreas Schranzhofer, and Lothar Thiele. Energy minimization for periodic real-time tasks on heterogeneous processing units. In *Parallel & Distributed Processing. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

23 Yixin Chen and Minmin Chen. Extended duality for nonlinear programming. *Comput. Optim. Appl.*, 47:33–59, 2010.

24 Alexei Colin, Arvind Kandhalu, and Ragunathan Rajkumar. Energy-efficient allocation of real-time applications onto heterogeneous processors. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*, pages 1–10. IEEE, 2014.

25 Daniel Cordeiro, Gregory Mouni, Swann Perarnau, Denis Trystram, Jean-Marc Vincent, and Frederic Wagner. Random graph generation for scheduling simulations. In *Proceedings of the 3rd international ICST conference on simulation tools and techniques*, page 60. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010.

26 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

27 Vinay Devadas and Hakan Aydin. Coordinated power management of periodic real-time tasks on chip multiprocessors. In *Green Computing Conference, 2010 International*, pages 61–72. IEEE, 2010.

28 David Ferry, Jing Li, Mahesh Mahadevan, Kunal Agrawal, Christopher Gill, and Chenyang Lu. A real-time scheduling service for parallel tasks. In *RTAS'13*, 2013.

29 James H. Gawron, Gregory A. Keoleian, Robert D. De Kleine, Timothy J. Wallington, and Hyung Chul Kim. Life cycle assessment of connected and automated vehicles: Sensing and computing subsystem and vehicle level effects. *Environmental Science & Technology*, 52(5):3249–3256, 2018.

30 Michael Grant and Stephen Boyd. CVX: MATLAB software for disciplined convex programming, 2012. URL: `http://cvxr.com/cvx/`.

31 Akhil Guliani and Michael M. Swift. Per-application power delivery. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 5:1–5:16, 2019.

32 Yifeng Guo, Dakai Zhu, and Hakan Aydin. Reliability-aware power management for parallel real-time applications with precedence constraints. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–8. IEEE, 2011.

33 Z. Guo, A. Bhuiyan, D. Liu, A. Khan, A. Saifullah, and N. Guan. Energy-efficient real-time scheduling of dags on clustered multi-core platforms. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 156–168, 2019.

34 Zhishan Guo, Ashikahmed Bhuiyan, Abusayeed Saifullah, Nan Guan, and Haoyi Xiong. Energy-efficient multi-core scheduling for real-time DAG tasks. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 76. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

35 Jason Howard, Saurabh Dighe, Sriram R Vangal, Gregory Ruhl, Nitin Borkar, Shailendra Jain, Vasantha Erraguntla, Michael Konow, Michael Riepen, Matthias Gries, et al. A 48-core ia-32 processor in 45 nm cmos using on-die message-passing and dvfs for performance and power scaling. *IEEE Journal of Solid-State Circuits*, 46(1):173–183, 2011.

36 IPOPT. Interior point optimizer, 2011. URL: `https://projects.coin-or.org/Ipopt`.

37 Ravindra Jejurikar. Energy aware non-preemptive scheduling for hard real-time systems. In *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 21–30. IEEE, 2005.

38 Xu Jiang, Nan Guan, Xiang Long, and Wang Yi. Semi-federated scheduling of parallel real-time tasks on multiprocessors. *arXiv preprint arXiv:1705.03245*, 2017.

39 J. Kim, H. Kim, K. Lakshmanan, and R. Rajkumar. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *2013 ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, pages 31–40, 2013.

40 Junsung Kim, Hyoseung Kim, Karthik Lakshmanan, and Ragunathan Raj Rajkumar. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*, pages 31–40. ACM, 2013.

41 Fanxin Kong, Nan Guan, Qingxu Deng, and Wang Yi. Energy-efficient scheduling for parallel real-time tasks based on level-packing. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 635–640. ACM, 2011.

42 Wan Yeon Lee. Energy-efficient scheduling of periodic real-time tasks on lightly loaded multicore processors. *IEEE Transactions on Parallel and Distributed Systems*, 23(3):530–537, 2012.

43 Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Outstanding paper award: Analysis of global EDF for parallel tasks. In *25th Euromicro Conference on Real-Time Systems*, pages 3–13. IEEE, 2013.

44 Jing Li, Jian-Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *26th Euromicro Conference on Real-Time Systems*, pages 85–96. IEEE, 2014.

45 Keqin Li. Energy efficient scheduling of parallel tasks on multiprocessor computers. *J Supercomput*, 60:223–247, 2012.

46 Cong Liu, Jian Li, Wei Huang, Juan Rubio, Evan Speight, and Xiaozhu Lin. Power-efficient time-sensitive mapping in heterogeneous systems. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 23–32. ACM, 2012.

47 Di Liu, Jelena Spasic, Gang Chen, and Todor Stefanov. Energy-efficient mapping of real-time streaming applications on cluster heterogeneous mpsocs. In *Embedded Systems For Real-time Multimedia (ESTIMedia), 2015 13th IEEE Symposium on*, pages 1–10. IEEE, 2015.

48 José María López, José Luis Díaz, and Daniel F García. Utilization bounds for edf scheduling on real-time multiprocessor systems. *Real-Time Syst.*, 28(1):39–68, October 2004.

49 Junyang Lu and Yao Guo. Energy-aware fixed-priority multi-core scheduling for real-time systems. In *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 277–281, 2011.

50 G. Luo, B. Guo, Y. Shen, H. Liao, and L. Ren. Analysis and optimization of embedded software energy consumption on the source code and algorithm level. In *2009 Fourth International Conference on Embedded and Multimedia Computing*, pages 1–5, 2009.

51 Oliver Mitchell. Self-driving cars have power consumption problems, 2018. URL: `https://www.therobotreport.com/self-driving-cars-power-consumption/`.

52 Sujay Narayana, Pengcheng Huang, Georgia Giannopoulou, Lothar Thiele, and R Venkatesha Prasad. Exploring energy saving for mixed-criticality systems on multi-cores. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12. IEEE, 2016.

53 Geoffrey Nelissen, Vandy Berten, Joël Goossens, and Dragomir Milojevic. Techniques optimizing the number of processors to schedule multi-threaded tasks. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 321–330. IEEE, 2012.

54 ODROID XU-4. http://www.hardkernel.com/.

55 Santiago Pagani and Jian-Jia Chen. Energy efficient task partitioning based on the single frequency approximation scheme. In *Real-Time Systems Symposium (RTSS), IEEE 34th*, pages 308–318. IEEE, 2013.

56 Santiago Pagani and Jian-Jia Chen. Energy efficiency analysis for the single frequency approximation (SFA) scheme. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(5s):158, 2014.

57 Antonio Paolillo, Joël Goossens, Pradeep M Hettiarachchi, and Nathan Fisher. Power minimization for parallel real-time systems with malleable jobs and homogeneous frequencies. In *IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10. IEEE, 2014.

58 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*, pages 329–338, 2016.

59 Sangyoung Park, Jaehyun Park, Donghwa Shin, Yanzhi Wang, Qing Xie, Massoud Pedram, and Naehyuck Chang. Accurate modeling of the delay and energy overhead of dynamic voltage and frequency scaling in modern microprocessors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(5):695–708, 2013.

60 Xuan Qi and Dakai Zhu. Energy efficient block-partitioned multicore processors for parallel applications. *Journal of Computer Science and Technology*, 26(3):418–433, 2011.

61 Abusayeed Saifullah, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. In *RTSS '11*, 2011.

62 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.

63 Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, 49(4):404–435, 2013.

64 Abusayeed Saifullah, Chengjie Wu, Paras Tiwari, You Xu, Yong Fu, Chenyang Lu, and Yixin Chen. Near optimal rate selection for wireless control systems. *ACM Transactions on Embedded Computing Systems*, 13(4s):128:1–128:25, 2013. Special Issue on Real-Time and Embedded Systems.

65 Euiseong Seo, Jinkyu Jeong, Seonyeong Park, and Joonwon Lee. Energy efficient scheduling of real-time tasks on multicore processors. *IEEE transactions on parallel and distributed systems*, 19(11):1540–1552, 2008.

66 Jack Stewart. Self-driving cars use crazy amounts of power, and it's becoming a problem, 2018. URL: `https://www.wired.com/story/self-driving-cars-power-consumption-nvidia-chip/`.

67 Corey Tessler, Venkata Modekurthy, Nathan Fisher, and Abusayeed Saifullah. Bringing inter-thread cache benefits to federated scheduling. In *The 26th IEEE/USENIX Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.

68 Leping Wang and Ying Lu. Efficient power management of heterogeneous soft real-time clusters. In *Real-Time Systems Symposium, 2008*, pages 323–332. IEEE, 2008.

69 Lizhe Wang, Samee U. Khan, Dan Chen, Joanna KolOdziej, Rajiv Ranjan, Cheng-Zhong Xu, and Albert Zomaya. Energy-aware parallel task scheduling in a cluster. *Future Gener. Comput. Syst.*, 29(7):1661–1670, 2013.

70 Lizhe Wang, Gregor Von Laszewski, Jay Dayal, and Fugang Wang. Towards energy aware scheduling for precedence constrained parallel tasks in a cluster with dvfs. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 368–377. IEEE Computer Society, 2010.

71 Huiting Xu, Fanxin Kong, and Qingxu Deng. Energy minimizing for parallel real-time tasks based on level-packing. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 98–103. IEEE, 2012.

72 Ruibin Xu, Dakai Zhu, Cosmin Rusu, Rami Melhem, and Daniel Mossé. Energy-efficient policies for embedded clusters. In *ACM SIGPLAN Notices*, volume 40(7), pages 1–10. ACM, 2005.

73 Chuan-Yue Yang, Jian-Jia Chen, and Tei-Wei Kuo. An approximation algorithm for energy-efficient scheduling on a chip multiprocessor. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, pages 468–473. IEEE Computer Society, 2005.

74 Dakai Zhu, Nevine AbouGhazaleh, Daniel Mossé, and Rami Melhem. Power aware scheduling for and/or graphs in multiprocessor real-time systems. In *Parallel Processing, 2002. Proceedings. International Conference on*, pages 593–601. IEEE, 2002.

75 Dakai Zhu, Daniel Mosse, and Rami Melhem. Power-aware scheduling for and/or graphs in real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):849–864, 2004.

PAStime: Progress-Aware Scheduling for Time-Critical Computing

Soham Sinha
Department of Computer Science, Boston University, MA, USA
soham1@bu.edu

Richard West
Department of Computer Science, Boston University, MA, USA
richwest@bu.edu

Ahmad Golchin
Department of Computer Science, Boston University, MA, USA
golchin@bu.edu

Abstract

Over-estimation of worst-case execution times (WCETs) of real-time tasks leads to poor resource utilization. In a mixed-criticality system (MCS), the over-provisioning of CPU time to accommodate the WCETs of highly critical tasks may lead to degraded service for less critical tasks. In this paper we present PAStime, a novel approach to monitor and adapt the runtime progress of highly time-critical applications, to allow for improved service to lower criticality tasks. In PAStime, CPU time is allocated to time-critical tasks according to the delays they experience as they progress through their control flow graphs. This ensures that as much time as possible is made available to improve the Quality-of-Service of less critical tasks, while high-criticality tasks are compensated after their delays.

This paper describes the integration of PAStime with Adaptive Mixed-criticality (AMC) scheduling. The LO-mode budget of a high-criticality task is adjusted according to the delay observed at execution checkpoints. This is the first implementation of AMC in the scheduling framework of LITMUS^{RT}, which is extended with our PAStime runtime policy and tested with real-time Linux applications such as object classification and detection. We observe in our experimental evaluation that AMC-PAStime significantly improves the utilization of the low-criticality tasks while guaranteeing service to high-criticality tasks.

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

Keywords and phrases progress-aware scheduling, code instrumentation, timing annotation

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.3

Funding This work is supported in part by the National Science Foundation (NSF) under Grant #1527050.

Acknowledgements Special thanks to Dr. Ramesh Peri and Intel for generous support and feedback. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

1 Introduction

In real-time systems, computing resources are typically allocated according to each task's worst-case execution time (WCET), to ensure timing constraints are met. However, worst-case conditions for an application are rather rare, resulting in poor resource utilization. Previous research work [54] shows that the worst-cases lie at the tiny tail-end of the probability distribution curve of the execution times for many programs. Instead, average-case execution times are significantly more likely, taking a fraction of the worst-case times.

Mixed-criticality systems (MCSs) provide a way to avoid over-estimation of resource needs, by considering the schedulability of tasks according to different estimates of their execution times at different criticality or assurance levels [52]. Higher criticality tasks are afforded more execution time at the cost of less time for lower criticality tasks, when it is impossible to meet all task timing constraints. There have been multiple proposals [6,7,12,15] since Vestal's work on MCSs [52]. Most prior work focuses on meeting task deadlines and ignores other Quality-of-Service (QoS) metrics [51] or average utilization. Although timely completion is most important for high-criticality applications, QoS is a significant metric for lower criticality tasks [11,36,48]. This has motivated our work on PAStime (**P**rogress-**A**ware **S**cheduling for *time*-critical computing), to maximize the QoS for low-criticality tasks.

In PAStime, we first identify a checkpoint in a high-criticality application at an intermediate stage in its source code. The application is then profiled offline to measure the time to reach the marked checkpoint. Using this timing data, the application evaluates its progress at the checkpoint during runtime. Based on the delay at the checkpoint, PAStime predicts the expected execution time of a high-criticality application. We consequently adjust the runtime of the application, given that the change does not hamper schedulability of co-running tasks. If at runtime a highly critical program is deemed to be making insufficient progress, it is given greater CPU time.

We combine PAStime with Adaptive Mixed-criticality (AMC) scheduling [7], to improve the performance of low-criticality tasks. In AMC, the system is started in *LO-mode*, where all tasks are scheduled with their LO-mode budgets. When a high-criticality task runs for more than its LO-mode budget, the system is switched to *HI-mode*. In HI-mode, all low-criticality tasks are stopped, and the high-criticality tasks are given their increased HI-mode budgets. However, switching to HI-mode should be avoided as much as possible [5], because it affects the performance of low-criticality tasks, which are not executed in HI-mode.

Several works extend the mixed-criticality task model to improve the performance of low-criticality tasks, such as providing an offline extra budget allowance to the high-criticality tasks [45], and estimating multiple budgets [35,43] and periods [46,48] for low-criticality tasks. However, these approaches do not utilize runtime information.

We extend AMC with PAStime to avoid mode switches, by dynamically adjusting the LO-mode budget for a high-criticality task, based on progress to execution checkpoints. Then, we predict the expected execution time of a high-criticality task based on the observed delay until a checkpoint. We carry out an efficient online schedulability test to determine whether we can increase the LO-mode budget of the delayed high-criticality task to our predicted execution time. In case the taskset is still schedulable with the increased budget, we extend the LO-mode budget of the high-criticality task to the predicted execution time. When a high-criticality task finishes within its extended budget, we keep the system in LO-mode and avoid a mode switch that would otherwise happen in AMC. Thus, PAStime improves the QoS of low-criticality tasks by keeping the system in LO-mode for a longer time.

Factors such as I/O events and hardware microarchitectural delays lead to actual execution times exceeding those predicted by PAStime. Any high criticality task running at the end of its predicted execution time causes a timer interrupt to switch the system into HI-mode, as is the case with AMC scheduling. Thus, a high-criticality task never misses its deadline.

1.1 Contributions

The central idea of PAStime is to help the OS make scheduling decisions based on a program's runtime progress. This work is the first implementation of AMC in the scheduling framework of LITMUS^{RT} [10,14]. Such an implementation helps in testing AMC scheduling with a wide range of real-time Linux applications. We also implement an extension to AMC scheduling with our PAStime runtime policy. We test our implementation with real-world applications: an object classification application from the Darknet neural network framework [44], an object tracking application from the dlib machine learning toolkit [16], and an MPEG video decoder [20]. We show that PAStime increases the average utilization of low-criticality tasks by 1.5 to 9 times for 2 to 20 tasks. We also demonstrate that our implementation of AMC-PAStime has minimal and bounded additional overhead in LITMUS^{RT}.

We provide a C library for PAStime to instrument checkpoints in high-criticality programs. In addition, we modify the LLVM compiler [29] to *automatically* identify potential locations of checkpoints during profiling for simple time-critical applications, and also instrument the checkpoints in the final binary executable file.

The next section describes our approach to PAStime. Section 3 details the theoretical background behind AMC and its extension with PAStime. Section 4 describes the design and implementation of PAStime, which is then evaluated in Section 5. Finally, we describe related work, followed by conclusions and future work in Sections 6 and 7, respectively.

2 Approach

Compiler infrastructures such as LLVM are capable of producing a program's control flow graph (CFG). A CFG represents the interconnection between multiple *Basic Blocks (BBs)*, where a block is a sequence of straight-line code without any internal branches. However, CFGs are not typically utilized by an OS to manage time for different computing resources, in spite of being a rich source of analytical information about a program. Consequently, current OSs are oblivious to a program's computing requirements (e.g., CPU utilization) at different points in its execution. A developer of an application can, instead, help the OS make decisions related to resource management, by providing runtime information about a program at certain points in its source code.

PAStime dynamically decides a program's execution budget based on its runtime progress and theoretical analysis of the allowable delay at specific checkpoint locations. At runtime, PAStime measures the time (i.e., CPU time) to reach a checkpoint from the start of a task, and then compares that time to a pre-profiled reference value. The task's execution budget which was previously set based on profiling, is then adjusted according to actual progress.

Figure 1 shows the CFG for a program with two loops, starting at BB2 and BB6. In this example, PAStime inserts a checkpoint between the two loops at the end of BB5. BB5 is a potentially good location for a checkpoint because there is one loop before and after this BB, providing an opportunity to increase the budget to compensate for the delay until BB5.

Suppose that we derive the LO-mode budget of the whole program to be 2000 ms by profiling, and the LO-mode time to reach the checkpoint at BB5 is 500 ms. The program is then executed in the presence of other tasks. The execution budget at the checkpoint (in BB5) is adjusted, to account for the program's actual runtime progress. For example, suppose the program experiences a delay of 100 ms to reach the checkpoint in BB5, thereby arriving at 600 ms instead of the expected 500 ms. Therefore, the program is delayed by $(\frac{100}{500} \times 100 =)$ 20% from its LO-mode progress.

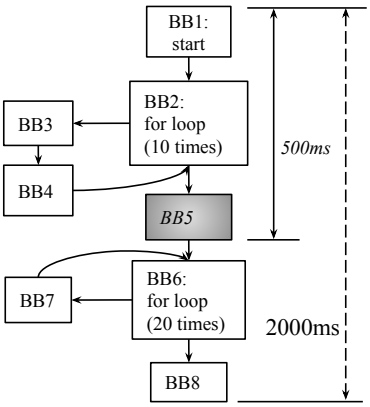


Figure 1 CFG and Average Time Estimates of a Program.

Depending on the relationship between the task's LO- and HI-mode progress to the checkpoint, the task's budget is adjusted according to the 20% observed delay at the checkpoint. One approach is to extrapolate a linear delay from the checkpoint to the end of the task's execution. Thus, the total execution time of 2000 ms is predicted to complete at $(2000 + 20\% \times 2000=)$ 2400 ms. PAStime uses the available information at a checkpoint to extend the LO-mode budget of a high-criticality application. In Section 5.8, we explore other possible execution time prediction models.

2.1 Benefits of Adaptive Mixed-criticality Scheduling

We see progress-aware scheduling as being beneficial in mixed-criticality systems. Adaptive Mixed-criticality scheduling [7] is the state-of-the-art fixed-priority scheduling policy for Mixed-criticality tasksets. In AMC, a system is first initialized to run in *LO-mode*. In LO-mode, all the tasks are executed with their LO-mode execution budgets. Whenever a high-criticality task overruns its LO-mode budget, the system is switched to *HI-mode*. In HI-mode, all low-criticality tasks are discarded (or given reduced execution time [5, 35]), and the high-criticality tasks are given increased HI-mode budgets. The system's switch to HI-mode therefore impacts the QoS for low-criticality tasks.

By combining PAStime with AMC (to yield AMC-PAStime), we extend the LO-mode budget of a high-criticality task to its predicted execution time at a checkpoint. Going back to our previous example in Figure 1, we try to extend the LO-mode budget of the task by 400 ms. We carry out an online schedulability test to determine if we can extend the task's LO-mode budget by 400 ms. If the whole taskset is still schedulable after an extension of the LO-mode budget of the delayed high-criticality task, the increment in the task's LO-mode budget is approved. We let the task run until the extended time and keep the system in LO-mode. In case the high-criticality task finishes within its extended time, we avoid an unnecessary switch to HI-mode. Thus, the low-criticality tasks run for an extended period of time and do not suffer from degraded CPU utilization and QoS, as occurs with AMC.

In case the task does not finish within the predicted time, then the system is changed to HI-mode, and low-criticality tasks are finally dropped. This behavior is identical to AMC, and every high-criticality task still finishes within its own deadline. Therefore, we improve the QoS of the low-criticality tasks when the high-criticality tasks finish within their extended LO-mode budgets, and otherwise, we fall back to AMC. When there is no delay at a checkpoint, we do not change a task's LO-mode budget.

3 Theoretical Background

In this section, we provide a response time analysis for AMC-PAStime by extending the analysis for AMC scheduling. We also describe details about the online schedulability test based on the response time values.

3.1 Task Model

We use the same AMC task model as described by Baruah et al [7]. Without loss of generality, we restrict ourselves to two criticality levels - LO and HI. Each task, τ_i, has five parameters: $C_i(LO)$ - LO-mode runtime budget, $C_i(HI)$ - HI-mode runtime budget, D_i - deadline, T_i - period, and L_i - criticality level of a task, which is either high (HC) or low (LC). We assume each task's deadline, D_i, is equal to its period, T_i. A HC task has two budgets: $C_i(LO)$ for LO-mode assurance and $C_i(HI)$ ($> C_i(LO)$) for HI-mode assurance. A LC task has only one budget of $C_i(LO)$ for LO-mode assurance.

3.2 Scheduling Policy

Both AMC and AMC-PAStime use the same task priority ordering, based on Audsley's priority assignment algorithm [2]. If a task's response time for a given priority order is less than its period, then the task is deemed schedulable. The details of the priority assignment strategy are discussed in previous research work [3, 7]. We do not change the priority ordering of the tasks at runtime.

AMC-PAStime initializes a system in LO-mode with all tasks assigned their LO-mode budgets. We extend a high-criticality task's LO-mode budget at a checkpoint if the task is lagging behind its expected progress, as long as the extension does not hamper the schedulability of the delayed task and all the lower or equal priority tasks. An increase to the LO-mode budget of a task that violates its own or other task schedulability is not allowed. If a high-criticality task has not finished its execution even after its extended LO-mode budget, then the system is switched to HI-mode.

3.3 Response Time Analysis

The AMC response time recurrence equations for (1) all tasks in LO-mode, (2) HC tasks in HI-mode, and (3) HC tasks during mode-switches are shown in Equation 1, 2 and 3, respectively. $hp(i)$ is the set of tasks with priorities higher than or equal to that of τ_i. Likewise, $hpHC(i)$ and $hpLC(i)$ are the set of high- and low-criticality tasks, respectively, with priorities higher than or equal to the priority of τ_i.

AMC provides two analyses for mode switches: AMC-response-time-bound (AMC-rtb) and AMC-maximum. We use AMC-rtb for our analysis, as AMC-maximum is computationally more expensive. However, AMC-rtb does not allow a taskset which is not schedulable by AMC-maximum. Therefore, AMC-rtb is *sufficient* for schedulability.

$$R_i^{LO} = C_i(LO) + \sum_{\tau_j \in hp(i)} \lceil \frac{R_i^{LO}}{T_j} \rceil \times C_j(LO) \tag{1}$$

$$R_i^{HI} = C_i(HI) + \sum_{\tau_j \in hpHC(i)} \lceil \frac{R_i^{HI}}{T_j} \rceil \times C_j(HI) \tag{2}$$

$$R_i^* = C_i(HI) + \sum_{\tau_j \in hpHC(i)} \lceil \frac{R_i^*}{T_j} \rceil \times C_j(HI) + \sum_{\tau_j \in hpLC(i)} \lceil \frac{R_i^{LO}}{T_j} \rceil \times C_j(LO) \quad (3)$$

With AMC-PAStime, we not only measure $C_i(LO)$ and $C_i(HI)$, but also the LO-mode time to reach a checkpoint $C_i^{CP}(LO)$. If a high-criticality task, τ_i, is delayed by $X\%$ at a checkpoint, compared to its LO-mode progress, then τ_i takes $(C_i^{CP}(LO) + \frac{C_i^{CP}(LO) \times X}{100})$ time to reach the checkpoint. Hence, τ_i's budget is tentatively increased from $C_i(LO)$ to $C_i'(LO)$, where $C_i'(LO) = f(C_i(LO), X)$. Here, $f(C_i(LO), X)$ is a function to predict the delayed total execution time, given that the observed delay until the checkpoint is $X\%$. For example, by a linear extrapolation of the observed delay of $X\%$ at a checkpoint, the original budget $C_i(LO)$ changes to $f(C_i(LO), X) = \big(C_i(LO) + \frac{C_i(LO) \times X}{100}\big)$. It is possible for f to depend on task-specific and hardware microarchitectural factors such as cache and main memory accesses. We show more execution time prediction techniques in Section 5.8.

With the increased LO-mode budget $C_i'(LO)$, an online schedulability test then calculates the the extended LO-mode response time, $R_i^{LO\text{-ext}}$, for τ_i, using Equation 4. Similarly, $R_i^{*\text{-ext}}$ is calculated using Equation 5. Equations 4 and 5 are extensions of Equations 1 and 3. AMC-PAStime checks at runtime whether both $R_i^{LO\text{-ext}}$ and $R_i^{*\text{-ext}}$ are less than or equal to τ_i's period to determine its schedulability.

The new response times are then calculated for all tasks in LO-mode with priorities less than or equal to τ_i, using Equation 4. Similarly, new response times are calculated for all HC tasks with lower or equal priority to τ_i during a mode switch, using Equation 5. If all newly calculated response times are less than or equal to the respective task periods, the system is schedulable. In this case, AMC-PAStime approves the LO-mode budget extension to τ_i. If the system is not schedulable, then τ_i's budget remains $C_i(LO)$.

$$R_i^{LO\text{-ext}} = C_i'(LO) + \sum_{\tau_j \in hp(i)} \lceil \frac{R_i^{LO\text{-ext}}}{T_j} \rceil \times C_j'(LO) \quad (4)$$

$$R_i^{*\text{-ext}} = C_i(HI) + \sum_{\tau_j \in hpHC(i)} \lceil \frac{R_i^{*\text{-ext}}}{T_j} \rceil \times C_j(HI) \quad + \sum_{\tau_j \in hpLC(i)} \lceil \frac{R_i^{LO\text{-ext}}}{T_j} \rceil \times C_j(LO) \quad (5)$$

AMC-PAStime only extends the LO-mode budget of a delayed HC task for its current job. When a new job for the same HC task is dispatched, it starts with its original LO-mode budget. If another request for an extension in LO-mode for the same task appears, AMC-PAStime tests the schedulability with the maximum among the requested extended budgets. The system keeps track of the maximum extended budget for a task and uses that value for online schedulability testing. We explain the AMC-PAStime scheduling scheme with an example taskset in Table 1.

Table 1 A Mixed-criticality Taskset Example.

Task	Type	C(LO)	C(HI)	T	Pr	R^{LO}	R^*
τ_1	HC	3	6	10	1	3	6
τ_2	LC	2	-	9	2	5	-
τ_3	HC	5	10	50	3	15	38

Suppose, task τ_1 is delayed by 66% at a checkpoint in the task's source code. PAStime will then try to extend the budget by $(3 \times \frac{66}{100}) \approx 2$ time units. Therefore, the potential extended budget $C_1'(LO)$ for τ_1 would be $(3 + 2) = 5$ time units. PAStime will calculate the

response times, $R_i^{LO\text{-ext}}$ and $R_i^{*\text{-ext}}$, for τ_1 and the lower priority tasks τ_2 and τ_3. $R_1^{LO\text{-ext}}$ would just be 5, and $R_1^{*\text{-ext}}$ would remain the same as $R_1^* = 6$, as τ_1 is the highest priority task. The new $R_2^{LO\text{-ext}}$ would be 7 (by Equation 4) which is smaller than its period of 9. Therefore, τ_2 would still be schedulable if we extend τ_1's LO-mode budget from 3 to 5.

For τ_3, the new $R_3^{LO\text{-ext}}$ and $R_3^{*\text{-ext}}$ would be, respectively, 26 (by Equation 4) and 40 (by Equation 5) which are also smaller than τ_3's period of 50. Therefore, the extended budget of 5 for τ_1 would be approved by AMC-PAStime. In conventional AMC scheduling, the system would be switched to HI-mode if τ_1 did not finish within 3 time units. However, AMC-PAStime will extend τ_1's LO-mode budget to 5 because of the observed delay at its checkpoint, so the system is kept in LO-mode. Consequently, LC task τ_2 is allowed to run by AMC-PAStime, if τ_1 finishes before 5 time units. If τ_1 does not finish even after 5 time units, the system would be switched to HI-mode.

In this example, 5 jobs of τ_1 are dispatched for every single job of τ_3, as τ_3's period of 50 is 5 times the period of τ_1. Suppose τ_1 asks for the 66% increment in its LO-mode budget for the first job, as we have explained above. In its second job, τ_1 asks for an increment of 33% (1 time unit) in its LO-mode budget. In this case, we again need to calculate the response times for all tasks. If we calculate the online response times for τ_2 and τ_3 assuming $(3 + 1) = 4$ time units for $C_1'(LO)$, then we would not account for the first job of τ_1, which potentially executes for 5 time units. We would need to keep track of the extended LO-mode budgets for all jobs of τ_1, to accurately calculate online response times for τ_2 and τ_3.

To avoid the cost of recording all extended LO-mode budgets for a task, AMC-PAStime simply stores the maximum extended budget for a task. When calculating online response times to check whether to approve an extension to the LO-mode budget, the system uses the maximum extended budget of every high-criticality task. This value is stored in the `max_extended_budget` variable for each HC task. Therefore, when τ_1 asks for 4 time units as its extended LO-mode budget in its second job, the system calculates $R_1^{LO\text{-ext}}$, $R_1^{*\text{-ext}}$, $R_2^{LO\text{-ext}}$, $R_3^{LO\text{-ext}}$, $R_3^{*\text{-ext}}$ with $C_1'(LO) = 5$.

AMC-PAStime uses maximum extended budgets to calculate safe upper bounds for online response times. The `max_extended_budget` task property is reset when a task has not requested a LO-mode budget extension for any of its dispatched jobs within the maximum period of all tasks.

3.4 Online Schedulability Test

AMC-PAStime performs an online schedulability test whenever a high-criticality task asks for an extension to its LO-mode budget. The test calculates the response times ($R_i^{LO\text{-ext}}$ and $R_i^{*\text{-ext}}$) of the delayed high-criticality task and all lower priority tasks. Then, it checks whether the response times are less than or equal to the task periods. If any task's response time is greater than its period, the online schedulability test returns false, and the extension in LO-mode for the high-criticality task is denied. If the schedulability test is successful the high-criticality task is permitted to run for its extended budget in LO-mode.

Algorithm 1 shows the pseudocode for the online schedulability test. The offline response time values (R_i^{LO} and R_i^*) are stored in the properties for each task τ_i. When a high-criticality task τ_k is delayed, it asks for an extension of its LO-mode budget by e time units.

Line 6 in Algorithm 1 determines whether the newly requested extension, e, is more than a previously saved maximum extended budget for τ_k. In lines 8–11, the $C_i'(LO)$ is set to the maximum extended budget for all the lower priority tasks and τ_k. As we have explained above, it is practically infeasible to store the execution times of every job of all the tasks to calculate online response times. Therefore, we store the maximum extended budget of

Algorithm 1 Online Schedulability Test.

```
1:  Input: tasks - set of all tasks in priority order
2:         τ_k - delayed task
3:         e - extra budget for τ_k
4:  Output: true or false
5:  function IsBudgetChangeApproved(tasks, τ_k, e)
6:      e' = max(τ_k.max_extended_budget, C_k(LO) + e) − C_k(LO)
7:      for each task τ_i in {τ_k ∪ lower priority tasks than τ_k in tasks} do
8:          C'_i(LO) = τ_i.max_extended_budget
9:          if τ_i is τ_k then
10:             C'_i(LO) =max(C'_i(LO), C_i(LO) + e)
11:         end if
12:         Initialize R_i^{LO-ext} for Equation 4 with R_i^{LO} + e'
13:         Solve Equation 4
14:         if R_i^{LO-ext} ≤ T_i then
15:             if τ_i is high-criticality then
16:                 Initialize R_i^{*-ext} for Equation 4 with R_i^*
17:                 Solve Equation 5 with the new R_i^{LO-ext}
18:                 if R_i^{*-ext} > T_i then  Return false
19:                 end if
20:             end if
21:         else Return false
22:         end if
23:     end for
24:     τ_k.max_extended_budget = max(C_k(LO) + e, τ_k.max_extended_budget)
25:     Return true
26: end function
```

every task and use this to calculate response times online. For the currently delayed task, τ_k, $C'_k(LO)$ is set to the maximum value between a previously saved `max_extended_budget` and the currently requested $C_k(LO) + e$. Considering the example in Table 1 from the paper, line 10 would translate to $C'_i(LO)$=max(5, 4) for the second LO-mode extension request.

Then, Equation 4 is solved in line 13 by initializing $R_i^{LO\text{-ext}}$ to the R_i^{LO} plus extra budget e' from line 6. If a lower priority task is a high-criticality task, then $R_i^{*\text{-ext}}$ is calculated in line 17, with the newly derived value of $R_i^{LO\text{-ext}}$.

Online response time calculations may take significant time, depending on the number of iterations of Equations 4 and 5. However, the response time values from Equations 1, 2, 3 are already calculated offline to determine the schedulability of a taskset using Audsley's priority assignment algorithm [2] with AMC scheduling. Since AMC-PAStime uses the same priority ordering as AMC, the offline response times for schedulability remain the same.

AMC-PAStime initializes the online $R_i^{LO\text{-ext}}$ in Equation 4 with $R_i^{LO} + e$, where R_i^{LO} is calculated offline by Equation 1 and $e(> 0)$ is the extra budget of the delayed task.

Since the budget of a delayed task is extended by e, $R_i^{LO\text{-ext}}$ must be greater than or equal to $R_i^{LO} + e$. Hence, this is a good initial value to start calculating $R_i^{LO\text{-ext}}$ online. In Section 5, we establish an upper bound on the total number of iterations needed to check the schedulability of random tasksets, if the highest priority task's budget is extended by different amounts.

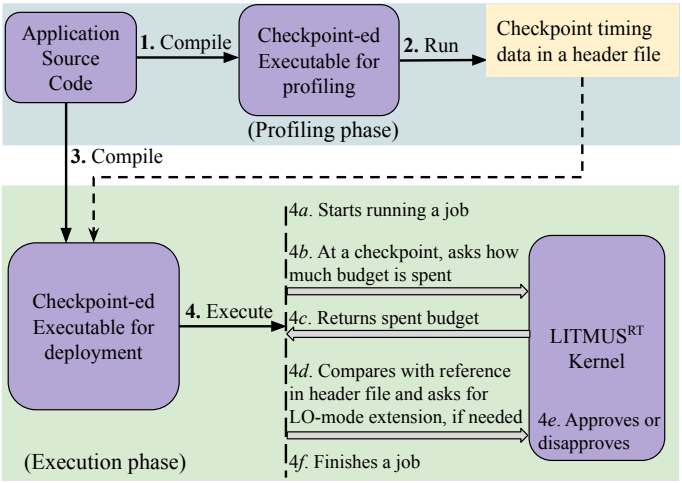


Figure 2 Implementation of AMC-PAStime in LITMUS^{RT}.

4 Design and Implementation of PAStime

In this section, we first describe the overall design of AMC-PAStime in LITMUS^{RT} [10, 14]. This is followed by a description of how checkpoints are instrumented in an application's source code. We then show the algorithm to determine and insert checkpoints, which is integrated into the LLVM compiler infrastructure. A high-criticality task requires profiling to determine the placement of checkpoints, before it is ready for execution with other tasks. We describe how the profiling and execution of a high-criticality task is performed, along with the scheduling mechanism in PAStime. The source code for PAStime in LITMUS^{RT} is publicly available [41].

AMC-PAStime has two phases: a Profiling phase for high-criticality tasks, and an Execution phase. In the Profiling phase, one or more checkpoints are placed at key stages in a program's source code. The *average* time to reach each checkpoint is then measured. After profiling all high-criticality tasks, the system switches into the Execution phase. The time taken to reach each checkpoint in every high-criticality task is observed by the system at runtime. Each observed time is compared against the profiled time to reach the same checkpoint. Any high-criticality task lagging behind its profiled time to a checkpoint is tentatively given increased LO-mode budget, according to the approach described in Section 3.

Figure 2 shows an overview of the design of AMC-PAStime in LITMUS^{RT}. Step 1 is the compilation of a high-criticality application's source code in the Profiling phase, which uses our compilation procedure [41]. The compiled executable has checkpoints embedded into its code for profiling. Step 2 executes the program to generate timing metadata for each checkpoint in a *timeinfo.h* file. Step 2 is performed multiple times with different program inputs to generate an average time to reach each checkpoint.

Step 3 compiles the source code along with the checkpoint timing metadata header file (*timeinfo.h*) for the Execution phase, producing a binary image that is used for deployment under working conditions. Finally, Step 4 runs the code in the Execution phase along with all other tasks. At some point after the system is started, Step 4a starts running a job for a high-criticality task. When a checkpoint is reached, Step 4b asks the LITMUS^{RT} kernel how much budget it has consumed. Step 4c returns the spent budget from the LITMUS^{RT} kernel to the application.

After receiving the spent budget, t_{spent}, the application compares it to the reference timing, t_{ref}, for the checkpoint from the *timeinfo.h* header file. It calculates the extra budget, e, needed in LO-mode, using an execution time prediction model as described later in Section 5.8. If $e > 0$, Step 4d asks LITMUS$^{\text{RT}}$ for extra budget. The AMC-PAStime scheduling policy in the LITMUS$^{\text{RT}}$ kernel runs an online schedulability test. If the test returns true, the LO-mode budget of the current task is extended by e. If the test algorithm returns false, the task budget is not altered. Finally, Step 4f finishes the current job of the running task.

4.1 Checkpoint Instrumentation and Detection

A checkpoint is a key stage in an application's code, used to evaluate the progress of a currently running task. Well placed checkpoints balance the number of instructions that are executed prior to the checkpoint, with those that remain to the next checkpoint or the end of the program. Ideally, there should be a meaningful number of instructions leading up to a checkpoint to determine progress. Likewise, there should be sufficient instructions after a checkpoint to increase the likelihood that a task is adequately compensated for execution delays using an extended budget.

A developer of a high-criticality application finds a potential checkpoint location in the program's source code for its Execution phase, after trying out multiple locations in the Profiling phase. PAStime includes a development library to instrument checkpoints for the two different phases. Additional modifications to the LLVM compiler [29] are used to automatically detect and instrument checkpoints in the Profiling phase.

4.2 Checkpoint Library

We have developed a C library to instrument checkpoints in the two PAStime phases. The main purpose of the library is to generate the necessary checkpoint timing information during the Profiling phase and then request a task's extended LO-mode budget from the LITMUS$^{\text{RT}}$ kernel during the Execution phase.

In the Profiling phase, a `writeTime` function call from our library is inserted into the application code at a desired checkpoint. `writeTime` takes a unique ID for each checkpoint. The function logs the time to reach that checkpoint in the source code since the start of a job. The average of multiple such timing entries is saved in *timeinfo.h* after the Profiling phase.

During the Execution phase, a preprocessor macro called `ANNOUNCE_TIME` is inserted at a checkpoint. This macro obtains the spent budget from the LITMUS$^{\text{RT}}$ kernel via a `get_current_budget` system call. Then, it compares the spent budget with the reference budget from the *timeinfo.h* header file, and calculates the extra budget using an execution time prediction model. If extra budget is needed, the macro makes a `set_rt_task_param` LITMUS$^{\text{RT}}$ system call.

4.3 Manual Checkpoint Instrumentation

A developer may attempt various strategies to identify a key stage [13,22,50] of an application to instrument a checkpoint. The developer uses either the `writeTime` function for the Profiling phase, or the `ANNOUNCE_TIME` macro for the Execution phase to instrument a checkpoint. Checkpoints should generally be avoided inside tight loops. Visiting a checkpoint every loop iteration incurs a small overhead that is accumulated across multiple iterations.

Algorithm 2 Determine and Insert Checkpoints.

```
1: isLoopBefore: identifies if a loop is in the paths from the starting BB to another BB
2: visited: set of already visited BBs in DFS
3: function INSERTCHECKPOINT(function)
4:     startingBB = function.getEntryBlock()
5:     DODFS(startingBB)
6: end function
7: function DODFS(currentBB)
8:     if currentBB is in visited then  return
9:     end if
10:    LoopID = getLoopFor(currentBB)
11:    if LoopID != null then
12:        if isLoopBefore[currentBB] ∧ isLoopHeader(currentBB) then
13:            insert checkpoint before currentBB
14:        end if
15:    end if
16:    insert currentBB in visited
17:    for each s in successors of currentBB do DODFS(s)
18:    end for
19: end function
```

4.4 Automatic Checkpoint Instrumentation

We have written a compiler pass in LLVM to automatically instrument checkpoints for the Profiling phase of PAStime. The instrumented code is run in the Profiling phase, and multiple checkpoint timing information is generated. Finally, the developer chooses one such checkpoint for the Execution phase.

The compiler pass automatically inserts checkpoints in the basic block preceding each loop in a function, except the first loop. The first loop is excluded so that enough instructions are executed before a checkpoint to determine meaningful delays.

For nested loops, we consider only the outer loop. Automatic instrumentation works with only simple program structures and ignores intersecting loops. LLVM's `LoopInfo` analysis identifies only natural loops [37]. We utilize the `LoopInfo` class in our checkpoint instrumentation implementation.

Algorithm 2 identifies checkpoint locations for the Profiling phase. It starts a Depth First Search (DFS) from the starting Basic Block (BB) of a function, by calling `DoDFS` in line 6. The algorithm then checks whether a BB is part of a loop using `LoopInfo`'s `getLoopFor` member function. This function returns a unique `LoopID` for every new loop. A nested inner loop has the same ID as its outer loop. If a BB is not part of a loop, then it returns `null`.

If a BB is part of a loop, line 12 first checks whether there is any loop before the current loop using a dictionary `isLoopBefore`. `isLoopBefore` is pre-populated for every BB in the CFG to indicate whether there is at least one loop seen in the paths from the starting BB to the current BB. To pre-populate `isLoopBefore`, the algorithm checks whether there is a path to a BB from the loop BBs.

Algorithm 2 then verifies that the current BB is a header of a loop using `LoopInfo`'s `isLoopHeader` member function. A header is the entry-point of a natural loop. We insert a checkpoint in the predecessor BB of a header.

Finally, if there is at least one loop before the current header BB, a checkpoint is added before the current BB in line 13. As natural loops have only one header, a checkpoint is avoided for any inner loop within a nested loop. We continue the DFS by marking the current BB as visited.

We have implemented the above algorithm in a compiler pass within LLVM [29], to automatically detect and instrument appropriate function calls as checkpoints in a C language program. This uses our previously described Checkpoint library for the Profiling phase. A developer uses our modified LLVM compiler with their high-criticality application written in C. Our compiler pass operates at the LLVM Intermediate Representation (IR) level. It takes a piece of IR logic as input, figures out the points of interest according to the above algorithm for checkpoints in a particular function, and generates the instrumented IR. These IRs are compiled into executable machine code. As the compiler pass operates at the IR level, it is easily extensible to other high-level languages and back-ends supported by LLVM.

4.5 Profiling and Execution Phases

The Profiling phase of PAStime determines viable checkpoints for use in the Execution phase, and also the LO- and HI-mode budgets for a high-criticality application. A checkpointed program is run multiple times in the Profiling phase against a set of test cases. The program is allowed to have multiple test checkpoints, which are either generated automatically using our modified LLVM compiler, or manually by a developer. Each checkpoint has a unique ID (function ID, BasicBlock ID) given to the checkpoint function call `writeTime`. A Python package then runs the Profiling phase to collect the average times (LO-mode) to reach different checkpoints.

Step 4 in Figure 2 is the start of the Execution phase. One checkpoint from those generated in the Profiling phase for a high-criticality application is instrumented with an `ANNOUNCE_TIME` macro. Although in general it is possible to use multiple checkpoints within the same application in the Execution phase, our experience shows that one is sufficient to improve LO-mode service. Multiple checkpoints add overhead to the task execution. Moreover, later checkpoints account for execution delays that make earlier checkpoints redundant, as long as they are reached before the LO-mode budget expires.

The key issue in deciding on a single checkpoint for the Execution phase is to ensure it is not placed too late in the instruction stream. If it is placed too far into the program code a mode switch may occur before the task's LO-mode budget is extended due to delays. We show in Section 5.6 the effects of using checkpoints at different locations in a program's code.

4.6 LITMUS^{RT} Implementation

As a first step to applying PAStime for use in adaptive mixed-criticality scheduling, we extended LITMUS^{RT} with AMC support. This required modifications to the existing partitioned fixed-priority scheduling policy, to include the following new variables in the task properties: `c_lo`, `c_hi`, `r_lo`, `r_star`, `c_extended`, `max_extended_budget`.

Tasks are divided into low- and high-criticality classes. By default, the HI-mode budget for low-criticality tasks, `c_hi`, is set to zero. If desired, we also support a reduced, non-zero HI-mode execution budget for low-criticality tasks, as discussed in the work on Imprecise Mixed-criticality scheduling [35]. Task priorities are determined offline, along with all response times for a system operating in LO-mode (`r_lo`) and during a mode switch (`r_star`). These parameters are then initialized in the kernel when the system starts executing a set of tasks.

The system is started in LO-mode, with all tasks assigned their `c_lo` budgets. Whenever a high-criticality task is out of its LO-mode budget, an enforcement timer handler (based on Linux's high-resolution timer [33]) is fired, and the system is switched to HI-mode.

For AMC-PAStime, `c_extended` is the extended LO-mode budget for a delayed job of a high-criticality task. An enforcement timer handler is therefore triggered only when a high-criticality task is still unfinished after the depletion of it `c_extended` time.

As Baruah et al. suggest [7], an AMC system switches back to LO-mode when none of the high-criticality tasks have been running for more than their LO-mode budgets. In the case of AMC-PAStime, the system will switch to a lower criticality level when none of the high-criticality tasks have been running for more than their extended LO-mode budgets. A list is used to keep track of which high-criticality tasks complete within their (extended) LO-mode budgets, to determine when to revert to a lower system criticality level.

A task's LO-mode budget is extended by AMC-PAStime by making a `set_rt_task_param` system call inside the `ANNOUNCE_TIME` macro. We have implemented the `task_change_params` callback of a LITMUS^{RT} `sched_plugin` interface to support runtime adjustment of task parameters. In the `task_change_params` callback, the system runs an online schedulability test. If the test returns *true*, the budget extension is approved, and the enforcement timer for the current task is adjusted accordingly. The task's `c_extended` variable is updated, along with the maximum value of its extended budget in `max_extended_budget`.

It is worth noting that a budget extension could be delayed until a mode switch is about to happen. However, this strategy needs the scheduler to save the delay at a checkpoint and later decide about the budget extension at a mode switch point. This approach potentially increases runtime overhead while repeatedly accumulating task delays. Conversely, redundant budget extensions in PAStime are avoided by following a lazy budget extension approach. Such strategies are open to further study.

5 Evaluation

We tested our implementation of AMC and AMC-PAStime in LITMUS^{RT} with real-world applications on an Intel NUC Kit [24]. The machine has an Intel Core i7-5557U 3.1GHz processor with 8GB RAM, running Linux kernel 4.9. We use three applications in our evaluations: a high-criticality object classification application from the Darknet neural network framework [44], a high-criticality object tracking application from dlib C++ library [16], and a low-criticality MPEG video decoder [20]. These applications are chosen for their relevance to the sorts of applications that might be used in infotainment and autonomous driving systems. The dlib object tracking application is only used for the last set of experiments to test two different execution time prediction models.

For object classification, we use the COCO dataset [32] images for both profiling and execution. The dataset for the Profiling phase was chosen from random images from the dataset and used to determine the LO- and HI-mode budgets of the Darknet high-criticality application. The dataset for the Execution phase was also randomly chosen but similarly distributed in terms of image dimensions as the data of the Profiling phase was. For the video decoder application, we use the *Big Buck Bunny* video [8] as the input. We have turned off memory locking with `mlock` by the `liblitmus` library, as multiple object classification tasks collectively require more RAM than the physical 8GB limit.

5.1 Task Parameters

Table 2 shows the LO- and HI-mode budgets for the three applications. Each object classification and tracking task consist of a series of jobs that classify objects in a single image. Each video decoder task decodes 30 frames in a single job.

Table 2 Applications and their Budgets.

Application	C(LO)	C(HI)
Darknet object classification	345 ms	627 ms
dlib object tracking	10.7 ms	18 ms
video decoder	250 ms	-

The LO-mode budget, $C(LO)$, is the average time that a task takes to complete its job. In Section 5.5, we also present experiments where we increase our LO-mode budget estimate. The HI-mode budget, $C(HI)$, accounts for the worst-case running time of the high-criticality task for any of its jobs. The LO-mode utilization of each individual task is generated by the UUnifast algorithm [9]. We then calculate a task's period by dividing its LO-mode budget by its utilization.

In all experiments, we observed that none of the tasks exceed their HI-mode budget. Consequently, none of the high-criticality tasks miss their deadlines in any of our tests. Hence, we assume that our derivation of LO- and HI-mode budgets are safe and correct for our experiments. The main focus of our evaluation now is to compare the QoS of the LC tasks.

5.2 QoS Improvements for Low-criticality Tasks

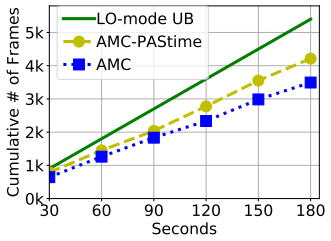
We compare the QoS for low-criticality tasks using AMC and AMC-PAStime in different cases. In every case, each taskset has an equal number of high-criticality object classification tasks and low-criticality video decoder tasks. We experiment with ten schedulable tasksets in all cases except the base case described below. We run each of the tests ten times, and we report the average of the measurements for low-criticality tasks. As stated earlier, all high-criticality tasks meet their timing requirements in each case. Execution time prediction at a checkpoint for PAStime is done with a linear extrapolation model for all the experiments ($f(C_i(LO), X) = C_i(LO) + \frac{C_i(LO) \times X}{100}$ from Section 3) except in Section 5.8 where we discuss other approaches.

Our base case is to run one high-criticality object classification task and one low-criticality video decoder task for 180 seconds. Here, we set the periods of both tasks to 1000 ms rather than using the UUnifast algorithm, yielding a total LO-mode utilization of ~60%.

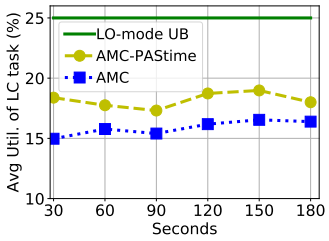
Figure 3a shows the cumulative number of frames decoded by the video decoder task. The LO-mode Upper Bound (UB) line shows the cumulative number of decoded frames if the system is kept in LO-mode all the time. This line represents a theoretical UB for a decoded frame playback rate of 30 frames per second over the entire experimental run.

We see in Figure 3a that AMC-PAStime has a 9–21% increase in the cumulative number of decoded frames compared to AMC scheduling. The performance of the low-criticality task is related to the number of HI-mode switches in the two scheduling policies. AMC-PAStime decreases the number of HI-mode switches by 35%, compared to AMC scheduling.

Although the number of decoded frames is an illustrative metric for a video decoder's QoS, the average utilization of an application is a more generic metric. Average utilization represents the CPU share a task receives over a period of time. Figure 3b shows that AMC-



(a) Cumulative # of Frames.



(b) Average Utilization.

Figure 3 Video Decoder Performance.

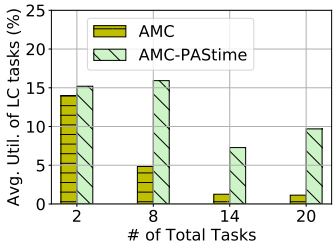


Figure 4 Varying # of Tasks.

Table 3 Number of Mode Switches.

(a) Varying Number of Tasks.

# of tasks	AMC	AMC-PAStime
2	4	2
8	9	4
14	7	5
20	5	3

(b) Varying Initial LO-mode Utilization.

Utilization (%)	AMC	AMC-PAStime
40	11	4
50	11	4
60	9	4
70	10	5
80	11	9

PAStime achieves 10% more utilization on average for the video decoder task, compared to AMC scheduling. The LO-mode UB line is the maximum utilization of the video decoder task, which is 25% (i.e., $C(LO)/Period$ = 250 ms/1000 ms).

5.3 Scalability

To test system scalability, we increase the number of tasks in a taskset up to 20 tasks. As explained above, we generate the periods of the tasks by distributing the total LO-mode utilization of ~60% to all the tasks using the UUnifast algorithm. This setup is inspired by the theoretical parameters in previous mixed-criticality research work [7]. The LO-mode utilization bound for low-criticality tasks remains between 25–35%.

Figure 4 shows the average utilization of the low-criticality tasks, when the total tasks vary from 2 to 20. Each task in this case consists of 20 jobs. We see that the average utilization drops for AMC scheduling as the number of tasks increases.

AMC-PAStime achieves significantly greater average utilization for the low-criticality tasks, by deferring system switches to HI-mode until much later than with AMC scheduling. This is because the LO-mode budgets for the high-criticality tasks are extended due to runtime delays.

AMC-PAStime decreases the number of mode switches by 28–55%. AMC-PAStime's resistance to switching into HI-mode allows low-criticality tasks to make progress. This in turn improves their QoS. In these experiments, AMC-PAStime improves the utilization of the low-criticality tasks by a factor of 3, 5 and 9, respectively, for 8, 14 and 20 tasks. Table 3a shows that AMC-PAStime reduces the number of mode-switches compared to AMC.

5.4 Varying the Initial Total LO-mode Utilization

In this test, we vary the initial total LO-mode utilization for 8 tasks from 40% to 80% by adjusting the periods of all tasks. The initial utilization does not account for increases caused by LO-mode budget extensions to high-criticality tasks.

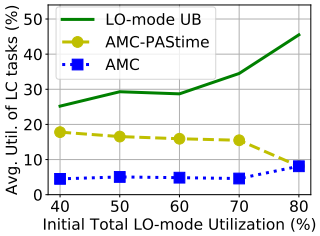


Figure 5 Varying LO-mode Utilization.

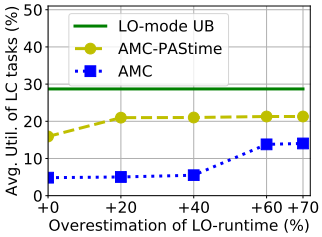


Figure 6 Overestimated $C(LO)$.

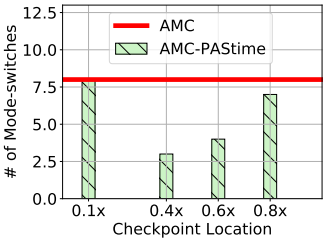


Figure 7 Checkpoint Location.

Figure 5 demonstrates that AMC-PAStime improves average utilization of the low-criticality tasks by more than 3 times, up to 70% total LO-mode utilization. After that, AMC-PAStime and AMC scheduling converge to the same average utilization for low-criticality tasks. This is because there is insufficient surplus CPU time in LO-mode for AMC-PAStime to accommodate the extended budget of a high-criticality task. Therefore, the LO-mode extension requests are disapproved by AMC-PAStime. The reduced number of mode switches for AMC-PAStime, shown in Table 3b, also corroborates the rationale behind AMC-PAStime's better performance than AMC.

We note that the schedulability of random tasksets decreases with higher LO-mode utilization in AMC scheduling. Therefore, many real-world tasksets may not be schedulable because of their HI-mode utilization. Thus, AMC-PAStime's improved performance is significant for practical use-cases.

5.5 Estimation of LO-mode Budget

In our evaluations until now, we estimate a LO-mode budget based on the average execution time of the high-criticality object classification application. In the next set of experiments, we estimate the LO-mode budget of a high-criticality task to be a certain percentage above the average profiled execution time. An increased LO-mode budget for high-criticality tasks benefits AMC scheduling. This is because high-criticality tasks are now given more time to complete in LO-mode, and therefore low-criticality tasks will still be able to execute as well. As a result, the utilization of low-criticality tasks is able to increase.

Suppose that $C(LO)$ is an average execution time estimate for the LO-mode budget of a high-criticality task. Let $(C(LO) + o)$ be an overestimate of the LO-mode budget. As before, AMC-PAStime detects an $X\%$ lag at a checkpoint and predicts the total execution time to be $C(LO) + e$. If the actual LO-mode budget is $(C(LO) + o)$ then AMC-PAStime requests for an extra budget of $(e - o)$, assuming $(e - o) > 0$.

Even for overestimated $C(LO)$, Figure 6 shows that AMC-PAStime still improves utilization for the low-criticality tasks by more than a factor of 3 up to an overestimation of 40%. Overestimation helps in reducing the number of mode switches for AMC scheduling after 40%, as high-criticality tasks have larger budgets in LO-mode.

There is no improvement by AMC scheduling after 60% LO-mode budget overestimation. AMC-PAStime also shows no benefits with increased overestimation, because the LO-mode budget extensions are disapproved by the online schedulability test. Therefore, the system is switched to HI-mode by an overrun of a high-criticality task.

5.6 Checkpoint Location

We use our modified LLVM compiler in the Profiling phase to determine a viable checkpoint for the high-criticality object classification task. We instrument checkpoints in the

`forward_network` function of the Darknet neural network module. We consider four checkpoint locations in the Profiling phase, which are both automatically and manually instrumented. In the Execution phase, we measure performance for each of these checkpoint locations. Figure 7 shows the variation in the number of mode switches against the location of a checkpoint. The x-axis is the approximated division point of a checkpoint location with respect to $C(LO)$. For example, $0.1\times$ means that the checkpoint is at $(0.1 \times C(LO))$.

We see that the number of mode switches decreases if the location of a checkpoint is more towards the middle of the code. However, a checkpoint near the start and the end of the source code have nearly the same number of mode switches, as with AMC scheduling. A checkpoint near the beginning of a program is not able to capture sufficient delay to increase the LO-mode budget enough to prevent a mode switch. Likewise, a checkpoint near the end of a program is often too late. A HI-mode switch may occur before the high-criticality task even reaches its checkpoint. Hence, a checkpoint at $0.8\times$ in the source code of a program reduces the number of mode switches by just 1.

5.7 Overheads

The main overheads of AMC-PAStime compared to AMC are the online schedulability test and budget extension. We first derive an upper bound on the overhead by offline analysis and compare with the experimental measurements.

Our offline upper bound is the total number of iterations in solving the response time recurrence relations during the schedulability test in AMC-PAStime. We generate 500 random tasksets of 20 tasks for different initial LO-mode utilizations. Initial LO-mode utilizations range from 40% to 90%. The utilization of each individual task is generated using the UUnifast algorithm, and each period is taken from 10 to 1000 simulated time units, as done in previous work [7, 23]. As our experimental taskset has a criticality factor ($\text{CF} = \frac{C(HI)}{C(LO)}$) of ~1.8, we also test with a CF of 1.8.

Among the schedulable tasks with AMC scheduling, we increase the demand in LO-mode budget of the highest priority task. Then, we calculate the total number of iterations needed to determine whether an extension of the budget can be approved by an offline version of the online schedulability test. Here, one iteration is a single update to the response time in any one of the recurrence equations (in Equations 4 and 5) used to test for schedulability.

We increase the demand for extra budget from 10 to 80% in this analysis because the CF is 1.8. We check the maximum number of iterations to decide the schedulability of a taskset across the 500 tasksets. We carry out this offline analysis with 40-90% initial LO-mode utilization.

In Figure 8, we show the maximum number of iterations to decide the schedulability of a taskset, against a demand of 10 to 80% extra budget in LO-mode by the highest priority task. Each point in the figure represents the maximum iterations across the 500 tasksets to either approve or disapprove of schedulability.

We have observed the number of iterations to be as high as 120. Therefore, we set 120 as the highest number of allowed iterations for the online schedulability test. When the number of iterations exceed 120 at runtime, we disapprove a LO-mode budget extension. This strategy maintains a safe and known upper bound on the online overhead of AMC-PAStime.

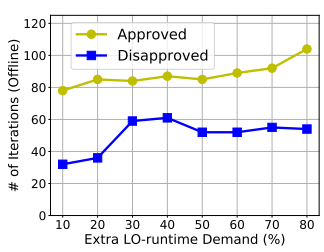


Figure 8 Offline Iterations.

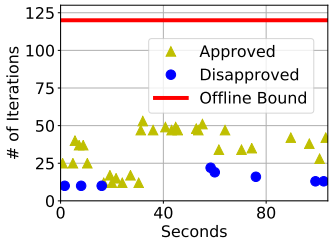


Figure 9 Online Iterations.

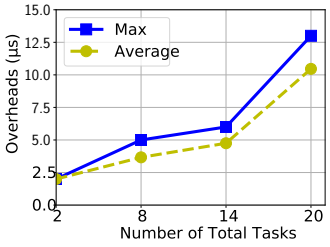


Figure 10 Budget Extension.

5.7.1 Microbenchmarks

Each iteration of a response time calculation has a worst-case time of 1μs, for our implementation of the online schedulability test in LITMUS$^{\text{RT}}$. Therefore, we bound the worst-case delay for the online schedulability test in LITMUS$^{\text{RT}}$ at 120μs for our test cases. Additionally, the worst-case execution time for the `ANNOUNCE_TIME` macro is 10μs. Hence, the maximum total overhead of an `ANNOUNCE_TIME` call, accounting for the schedulability test, is 130μs. The 130μs overhead is factored into the LO-mode extension inside the LITMUS$^{\text{RT}}$ kernel.

Figure 9 shows the number of iterations for the online schedulability test for a taskset of 20 tasks with 60% initial total LO-mode utilization, in cases where an extension was approved and disapproved. We see that the offline bound of 120 iterations is much higher than the actually observed number of iterations. Hence, we never need to abandon the schedulability test because of excessive overheads. In addition, disapproval takes less time than approval online, which corroborates our offline observations. In addition, disapproval takes less time than approval online, which corroborates our offline observations in Figure 8.

Figure 10 shows the maximum and average times for a LO-mode budget extension decision including the online schedulability test. It demonstrates that the extension approval decision takes more time with increasing number of tasks. However, the maximum times are still significantly lower than the offline upper bound of 130μs for 20 tasks. In general, budget extension overheads can be bounded according to the number of tasks in the system.

5.8 Execution Time Prediction Model

As we have explained in Section 3, the execution time after a checkpoint is predicted by a function $f(C_i(LO), X)$, where $C_i(LO)$ is the LO-mode budget, and $X\%$ is the observed delay percentage relative to the LO-mode time to reach the checkpoint. The parameter X in f is a *timing progress metric*, which is used to make runtime scheduling decisions in PAStime.

5.8.1 Linear Extrapolated Delay

We have already shown in the previous experiments how a straightforward linear extrapolated delay improves the utilization of LC tasks compared to AMC for an object classification application. In such a case, $f(C_i(LO), X) = C_i(LO) + \frac{C_i(LO) \times X}{100}$. A linear extrapolation of delay at a checkpoint applied to the entire LO-mode time makes sense in the absence of additional knowledge that could influence the remaining execution time of the task.

We now investigate further whether the linear extrapolation is effective in detecting the amount of delay in our Darknet high-criticality object classification task. We compare the predicted and actual times taken by the high-criticality tasks when LO-mode is extended by

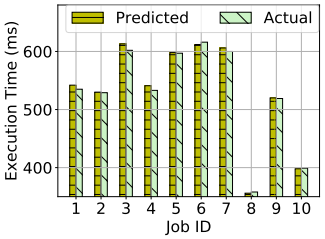


Figure 11
Prediction Accuracy.

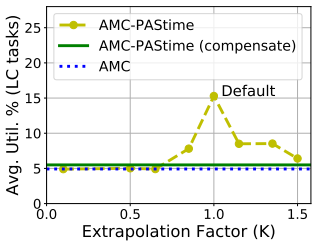


Figure 12
Linear Extrapolation.

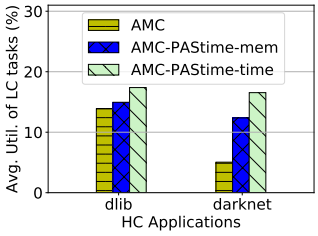


Figure 13
Prediction Models.

AMC-PAStime. This experiment is performed with 8 tasks for 40 jobs each. The initial total LO-mode utilization is 60%, before applying budget extensions.

We show ten of the extended jobs in Figure 11. We see that the predicted execution times are close to the actual execution times. For cases where the predicted times exceed the actual budget expenditure, the predictions overestimate execution by just 0.88% on average for this experiment. In Figure 11, Job ID 6, 8 and 10 show lower predicted times than the actual spent budgets. For these jobs, the system is switched to HI-mode because the extended LO-mode is not enough for a task to complete its job. In these cases, the predicted times are smaller than the spent budgets by 0.49%.

This experiment shows that the checkpoint is effectively being used to predict the execution time of a high-criticality task in most cases. The budget extensions are a reasonable estimate of the actual task requirements.

5.8.2 Alternative Execution Time Compensation Models

In our experiments, we compare the linear extrapolation model with an alternative compensatory execution time prediction model. The alternative approach compensates for the observed delay at a checkpoint, by adding the delay amount to the predicted execution time. If the observed delay is Y, then the estimated total execution time is $C_i'(LO) = C_i(LO) + Y$.

We have also investigated variations to the linear extrapolation model, even though it has proven effective in our previous experiments. A further experiment multiplies the extrapolated delay by a factor K, such that the predicted execution time is $f(C_i(LO), X) = C_i(LO) + K \times \frac{C_i(LO) \times X}{100}$.

Figure 12 compares AMC-PAStime with the alternative compensatory model `compensate`, and linear extrapolation model where K ranges from 0.1 to 1.5. The linear extrapolation model ($K = 1$) outperforms the other approaches, while the compensatory model improves the utilization slightly compared to AMC.

5.8.3 Prediction based on Memory Access Time

Memory accesses by different processes compete for shared cache lines and consequently cause unexpected microarchitectural delays. There are previous works that model the memory [38, 42, 56] and cache [53, 55] accesses in a multicore machine to deal with the issue of predictable execution. Here, we demonstrate PAStime with an execution time prediction model based on the number of memory accesses, to increase the LO-mode budget of a high-criticality task.

In this model, we note the number of memory accesses by a high-criticality task as we measure the LO- and HI-mode execution time of the task during the Profiling phase. We

measure the average number of memory accesses in LO-mode for each profiled run of a task with different inputs, and denote it by $M(LO)$.

During the Profiling phase, we also measure the number of memory accesses in LO-mode before and after a checkpoint: $M_{\text{pre\_}CP}(LO), M_{\text{post\_}CP}(LO)$. These values are also averaged across all runs of a given task. Therefore, $M_{\text{pre\_}CP}(LO) + M_{\text{post\_}CP}(LO) = M(LO)$

We define a new *memory instructions progress metric* to be the ratio of task execution time to the number of memory accesses. The LO-mode value of this metric is $Pr_{mem,LO} = \frac{C(LO)}{M(LO)}$.

In the Execution phase, we measure the number of memory accesses and the used budget up to a checkpoint, respectively denoted by M_{CP} and C_{used} for a given task. During Execution, we calculate the memory instructions progress metric at a checkpoint, $Pr_{mem,CP} = \frac{C_{used}}{M_{CP}}$. If $Pr_{mem,CP} \leq Pr_{mem,LO}$, we predict the task is progressing as expected in LO-mode in terms of memory access-related delays. There are other factors such as I/O-related delays that affect the execution time of a task. However, I/O should be budgeted separately from the main task, as shown in prior work for AMC [39]. Notwithstanding, PAStime is capable of supporting even richer prediction models based on a multitude of factors that cause delays due to microarchitectural and task interference overheads.

When $Pr_{mem,CP} > Pr_{mem,LO}$, we predict a task's memory access delays. We calculate the expected number of memory instructions after a checkpoint: $M_{\text{expected\_post\_CP}} = \frac{M_{CP} \times M_{\text{post\_CP}}(LO)}{M_{\text{pre\_}CP}(LO)}$, and increase the LO-mode budget by $(M_{\text{expected\_post\_}CP} \times (Pr_{mem,CP} - Pr_{mem,LO}))$. This increment helps cover future memory delays after a checkpoint.

We tested both high-criticality dlib object tracking and Darknet object classification applications with this model. Figure 13 shows experimental results with 4 HC tasks (either dlib or Darknet) and 4 LC video decoder tasks, with initial 50% LO-mode utilization. We see that `AMC-PAStime-mem` with this memory access progress metric provides better utilization to the LC tasks than AMC. However, it is not better than `AMC-PAStime-time`, which uses the default linear extrapolated execution time prediction. Nevertheless, the result shows that some form of progress metric dynamically improves the utilization of LC tasks.

6 Related Work

The problem of determining tight worst-case execution time (WCET) bounds for tasks [54] is compounded by timing variations caused by caches, buses and other hardware features. Recently, mixed-criticality systems (MCSs) [5–7,12,15,52] have gained popularity as they allow tasks to have multiple estimates of execution time at different criticality, or assurance, levels. Baruah et al. proposed Adaptive Mixed Criticality (AMC) as a fixed-priority scheduling policy for mixed-criticality systems [7]. AMC dominates other fixed-priority scheduling schemes, such as Static Mixed-criticality with Audsley's priority assignment and Period Transformation for random tasksets [7, 23].

AMC scheduling affects the QoS of low-criticality tasks by dropping them in HI-mode. Further research work explored adjustments to the task model, including stretching the periods [21, 46–49], and using reduced budget in HI-mode [4, 5, 35, 43], to provide improved service to the low-criticality tasks. AdaptMC employs control-theoretic feedback to manage the budgets of dual-criticality workloads [40]. In contrast, PAStime adjusts the budget of a currently running task based on the observed delay at a checkpoint, as we want a simple budget adjustment mechanism to minimize the runtime overhead. However, future work may explore integrating control-theoretic feedback, such as the one used by AdaptMC, into PAStime's runtime strategy.

Santy et al. proposed the idea of a statically-calculated task allowance [45], for the theoretical modeling and scheduling of mixed-criticality tasks. In contrast to Santy's work, PAStime dynamically decides whether a task is given extra budget in LO-mode based on the

observed runtime delay at a checkpoint. PAStime is then able to decide which task's budget to extend in LO-mode, given the slack in computational resources [15].

The work by Kritikakou et al. uses run-time monitoring and control in mixed-criticality systems, to increase task parallelism [26–28]. The authors run high- and low-criticality tasks together and monitor high-criticality tasks at multiple *observations points* embedded into their control flow graphs. If interference from the low-criticality tasks is too prohibitive for the high-criticality tasks, low-criticality tasks are stopped to ensure that the high-criticality tasks meet their deadlines. The authors use static execution time analysis to decide whether to run the low-criticality tasks after an observation point in a high-criticality task. In our work, we dynamically adjust LO-mode budgets of the high-criticality tasks when we detect delays at intermediate checkpoints. We decide about the LO-mode budgets based on the observed progress, instead of using static offline remaining time as used by other work. In addition, we have implemented an LLVM compiler pass to *automatically* instrument checkpoints in C and C++ programs, which have been tested with a PAStime implementation in LITMUS^{RT} for real-world applications developed for Linux. Kritikakou et al.'s research was implemented for a specific DSP platform, which is yet to be tested for real-world applications. Notwithstanding, the authors provide an important WCET analysis using CFGs for high-criticality tasks.

Previous ideas of progress-based scheduling were proposed to improve GPU performance [1, 25], fairness among multiple threads [18, 19] and to account for instruction cycles [17]. Most of these works run a task in an isolated environment and compare its progress to an online parallel execution with other tasks. Somewhat similar to the motivation behind Jeong et al's work [25], we also meet the deadlines of high-criticality tasks. However, PAStime uses CFGs to monitor progress rather than application-specific features such as frame processing rates in multimedia applications as done in the other works.

7 Conclusions and Future Work

This paper presents PAStime, a scheduling strategy based on the execution progress of a task. Progress is measured by observing the time taken for a program to reach a designated checkpoint in its control flow graph (CFG). We integrate PAStime in mixed-criticality systems by extending AMC scheduling. PAStime extends the LO-mode budget of a high-criticality task based on its observed progress, given that the extension does not violate the schedulability of any tasks. Our extension to AMC scheduling, called AMC-PAStime, is shown to improve the QoS of low-criticality tasks. Moreover, we implement an algorithm in the LLVM compiler to automatically detect and instrument viable program checkpoints for use in task profiling.

This paper presents the first implementation of AMC scheduling in LITMUS^{RT}. We have also implemented AMC-PAStime in LITMUS^{RT} and compared it against AMC. While both meet deadlines for all high-criticality tasks, AMC-PAStime improves the average utilization of low-criticality tasks by 1.5–9 times for 2–20 total tasks. AMC-PAStime is shown to improve performance for low-criticality tasks while reducing the number of mode switches. Finally, we have shown that different progress metrics also improve the LC tasks' utilization.

In future work, we will explore other uses of progress-aware scheduling in timing-critical systems. We plan to extend the Linux kernel `SCHED_DEADLINE` policy [30, 31, 34] to support progress-aware scheduling. We believe that PAStime is applicable to timing-sensitive cloud computing applications, where it is possible to adjust power (e.g., via Dynamic Voltage Frequency Scaling) based on progress. Application of PAStime to domains outside real-time computing will also be considered in future work.

References

1 Jayvant Anantpur and R Govindarajan. PRO: Progress-aware GPU Warp Scheduling Algorithm. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 979–988, 2015.

2 Neil C Audsley. On Priority Assignment in Fixed Priority Scheduling. *Information Processing Letters*, 79(1):39–44, 2001.

3 Sanjoy Baruah and Alan Burns. Fixed-priority Scheduling of Dual-criticality Systems. In *Proceedings of the 21st International conference on Real-Time Networks and Systems*, pages 173–181. ACM, 2013.

4 Sanjoy Baruah, Alan Burns, and Zhishan Guo. Scheduling Mixed-criticality Systems to Guarantee Some Service under All Non-erroneous Behaviors. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 131–138. IEEE, 2016.

5 Sanjoy Baruah, Haohan Li, and Leen Stougie. Towards the Design of Certifiable Mixed-criticality Systems. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 13–22, 2010.

6 Sanjoy Baruah and Steve Vestal. Schedulability Analysis of Sporadic Tasks with Multiple Criticality Specifications. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, pages 147–155, 2008.

7 Sanjoy K Baruah, Alan Burns, and Robert I Davis. Response-time Analysis for Mixed Criticality Systems. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS)*, pages 34–43, 2011.

8 Big Buck Bunny. `https://www.bigbuckbunny.org`, 2018.

9 Enrico Bini and Giorgio C Buttazzo. Measuring the Performance of Schedulability Tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

10 Björn Brandenburg and James H Anderson. *Scheduling and Locking in Multiprocessor Real-time Operating Systems*. PhD thesis, PhD thesis, The University of North Carolina at Chapel Hill, 2011.

11 Alan Burns and Sanjoy Baruah. Towards a more practical model for mixed criticality systems. In *Workshop on Mixed-Criticality Systems (colocated with RTSS)*, 2013.

12 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`.

13 Kevin Burr and William Young. Combinatorial Test Techniques: Table-based Automation, Test Generation and Code Coverage. In *Proceedings of the International Conference on Software Testing Analysis & Review*, 1998.

14 John M Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C Devi, and James H Anderson. LITMUS^{RT}: A Testbed for Empirically Comparing Real-time Multiprocessor Schedulers. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 111–126. IEEE, 2006.

15 Dionisio De Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. On the Scheduling of Mixed-criticality Real-time Task Sets. In *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS)*, pages 291–300, 2009.

16 dlib C++ Library. dlib: Video Tracking. `http://dlib.net/video_tracking_ex.cpp.html`, 2019.

17 Stijn Eyerman and Lieven Eeckhout. Per-thread Cycle Accounting in SMT Processors. *ACM Sigplan Notices*, 44(3):133–144, 2009.

18 Josue Feliu, Julio Sahuquillo, Salvador Petit, and José Duato. Addressing Fairness in SMT Multicores with a Progress-aware Scheduler. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 187–196, 2015.

19 Feliu, Josue and Sahuquillo, Julio and Petit, Salvador and Duato, Jose. Perf&Fair: A Progress-aware Scheduler to Enhance Performance and Fairness in SMT Multicores. *IEEE Transactions on Computers*, 66(5):905–911, 2017.

20 FFmpeg Multimedia Framework. `https://www.ffmpeg.org/`, 2019.

21 Chris Gill, James Orr, and Steven Harris. Supporting Graceful Degradation through Elasticity in Mixed-Criticality Federated Scheduling. In *Proc. 6th Workshop on Mixed Criticality Systems (WMC), RTSS*, pages 19–24, 2018.

22 Google. trucov. `https://code.google.com/archive/p/trucov/`, 2018.

23 Huang-Ming Huang, Christopher Gill, and Chenyang Lu. Implementation and Evaluation of Mixed-criticality Scheduling Approaches for Sporadic Tasks. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):126, 2014.

24 Intel. Intel® NUC Kit NUC6i7KYK (Skull Canyon): Features and Configurations. `https://www.intel.com/content/www/us/en/products/docs/boards-kits/nuc/nuc-kit-nuc6i7kyk-features-configurations.html`, 2019.

25 Min Kyu Jeong, Mattan Erez, Chander Sudanthi, and Nigel Paver. A QoS-aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC. In *Proceedings of the 49th annual Design Automation Conference*, pages 850–855. ACM, 2012.

26 Angeliki Kritikakou, Olivier Baldellon, Claire Pagetti, Christine Rochange, and Matthieu Roy. Run-time Control to Increase Task Parallelism in Mixed-critical Systems. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

27 Angeliki Kritikakou, Thibaut Marty, and Matthieu Roy. DYNASCORE: DYNAmic Software COntroller to increase REsource Utilization in Mixed-critical Systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 23(2), 2017.

28 Angeliki Kritikakou, Christine Rochange, Madeleine Faugère, Claire Pagetti, Matthieu Roy, Sylvain Girbal, and Daniel Gracia Pérez. Distributed Run-time WCET Controller for Concurrent Critical Tasks in Mixed-critical Systems. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*. ACM, 2014.

29 Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, page 75. IEEE Computer Society, 2004.

30 Juri Lelli, Giuseppe Lipari, Dario Faggioli, and Tommaso Cucinotta. An Efficient and Scalable Implementation of Global EDF in Linux. In *Proceedings of the 7th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, pages 6–15, 2011.

31 Juri Lelli, Claudio Scordino, Luca Abeni, and Dario Faggioli. Deadline Scheduling in the Linux kernel. *Software: Practice and Experience*, 46(6):821–839, 2016.

32 Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft COCO: Common Objects in Context. In *Proceedings of the European Conference on Computer Vision*, pages 740–755. Springer, 2014.

33 Linux. hrtimers - Subsystem for High-resolution Kernel Timers. `https://www.kernel.org/doc/Documentation/timers/hrtimers.txt`, Last Accessed: May 2019.

34 Linux. SCHED_DEADLINE Scheduling Policy. `https://www.kernel.org/doc/Documentation/scheduler/sched-deadline.txt`, Last Accessed: May 2019.

35 Di Liu, Nan Guan, Jelena Spasic, Gang Chen, Songran Liu, Todor Stefanov, and Wang Yi. Scheduling analysis of imprecise mixed-criticality real-time tasks. *IEEE Transactions on Computers*, 67(7):975–991, 2018.

36 Di Liu, Jelena Spasic, Nan Guan, Gang Chen, Songran Liu, Todor Stefanov, and Wang Yi. Edf-vd scheduling of mixed-criticality systems with degraded quality guarantees. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 35–46. IEEE, 2016.

37 LLVM LoopInfo Class. `https://llvm.org/doxygen/LoopInfo_8h_source.html`, Last Accessed: May 2019.

38 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*, pages 1–10. IEEE, 2014.

39 Eric Missimer, Katherine Missimer, and Richard West. Mixed-criticality scheduling with i/o. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 120–130. IEEE, 2016.

40 Alessandro Papadopoulos, Enrico Bini, Sanjoy Baruah, and Alan Burns. AdaptMC: A control-theoretic approach for achieving resilience in mixed-criticality systems. In *Proceeding ECRTS Conference*, page 14. LIPICS, 2018.

41 PAStime. Source Code, 2020. URL: `http://cs-people.bu.edu/soham1/pastime/`.

42 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.

43 Saravanan Ramanathan, Arvind Easwaran, and Hyeonjoong Cho. Multi-rate Fluid Scheduling of Mixed-criticality Systems on Multiprocessors. *Real-Time Systems*, 54(2):247–277, 2018.

44 Joseph Redmon. Darknet: Open Source Neural Networks in C. `http://pjreddie.com/darknet/`, 2013–2016.

45 Francois Santy, Laurent George, Philippe Thierry, and Joël Goossens. Relaxing mixed-criticality scheduling strictness for task sets scheduled with fp. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 155–165. IEEE, 2012.

46 Hang Su, Peng Deng, Dakai Zhu, and Qi Zhu. Fixed-priority Dual-rate Mixed-criticality Systems: Schedulability Analysis and Performance Optimization. In *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 59–68. IEEE, 2016.

47 Hang Su, Nan Guan, and Dakai Zhu. Service guarantee exploration for mixed-criticality systems. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10. IEEE, 2014.

48 Hang Su and Dakai Zhu. An Elastic Mixed-criticality Task Model and its Scheduling Algorithm. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 147–152. IEEE, 2013.

49 Hang Su, Dakai Zhu, and Scott Brandt. An Elastic Mixed-Criticality Task Model and Early-Release EDF Scheduling Algorithms. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 22(2):28, 2017.

50 Mustafa M Tikir and Jeffrey K Hollingsworth. Efficient Instrumentation for Code Coverage Testing. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 86–96. ACM, 2002.

51 Manohar Vanga, Andrea Bastoni, Henrik Theiling, and Björn B Brandenburg. Supporting Low-Latency, Low-Criticality Tasks in A Certified Mixed-Criticality OS. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 227–236. ACM, 2017.

52 Steve Vestal. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *Proceedings of the 28th IEEE Real-Time Systems Symposium (RTSS)*, pages 239–243, 2007.

53 Richard West, Puneet Zaroo, Carl A Waldspurger, and Xiao Zhang. Online Cache Modeling for Commodity Multicore Processors. *ACM SIGOPS Operating Systems Review*, 44(4):19–29, 2010.

54 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*, 7(3):36, 2008.

55 Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. COLORIS: A Dynamic Cache Partitioning System using Page Coloring. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 381–392. IEEE, 2014 .

56 Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM Bank-aware Memory Allocator for Performance Isolation on Multicore Platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166. IEEE, 2014.

Dynamic Interference-Sensitive Run-time Adaptation of Time-Triggered Schedules

Stefanos Skalistis
Raytheon Technologies, Cork, Ireland
SkalistSt@rtx.com

Angeliki Kritikakou
University of Rennes, Inria, IRISA, France
angeliki.kritikakou@inria.fr

Abstract

Over-approximated Worst-Case Execution Time (WCET) estimations for multi-cores lead to safe, but over-provisioned, systems and underutilized cores. To reduce WCET pessimism, interference-sensitive WCET (*is*WCET) estimations are used. Although they provide tighter WCET bounds, they are valid only for a specific schedule solution. Existing approaches have to maintain this *is*WCET schedule solution at run-time, via time-triggered execution, in order to be safe. Hence, any earlier execution of tasks, enabled by adapting the *is*WCET schedule solution, is not possible. In this paper, we present a dynamic approach that safely adapts *is*WCET schedules during execution, by relaxing or completely removing *is*WCET schedule dependencies, depending on the progress of each core. In this way, an earlier task execution is enabled, creating time slack that can be used by safety-critical and mixed-criticality systems to provide higher Quality-of-Services or execute other best-effort applications. The Response-Time Analysis (RTA) of the proposed approach is presented, showing that although the approach is dynamic, it is fully predictable with bounded WCET. To support our contribution, we evaluate the behavior and the scalability of the proposed approach for different application types and execution configurations on the 8-core Texas Instruments TMS320C6678 platform, obtaining significant performance improvements compared to static approaches.

2012 ACM Subject Classification Computer systems organization → Embedded software; Computer systems organization → Multicore architectures; Computer systems organization → Real-time systems

Keywords and phrases Worst-Case Execution Time, Interference-sensitive, Run-time Adaptation, Time-Triggered, Response Time Analysis, Multi-cores

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.4

Funding 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 constantly growing processing demand of applications has led the processor manufacturing industry towards multi-/many-core architectures. These architectures have multiple processing elements, called *cores*, providing massive computing capabilities, by being able to concurrently execute a high volume of tasks. Hard real-time systems have to provide *timing guarantees*, i.e., guarantee that tasks are completed before their respective latency requirements (deadlines). To rigorously provide such guarantees, deployment approaches schedule tasks on cores considering the Worst-Case Execution Time (WCET) of tasks.

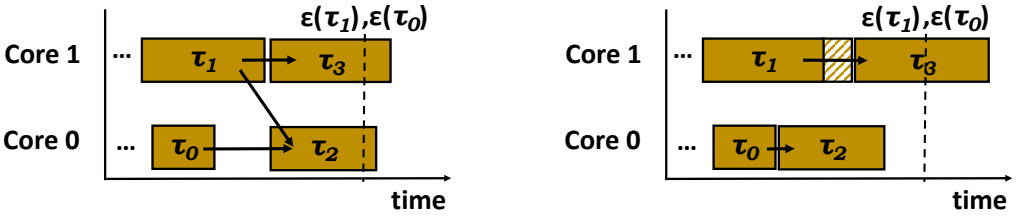
However, in multi-core architectures, several resources are shared among the cores (such as memories and interconnects) introducing timing delays (interferences), changing the timing behavior of tasks and varying their WCET. Indeed, tasks WCETs, which include interferences, can be 7.5 times larger than the corresponding estimations without interferences,

both experimentally measured [11, 13] and analytically computed [23, 24]. To account for all possible interferences, the WCET has to be over-approximated. This over-approximation practice has led to the "one-out-of-m processors" problem [8], where the additional processing capacity is negated by the pessimism of the WCET. As a result, the sequential execution (on a single core) may provide better timing guarantees than any parallel execution, which seriously undermines the advantages of utilizing multi-cores. To reduce the WCET pessimism, recent state-of-the-art research [15, 16, 19, 24] has proposed tighter WCET, called *interference-sensitive* WCET (*is*WCET). *is*WCET are computed by accounting for the interference that can occur only by the parallel-scheduled tasks. Hence, *is*WCETs are schedule-dependent, and they are valid only for the schedule solution they have been estimated for.

In order to guarantee a time-safe execution, this *is*WCET schedule solution has to be maintained during execution. Otherwise, additional interferences may occur, which have not been accounted for. To achieve that, time-triggered execution is usually applied, where the tasks are executed exactly at their start time assigned in *is*WCET schedule [18, 19]. Although time-triggered execution is time-safe, it prohibits any improvement on performance. Performance improvement can create slack, that can be used to increase the Quality-of-Service in safety-critical systems or execute other best-effort applications in mixed-critical systems. For example, in cruise control systems, the created slack can be used to further improve quality of the result produced by the control law, whereas in satellite systems less essential functions, such as scientific instrument data collection, can be activated [7]. The means to obtain any performance improvement is through run-time adaptation, using information of the task *actual execution time* (AET), that becomes available as the execution progresses. However, any adaptation occurring at run-time must be safe.

Existing *is*WCET run-time adaptation approaches [21, 22] allow tasks to be executed earlier-than-originally scheduled. Despite the potential earlier task execution, these approaches enforce the partial order of all tasks, provided by the *is*WCET schedule. In this way, additional interference due to earlier task execution cannot be introduced, maintaining the *is*WCET estimations valid. However, this enforced partial order of tasks limits the performance improvements that can be achieved through run-time adaptation. This limitation is illustrated in Figure 1, where arrows denote the partial order of tasks. As depicted in Figure 1a, a static run-time mechanism with enforced partial order can allow an earlier execution of successor tasks (τ_2 and τ_3), only when all their predecessor tasks have finished (τ_0 and τ_1). However, in permissive cases, τ_2 could be executed even earlier, since τ_0 has already finished execution before τ_1. Static mechanisms, that enforce the partial order, do not permit this earlier execution of τ_2, as τ_2 will insert interference to τ_1, which has not been computed during the creation of the *is*WCET schedule. Therefore, existing static mechanisms cannot exploit such opportunities created by the varying actual execution time of tasks across cores. However, assuming that τ_1 started earlier-than-originally scheduled, some time slack has been created at run-time, which can be exploited to further improve performance. As depicted in Figure 1b, if the additional *is*WCET, due to the interferences inserted by a new task running in parallel (τ_2 in this example, with interference illustrated in a stripped pattern), is less than the time slack, then the partial order of tasks can be safely relaxed, and τ_2 can be executed in parallel with τ_1.

In this work, we propose such a dynamic interference-sensitive run-time adaptation approach (*is*RA-DYN) that safely relaxes the partial order of tasks. The proposed approach exploits the actual execution time of tasks across cores to allow concurrent tasks to sustain more interference, than the one computed during the *is*WCET schedule, as long as the timing guarantees are preserved. Compared to existing approaches, the proposed approach



(a) Enforced partial order. **(b)** Relaxed partial order.

Figure 1 Motivational example: four tasks running on two cores and their *is*WCET dependencies.

is capable of exploiting the run-time variability due to a shorter task execution compared to the *is*WCET schedule computed offline. This run-time variability is created due to i) fewer interferences occurred during execution than the maximum possible interferences, used to offline compute the *is*WCET schedule, and ii) the executed path is different than the worst-case path of the task, used to compute the *is*WCET schedule. We provide the response timing analysis of the proposed approach, showing that timing guarantees are satisfied and any run-time adaptation does not alter the timing behavior of the system. To support our contributions, we perform extensive evaluation of the proposed approach on a real platform (8-core Texas Instruments TMS320C6678) for three applications and several execution configurations. The obtained results show that the proposed dynamic approach is able to provide performance improvements compared to the existing static approaches.

2 System model

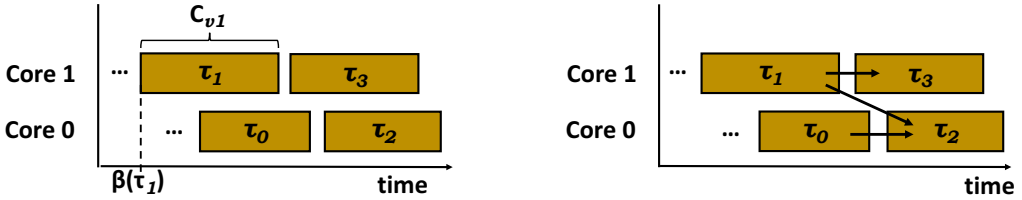
Let T denote the set of tasks of an application to be executed on the set of cores $\mathcal{K}$ of the target platform. The tasks of T can be either dependent, or independent, and are periodically executed in a non-preemptive manner. The proposed dynamic adaptation uses as input a *time-triggered schedule* that provides the start/end times of the tasks and their allocation to cores. Formally, we model such a *time-triggered schedule* S for the task-set T with the tuple (μ, β, ϵ), where $\mu(\tau)$ denotes the core allocation, i.e., the core k at which task τ is executed, and $\beta(\tau)$, $\epsilon(\tau)$ are the start and end times of the task, respectively. These times refer to the absolute time elapsed from the start of the period, and shall not be confused with the typical notion of *release time*. Such time-triggered schedule can be constructed by a scheduling algorithm providing timing guarantees, applied offline. Since the approach operates upon the input time-triggered schedule, any limitation will stem from the task model and scheduling algorithm, used offline to construct the time-triggered schedule. For clarity reasons, we will assume that tasks are released at the start of the period and their *is*WCET does not consider restrictions on the length of task overlapping or timing of the interference (see Section 6).

A time-triggered schedule S defines the partial ordering $\prec_S$ of the tasks, i.e., $\tau \prec_S \tau'$, iff task τ finishes its execution before τ' starts. Additionally, a schedule S is considered *safe*, iff it satisfies the system-defined timing constraints, i.e., each task deadline and/or a global deadline must be met. Given a safe time-triggered schedule S, let E_{isRA} be the transitive reduction of the tasks partial order, i.e. $(E_{isRA})^* \equiv \prec_S$. Essentially, E_{isRA} is a set of scheduling dependencies E_{isRA}, such that a task τ depends only on the tasks $\{\tau'\}$ that finished immediately before it, on all cores $\mathcal{K}$, according to $\beta(\tau)$, i.e.:

$$(\tau, \tau') \in E_{isRA} \iff \nexists\, \tau'' \text{ s.t. } \mu(\tau) = \mu(\tau'') \wedge \epsilon(\tau) < \epsilon(\tau'') \leq \beta(\tau')$$

Table 1 Notation Summary.

Tasks & Graph	
T, τ	Task τ belonging to task-set T
$\mathcal{K}$, k	Core k from set of cores $\mathcal{K}$
E_{isRA}, E_{dyn}	Offline and run-time scheduling dependencies
$deg^-(\tau)$, $deg^+(\tau)$	The *indegree* and *outdegree* of task τ
$pred(\tau)$, $succ(\tau)$	Predecessors/successors of task τ
$\iota_E(\tau)$, $\iota_{max}(\tau)$	Context-dependent and upper bound of interference of task τ
Time-triggered solution	
$\mu(\tau)$	Core allocation $\mu(\tau)$ of task τ
$\beta(\tau)$, $\epsilon(\tau)$	Start time $\beta(\tau)$ and end time $\epsilon(\tau)$ of task τ
Response Time & WCET	
$\mathcal{R}_\tau$, $\mathcal{S}_\tau$, $\mathcal{X}_\tau$, $\mathcal{N}_\tau$	Ready $\mathcal{R}_\tau$, relax $\mathcal{S}_\tau$, execute $\mathcal{X}_\tau$, notify $\mathcal{N}_\tau$ phase
$R(\tau)$, $R(\mathcal{R}_\tau)$, $R(\mathcal{X}_\tau)$, $R(\mathcal{N}_\tau)$	Absolute response time of task τ and its phases
σ_τ	Time-slack of task τ
$C^N_{[L]}$	The WCET of code snippet L of Algorithm N
$C_{\mathcal{R}_\tau}$, $C_{\mathcal{S}_\tau}$ $C_{\mathcal{N}_\tau}$	Controller WCET for the corresponding phase
$t^\tau_{\mathcal{R}}$, $t^\tau_{\mathcal{S}}$, $t^\tau_{\mathcal{N}}$,	Time instance when the corresponding phase can execute successfully (all branches are not taken)



(a) Time-triggered schedule. **(b)** Scheduling dependencies E_{isRA}.

Figure 2 Construction of E_{isRA} scheduling dependencies, based on a given time-triggered schedule.

Figure 2 illustrates the construction of E_{isRA} given a time-triggered schedule. Notice that, for any task τ in the dependency relation E_{isRA}, the number of incoming edges (denoted as $deg^-(\tau)$) and the number of outgoing edges (denoted as $deg^+(\tau)$) is upper bounded by the number of cores $|\mathcal{K}|$, i.e. $deg^-(\tau) \leq |\mathcal{K}|$ and $deg^+(\tau) \leq |\mathcal{K}|$. The proposed approach relaxes, whenever possible, the dependency relation E_{isRA}, which we shall denote as $E_{dyn} \subseteq E_{isRA}$.

The proposed dynamic adaptation mechanism is divided into four phases, namely *ready*, *relax*, *execute* and *notify*, which are respectively denoted with $\mathcal{R}_\tau$, $\mathcal{S}_\tau$, $\mathcal{X}_\tau$ and $\mathcal{N}_\tau$ for any task τ. Since time-triggered schedules refer to absolute time, we shall denote the *absolute* response time of the control phases with $R(\mathcal{R}_\tau)$, $R(\mathcal{S}_\tau)$, $R(\mathcal{X}_\tau)$, $R(\mathcal{N}_\tau)$.

Finally, given a set of potentially parallel tasks T^E_τ to task τ, given a dependency relation $E \subseteq E_{isRA}$, we assume that the interference $\iota_E(\tau)$ that τ can cause to, and sustain from, T^E_τ is computable and upper-bounded by $\iota_{max}(\tau)$. This is a realistic assumption, e.g., a task τ with Worst Case Resource Accesses, $WCRA(\tau)$, to an arbitrated resource considering a fair Round-Robin arbiter with arbitration delay of D_{RR} will cause/sustain [18]:

$$\iota_E(\tau) = D_{RR} * \sum_{k \in \mathcal{K} \setminus \{\mu(k)\}} \min(WCRA(\tau), \sum_{\tau' \in T^E_\tau, \mu(\tau')=k} WCRA(\tau')) \quad (1)$$

$$\iota_{max}(\tau) = D_{RR} * (|\mathcal{K}| - 1) * WCRA(\tau) \quad (2)$$

3 Dynamic *is*RA (*is*RA-DYN)

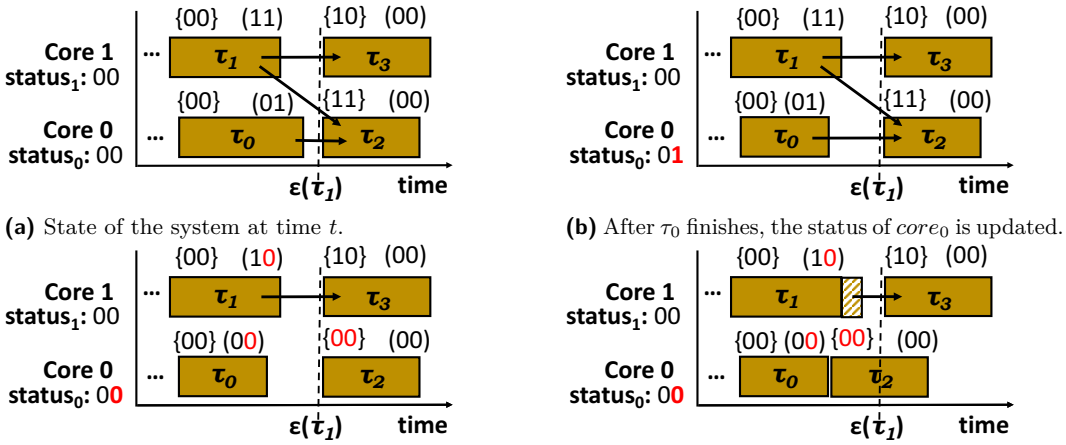
Run-time adaptation mechanisms for hard real-time systems, must guarantee that any adaptation decision does not violate the real-time constraints and the resulting execution is correct, i.e., no concurrency issue can occur. Furthermore, compared to traditional WCET schedules based on pessimistic WCET estimations, adapting interference-sensitive schedules poses an extra challenge: re-scheduling a task can increase the interference that another task sustains, as shown in Figure 1, potentially violating the timing guarantees. *Static* run-time adaptation [22] can safely adapt interference-sensitive time-triggered schedules by keeping fixed the partial order of task execution, preventing additional tasks overlaps, thus unaccounted interference to occur. Yet, static approaches miss adaptation opportunities due to this fixed partial order, being unable to provide further performance improvements.

Algorithm 1 *is*RA-DYN mechanism on core k.

```
   Input: Task τ, Array of all status vectors. (status_i[j]: the j-th status bit of the
          i-th core)
1  Function isRA-DYN (τ, status[ ]):
2      updateMinStart(k, β(τ))
3      while ¬ isReady(τ, status_k) do
4          if relax(τ, status_k) then
5              break;
6      execute(τ)
7      updateStatus(τ, status)
```

To address such cases, we propose a *dynamic* run-time adaptation mechanism, outlined in Algorithm 1, that is executed independently on each core and for each task. Each task *execution* is extended with control phases, *ready*, *relax* and *notify*. During the *ready* phase (L. 3), the controller checks if the current *active* task is ready, i.e., all previous tasks have finished, and thus, its dependencies have been met. In case the task is not ready, it tries to relax the partial order (L. 4) and checks again if the task is ready. To achieve a behavior that allows the partial order to change only when it is safe, a *global slack* is computed during execution, which is the minimum time-slack among all cores. The time-slack of a core is given by the amount by which the execution of its tasks has been sped up. The partial order, according to E_{isRA}, is allowed to be modified, if the introduced interference by any new task running in parallel is less than this global slack. This process continues, alternating a *ready* phase and a *relax* phase, until the task becomes ready and it is executed (L. 6). When the task finishes, the controller performs the *notify* phase, where it notifies all relevant cores that the task has finished (L. 7) and updates the necessary information for slack calculation (L. 2). A core k' is called relevant for any task τ executed on core k, when there exists an outgoing edge from task τ towards a task τ' on core k'. In order to enforce a particular ordering of the tasks (either the original partial ordering of E_{isRA} or any relaxation of it E_{dyn}) each core holds its own *status* vector (of size $|\mathcal{K}|$). Each bit of the *status* vector corresponds to a core. The *status* vector of each core represents the notifications received from other cores at any point in time and it must be updated during execution by all cores.

The following sections explain the controller phases with respect to the dependencies where relaxation can occur, i.e., scheduling dependencies. In case of data-dependent tasks, the data-dependencies are never removed.



(a) State of the system at time t.

(b) After τ_0 finishes, the status of $core_0$ is updated.

(c) Vector and status updates after τ_3 is relaxed.

(d) $core_1$ re-schedules τ_3.

Figure 3 Example of control phases for four tasks on two cores. For each task the *ready* vector is in curly brackets. The *notification* vector is in parentheses and illustrated with arrows.

3.1 Ready phase

To implement the *ready* phase, a *ready* vector (of size $|\mathcal{K}|$) is required for each task τ. Each bit in the *ready* vector represents the core k on which the incoming edge of the scheduling dependencies originates from, i.e.:

$$readyVector_\tau[k] = 1 \quad \Leftrightarrow \quad (\tau', \tau) \in E_{dyn} \wedge \mu(\tau') = k \tag{3}$$

where $readyVector_\tau$ is the ready vector of task τ. The *ready* vectors are created offline for each task τ, based on the dependency relation E_{isRA}, but may be modified during a relax phase of the same core, or by a notify phase of another core to reflect the dependency relation E_{dyn}. For instance, in Figure 3a, the *ready* vector of task τ_2 is $\{11\}$, since it is has to wait for i) task τ_1 running on core 1 and ii) task τ_0 running on core 0, to finish before being executed. These dependencies ensure that the number of interferences will not increase due to an earlier execution of τ_2. On the other hand, the *ready* vector of task τ_1 is $\{00\}$, as no dependency exists from another task.

The functionality of the *ready* phase of the controller is described in Algorithm 2. Initially, the controller tries to gain access to the critical section of the *status* vector through the protection mechanism related to core k (L. 2). Once it has been granted, it checks if all task

Algorithm 2 Ready phase of *is*RA-DYN mechanism on core k.

Input: Task τ, $status_k[\,]$ bit vector.
Output: **true** if all dependencies $readyVector_\tau$ have been met; otherwise **false**

```
1 Function isReady(τ, status_k[ ]):
2     enterSection(k)
3     if (status_k & readyVector_τ) = readyVector_τ then
4         status_k ← status_k ⊕ readyVector_τ
5         exitSection(k)
6         return true
7     exitSection(k)
8     return false
```

dependencies have been already met, encoded by the task's ready vector (L. 3). If this is true, the task τ can be executed. For instance, tasks τ_0 and τ_1 in Figure 3a are considered ready, since the corresponding bits of the status vectors of Core 0 and Core 1 are clear and the status vectors are equal with the ready vectors. Before advancing to the execution phase, the controller has to reset the bits indicated by the *ready vector* of task τ in its *status* (L. 4). In this way, any already-met dependencies from other cores to subsequent tasks on core k are preserved. Then, the protection mechanism is released and the task is executed.

3.2 Notify phase

To implement the *notify* phase, a *notification* vector (of size $|\mathcal{K}|$) is required for each task τ that describes which cores have to be informed that the task has finished execution. Each bit in the *notification* vector represents the core k, to which the outgoing edge of scheduling dependencies ends, i.e.:

$$notifyVector_{\tau}[k] = 1 \;\Leftrightarrow\; (\tau, \tau') \in E_{dyn} \wedge \mu(\tau') = k \tag{4}$$

where $notifyVector_{\tau}$ is the notify vector of task τ. The *notification* vectors are created offline for each task τ, based on the dependency relation E_{isRA}, but may be modified during a relax phase of another core to reflect the dependency relation E_{dyn}. For instance, in Figure 3a, the notification vector of task τ_1 is (11); when it finishes execution, it has to notify task τ_2 running on core 0 (bit 0) and task τ_3 running on core 1 (bit 1). Through the notification, the k-th bit in the *status* vector of core i is set by core k, when the finished task of core k has an outgoing edge to a task on core i. For example, in Figure 3b, the *status* vector of core 0 is 01, since task τ_0 finished execution and notified only core 0.

Algorithm 3 describes the functionality of the *notify* phase of the controller on core k. After the task τ on core k completes its execution, the controller has to update the *status* of all the relevant cores. To do so, for each successor τ' of task τ, the controller tries to gain access to the critical section of the successor's core protection mechanism (L. 3). If access is granted, the controller verifies if the dependency still exists (L. 4). If it exists, the controller tests if the previously occurred update of the core k has been already consumed by the core $\mu(\tau')$, where τ' is mapped to (L. 5). If this is true, the k-th bit in the status of core $\mu(\tau')$ is set, otherwise it clears the k-th bit from the ready vector of task τ', indicating that the dependency from core k has been met. For instance, Figure 3b illustrates this case

Algorithm 3 Notify phase of *is*RA-DYN mechanism on core k.

Input: Task τ, Array of all *status* vectors. ($status_i[j]$: the j-th *status* bit of the i-th core)

```
1 Function updateStatus(τ, status[ ]):
2     for τ' ∈ succ(τ) do
3         enterSection(μ(τ'))
4         if notifyVector_τ[μ(τ')] = 1 then
5             if status_μ(τ')[k] = 0 then
6                 status_μ(τ')[k] ← 1
7             else
8                 readyVector_τ'[k] ← 0
9         exitSection(μ(τ'))
```

where Core 0 updates its own bit after task τ_0 finishes. After the controller has updated all relevant status, it updates the start time of its active task with the time of the next task (L. 2, Algorithm 1) and computes the minimum among the active tasks (see Section 3.4).

3.3 Relax phase

In case the task is not ready to be executed, *is*RA-DYN tries to relax the partial ordering of the tasks, iff the introduced interference is less than the global slack (the amount that the execution has already advanced). That is, task τ is allowed to overlap with the active tasks, iff all active tasks started at least n time units before their time-triggered start time β, and task τ would introduce interference less than n time units to each one of them. This is illustrated in Algorithm 4 (L. 2) where the global *slack* has to be greater than the interference that task τ will introduce, denoted as $\iota_{max}(\tau)$, in addition to the WCET required of executing the relaxation, denoted as $C_{\mathcal{S}}$.

The relaxation strategy that *is*RA-DYN follows is an "all-or-nothing" approach, in the sense that either all the incoming scheduling dependencies, but no data ones, E^-_τ of task τ will be removed or the relaxation is postponed for a later invocation. The reasoning behind such design choice is that it provides short alternation between ready and relax phases. This minimizes the worst-case response time from the time a task becomes ready to when the task is executed by the controller. More formally, the result of such relaxation is:

$$E'_{dyn} = E_{dyn} \setminus E^-_\tau \quad \text{where} \quad E^-_\tau \subseteq pred(\tau) \times \{\tau\} \subset E_{isRA} \tag{5}$$

To achieve such relaxation, the controller of core k tries to gain access to its critical section (L. 4) and clears the k-th bit of the notification vector for each predecessor task τ' (L. 5-8), indicating that the dependency has been removed, as illustrated in Figure 3c. In order to reflect these changes to its own status and ready vectors, it registers which dependencies have been removed in a local variable, i.e., $modMask$ (L. 8). By definition, a dependency from a predecessor task τ' on the same core k is met, i.e., the notification from

Algorithm 4 Relax phase of *is*RA-DYN mechanism on core k.

Input: Task τ, $status_k[\,]$ bit vector.
Output: **true** if task τ is ready after the relaxation; otherwise **false**

```
1  Function relax(τ, status_k[ ]): bool
2      if getSlack() ≥ ι_max(τ) + C_{S_τ} then
3          modMask ← ¬(1 ≪ k)
4          for τ' ∈ pred(τ) do
5              if μ(τ') == k then continue
6              if ¬isDataDependent(τ',τ) then
7                  notifyVector_{τ'}[k] ← 0
8                  modMask[μ(τ')] ← 0
9          enterSection(k)
10         status_k ← status_k & modMask
11         readyVector_τ ← readyVector_τ & modMask
12         exitSection(k)
13         return readyVector_τ == 0
14     return false
```

core k has already occurred. Hence, the local variable is initialized with all bits set, except the k-th bit (L. 3). For the same reason, the k-th bit of that task's τ' notification vector is not reset (L. 6). Finally, the controller resets all the bits of its status and ready vector that were modified by the relaxation process, according to the local variable (L. 10-11), and tests (L. 13) if the task is indeed ready (to avoid re-execution of the ready phase), as shown in Figure 3d. Notice that, in case some of the tasks are data-dependent, the dependency is preserved (L. 6), thus ensuring proper ordering of data-dependent tasks.

3.4 Global slack computation

In order to relax the partial order of tasks, it is essential to know at run-time the amount of *global slack*, i.e., the minimum current time-slack across all cores. The time-slack of a core expresses the amount of time by which the execution of its tasks has been sped up. Speed-up occurs when the actual execution of a task is shorter than its *is*WCET. Formally, we define time-slack as the difference between the actual response time $R(\tau)$ of a task τ and its end time $\epsilon(\tau)$, and shall not be confused with the typical term slack, meaning laxity. As the actual response time $R(\tau)$ is not known a-priori, we use a safe slack approximation σ_τ:

$$\sigma_\tau = \beta(\tau) - t \quad \text{with} \quad \max_{\tau' \in pred(\tau)} R(\tau') \leq t \leq \beta(\tau) \tag{6}$$

where t is any time instance between the time-triggered start time and when the task becomes ready, i.e., all its predecessors have finished. Notice that σ_τ is safe since, $\epsilon(\tau) - R(\tau) \geq \beta(\tau) - \max_{\tau' \in pred(\tau)} R(\tau')$, i.e., the difference in response time between two consecutive (in time) tasks cannot be greater than the *is*WCET of the latter task, $\epsilon(\tau)$ - $\beta(\tau)$.

This safe approximation enables an efficient computation of the *global slack*, i.e., the minimum slack of all cores, at any time instance t, in a distributed manner, without requiring any sort of synchronisation or explicit exchange of information among cores. This is achieved by subtracting the current time instance t from the minimum start time of all active tasks, as outlined in Algorithm 5 (L. 9). To avoid inter-core information exchange, a global array is used to store the start time of the active task on each core and a global variable obtains the minimum value of the array. The start time of an active task is updated every time a core has to execute a new task (L. 3 of Algorithm 1). As soon as a core k finishes the notify phase of a task, it proceeds to its next task τ. As a new task is now active, the controller of

Algorithm 5 Slack computation on core k.

Input: Start time $\beta(\tau)$ of active task τ, Global array $startTimes$ of active tasks τ and global variable $minTime$ with the minimum value of $startTimes$.

```
1 Function updateMinStart(β(τ)):
2     prevStart ← startTimes[k]
3     startTimes[k] ← β(τ)
4     if minTime = prevStart then
5         for i ← 0 to |K| do
6             if minTime > startTimes[i] then
7                 minTime ← startTimes[i]
8 Function getSlack():
9     return minTime − getCurrentTime()
```

core k stores the old start time in a local variable and updates the start time of its active task with the new one (L. 2-3). If its old start time is equal to the minimum value of the array, it means that this controller was the owner of the minimum value, and thus, it has to recalculate the new minimum of the global array (L. 4-7). Otherwise, it delegates this computation to the controller that is the owner of the minimum value of the array.

3.5 Deadlock freedom

Since *is*RA-DYN is a distributed approach, it is important to establish its correctness. Here we prove that *is*RA-DYN is free from deadlocks, while time-correctness is addressed in Section 4. As a stepping stone, we first prove its static behavior, i.e., no relaxation occurs.

▶ **Lemma 1.** *Assuming that each set/reset operation on the k-th bit of the bit-vectors (status, readyVector, notifyVector) is atomic, the static behavior of* is*RA-DYN is free from deadlocks.*

Proof. Consider two dependent tasks, $(\tau, \tau') \in E_{isRA}$; there are two distinct cases when task τ finishes its execution and notifies core $\mu(\tau')$:

1. the $\mu(\tau)$-bit of the status for core $\mu(\tau')$ is not set ($status_{\mu(\tau')}[\mu(\tau)] = 0$), which results in setting the bit after notification (L. 6, Algorithm 3).
2. the $\mu(\tau)$-bit is already set ($status_{\mu(\tau')}[\mu(\tau)] = 1$) by some other task, which results in resetting the $\mu(\tau)$-bit of $readyVector_{\tau'}$ of task τ'. (L. 8, Algorithm 3).

At the ready phase of task τ' either the $\mu(\tau)$-bit of the ready vector is zero, or both the $\mu(\tau)$-bits of the status and the ready vector are set; in both cases task τ' is considered ready w.r.t. its dependency with task τ. In the former case, the value of that status vector bit is preserved (via XOR with the zero of the ready vector), in order to be reset by the corresponding task, while in the latter case that bit is reset. Since the value of the $\mu(\tau)$-bit of the status vector is the same before the notify phase and after the ready phase, it is straight forward to show that *is*RA-DYN is deadlock-free, via induction on E_{isRA} on all $|\mathcal{K}|$ bits. ◀

▶ **Theorem 2.** *Assuming that each set/reset operation on the k-th bit of the bit-vectors (status, readyVector, notifyVector) is atomic,* is*RA-DYN is free from deadlocks.*

Proof. Consider two dependent tasks, $(\tau, \tau') \in E_{dyn}$ and τ' is about to be executed; there are two distinct cases for task τ', namely either it is ready or it could be relaxed. The former case corresponds to the static behavior, which we have established its correctness from Lemma 1. In the latter case, there are two options for the $\mu(\tau)$-bit of $status_{\mu(\tau')}$:

1. the $\mu(\tau)$-bit has been set by τ
2. the $\mu(\tau)$-bit is not set

In both cases the $\mu(\tau)$-bit of $status_{\mu(\tau')}$ is reset, resetting also the $\mu(\tau)$-bit of $readyVector_{\mu(\tau')}$ (L. 10-11, Algorithm 4). Since τ' is about to execute, if the $\mu(\tau)$-bit is set, it cannot have been set by any other task τ'' than τ, as it would have been already reset by the succesive task of τ'' on core $\mu(\tau')$. It is thus, straightforward to show that *is*RA-DYN is deadlock-free, via induction on E_{dyn} on all $|\mathcal{K}|$ bits. ◀

In the presented algorithms, the modification of the bit-vectors is protected, satisfying the assumption for atomic set/reset operation of bits. In particular, since a ready and a relax phase cannot be executed simultaneously on the same core, it suffices to use a single protection mechanism per core. Additionally, since data-dependencies are always preserved (L. 7 in Algorithm 4), the execution with *is*RA-DYN cannot introduce race conditions to the application itself. It should be stressed that the global variable, with the minimum start

time of all active tasks across cores, does not require protection in order to be safe. The minimum start time is genuinely increasing, as the execution progresses, and its used for the calculation of the global slack. Accessing the global variable without protection can result in missing write from another core. This means that the controller uses an older value, which is smaller then the new one. This only results in smaller global slack computation, and thus, only missed opportunities of relaxation. This is deliberately done so, in favor of run-time performance.

4 Response time analysis

Introducing run-time mechanisms into time-critical systems can improve system performance, by better utilisation of the system resources. Nevertheless, the controllers themselves require processing time, thus they can alter the timing behavior and potentially violate timing guarantees, if the controller WCET is not properly accounted for. To overcome this issue, either additional tasks are incorporated into the model, used to derive the safe time-triggered schedule, or the WCET of the controller is incorporated into the WCET of each task (modulo some timing alignment). Especially for interference-sensitive system, attention must be paid to potential interference created by the controller, i.e., accesses to shared variables among cores (*status*, *ready* and *notify* vectors). If these variables are placed alongside with the task data, additional interferences must be accounted, due to the parallel execution of a control phase and a task. Multiple approaches exist to mitigate this effect; the control data can be placed in a separate memory accessed by a separate bus, when the platform provides such a feature. Alternatively, shared control data can be placed in the local memories of each core and accessed through remote reads/writes [2]. If such solutions are not possible, the amount of induced interference can be controlled, either by using resource-partitioning techniques common in real-time systems or by bounding the number of invocations of the controller, e.g., using non-interrupting hardware events [21].

In Algorithm 1, the proposed control mechanism is divided into two alternating phases, namely *ready* and *relax*, followed by two consecutive phases, *execute* and *notify*. We consider the absolute response time of a task τ to be when it finishes execution and the notify phase has performed all the status updates, i.e., the absolute response time of a task τ is the same as response time of its update phase:

$$R(\tau) = R(\mathcal{N}_\tau) \tag{7}$$

Notice that, while the execution phase $\mathcal{X}_\tau$ has a fixed *is*WCET (without considering the additional interferences due to relaxation), the *ready* and *notify* phases have varying *is*WCET, which depends on a number of factors. For any task τ, the *is*WCET of the ready phase depends on:

- **(i)** when it will gain access to its critical section, and
- **(ii)** when the task is going to be ready, i.e., all previous tasks have finished and all updates have been performed.

The WCET of the update phase depends on:

- **(i)** when it will gain access to its critical section,
- **(ii)** the number of cores to notify, and
- **(iii)** when previous tasks, which depended on this core, finish their ready phase (s.t. $status[i][k] = 0$).

In order to make our response time analysis accurate, we derive parametric response times R, based on the number of outgoing edges of a task τ, according to the scheduling dependencies E_{isRA}. We denote with $C^N_{[L]}$ the WCET of the controller part that corresponds

to the snippet L, i.e., the sequence of lines L of Algorithm N. We perform the RTA for the most restrictive case, i.e. for tasks with data-dependencies. An RTA for independent tasks, would at least provide the same response time bounds for the same task set, if not better.

Accessing a critical section. While one core can be only at one control phase at any time instance, different cores can be in different phases. Hence, for a core to enter any particular critical section, it may have to wait for all the other cores to finish their critical section. The WCET of the critical sections of the ready, notify and relax phase, are $C^2_{[3-6]}$, $C^3_{[3-9]}$ and $C^4_{[9-12]}$, respectively. Thus, the worst-case wait time for a core to access critical section i is:

$$C_i = (|\mathcal{K}| - 1) * \max\left(C^2_{[3-6]}, C^3_{[3-9]}, C^4_{[9-12]}\right) \tag{8}$$

Ready Phase. Let $t^\tau_{\mathcal{R}}$ be the time instance that task τ, running on core k, becomes ready, i.e., all predecessor tasks have finished their corresponding notify phases:

$$t^\tau_{\mathcal{R}} = \max_{\tau' \in pred(\tau)} R(\tau') \tag{9}$$

The response time of the ready phase, if the controller is invoked precisely at time $t^\tau_{\mathcal{R}}$, is the WCET of acquiring access to critical section k plus the WCET of executing that section:

$$R(\mathcal{R}_\tau) \leq t^\tau_{\mathcal{R}} + C_{\mathcal{R}_\tau} + C_{TA_{\mathcal{R}}} \quad \text{with} \quad C_{\mathcal{R}_\tau} = C^2_{[3-6]} + C_k \tag{10}$$

where $C_{TA_{\mathcal{R}}}$ is a timing alignment constant, analysed below. The response time $R(\mathcal{R}_\tau)$ is defined recursively, as it depends on the maximum response time of preceding tasks ($t^\tau_{\mathcal{R}}$). This will assist us in proving that under any AET, the execution respects the timing guarantees.

Execute Phase. As there are no preemptions during the execution of a task, if $\iota_{E_{dyn}}(\tau)$ is the interference that task sustains because of the relaxed dependency relation E_{dyn}, the response time for the execution phase of the controller is:

$$R(\mathcal{X}_\tau) \leq R(\mathcal{R}_\tau) + C_\tau + \iota_{E_{dyn}}(\tau) \tag{11}$$

Notify Phase. Following the task execution τ, on core k, the controller updates each core's status and the minimum start time of the active tasks. For each outgoing dependency, the core k has to gain access to a distinct critical section and perform a write to either the status or the ready vector:

$$R(\mathcal{N}_\tau) \leq R(\mathcal{X}_\tau) + C_{\mathcal{N}_\tau} \quad \text{with} \tag{12}$$

$$C_{\mathcal{N}_\tau} = C^5_{[1-7]} + \sum_{\tau' \in succ(\tau)} \left(C_{\mu(\tau')} + C^3_{[4-8]}\right) = C^5_{[1-7]} + deg^+(\tau) * \left(C_k + C^3_{[4-8]}\right) \tag{13}$$

Since the worst-case waiting time in Equation 8 is constant, we have replaced $C_{\mu(\tau')}$ with C_k to derive the final expression.

Relax Phase. Let time instance $t^\tau_{\mathcal{S}}$ be a time instance, where task τ running on core k is not ready yet, i.e., $t^\tau_{\mathcal{S}} < t^\tau_{\mathcal{R}}$, but the global slack is large enough to accommodate the additional interferences. If the relax phase is invoked at time $t^\tau_{\mathcal{S}}$, it has to remove all the

dependencies (excluding the data-dependencies) and acquire access to its critical section, in order to write the status vector. Hence, its response time is:

$$R(\mathcal{S}_\tau) \leq t^\tau_{\mathcal{S}} + C_{\mathcal{S}_\tau} \quad \text{with} \tag{14}$$

$$C_{\mathcal{S}_\tau} = C^4_{[2-4]} + \left(deg^+(\tau) - 1\right) * C^4_{[9-13]} + C^4_{[4-5]} + C_k + C^4_{[5-8]} \tag{15}$$

Notice that the loop body (L. 9-13) in Algorithm 4 is executed only $deg^+(\tau) - 1$ times, since the dependency from the core itself is by default met. This is reflected by the term $(deg^+(\tau) - 1) * C^4_{[9-13]} + C^4_{[4-5]}$ in the response time $R(\mathcal{S}_\tau)$.

Timing alignment. In the RTA of the ready phase, we assumed that the controller starts precisely at the time when all the required status updates have been performed (for the ready phase). Nevertheless, since the tasks can be executed in less time than their *is*WCET, there is a possibility that the controller is already inside a relax phase, when the last status update occurs. In the worst-case, there will be a single data-dependency that is not removed by the relax phase. Therefore, the timing alignment constant $C_{TA_\mathcal{R}}$ for the ready phase is:

$$C_{TA_\mathcal{R}} = C_\mathcal{S} - C^4_{[6-8]} + C^1_{[3]} \tag{16}$$

where $C_\mathcal{S}$ is the WCET of relax phase, with the maximum number of dependencies, i.e., $|\mathcal{K}|$.

4.1 Safety

Having the WCET and the response time of the controller phases, we need to prove that, if these costs are added upfront to the *is*WCET of tasks, the timing guarantees of any time-triggered schedule are not violated, under any run-time reduction of execution times, i.e., $R(\tau) \leq \epsilon(\tau)$ for all tasks τ.

Let $C_{\mathcal{R}_\tau}$, $C_{\mathcal{N}_\tau}$, $C_{\mathcal{S}_\tau}$ denote the WCET of the control phases, and $C_{TA_\mathcal{R}}$ the timing alignment constants, as analysed in the previous sections. Assume a safe solution (μ, β, ϵ), derived by a safe scheduling algorithm, such that it includes the controller WCETs in the *is*WCET of each task:

$$\epsilon(\tau) - \beta(\tau) = C_{\mathcal{R}_\tau} + C_{TA_\mathcal{R}} + C_\tau + C_{\mathcal{N}_\tau} + C_{\mathcal{S}_\tau} \tag{17}$$

Before proving the safety of the approach, we establish some properties regarding the impact of relaxations to *is*WCET of the task and control phases.

▶ **Property 1.** *Relaxation does not increase the WCET of control phases ($C_{\mathcal{R}_\tau}$, $C_{\mathcal{N}_\tau}$, $C_{\mathcal{S}_\tau}$).*

Proof. The WCET of the control phases depends on the indegree and outdegree of each task τ according to the dependency relation E_{isRA} (Equations 10, 13, 15). The dependency relation E_{dyn} is a genuinely decreasing relation (Equation 5) starting from E_{isRA}, thus the WCET of the control phases decreases with each relaxation. ◀

▶ **Property 2.** *Given a relaxed dependency relation $E_{dyn} \subseteq E_{\text{is}RA}$, a task τ can suffer additional interference at most equal to its slack:*

$$\sigma_\tau \geq \iota_{E_{dyn}}(\tau) \tag{18}$$

Proof. In case task τ is the task with the minimum start time among the active tasks, then $\sigma_\tau \geq \iota_{max}(\tau)$ (L. 2 in Algorithm 4). Otherwise, let τ_{min} be the task with minimum start time, i.e., $\beta(\tau) \geq \beta(\tau_{min})$. By equation 6:

$$\sigma_\tau - t \geq \sigma_{\tau_{min}} - t \Rightarrow \sigma_\tau \geq \sigma_{\tau_{min}} \tag{19}$$

Therefore, $\sigma_\tau \geq \iota_{max}(\tau) \geq \iota_{E_{dyn}}(\tau)$. ◀

▶ **Theorem 3.** *For any safe scheduling solution, derived by adding the controller costs ($C_{\mathcal{R}_\tau}$, $C_{\mathcal{N}_\tau}$, $C_{\mathcal{S}_\tau}$, $C_{TA_\mathcal{R}}$) to the is*WCET *(C_τ) of the tasks T, the* is*RA-DYN execution is safe under any AET, i.e.:*

$$R(\tau) \leq \epsilon(\tau) \tag{20}$$

Proof. Proof by induction on the dependency relation E_{dyn}.
Base Case: For tasks τ with no predecessors ($pred(\tau) = \emptyset$, $\beta(\tau) = 0$) from Eq. 7 and Eq. 12 we establish:

$$R(\tau) \leq R(\mathcal{X}_\tau) + C_{\mathcal{N}_\tau} \overset{(11)}{\Longrightarrow} R(\tau) \leq R(\mathcal{R}_\tau) + C_\tau + C_{\mathcal{N}_\tau} + \iota_{E_{dyn}}(\tau) \overset{(10)}{\Longrightarrow} \tag{21}$$

$$R(\tau) \leq t^\tau_\mathcal{R} + C_{\mathcal{R}_\tau} + C_{TA_\mathcal{R}} + C_\tau + C_{\mathcal{N}_\tau} + \iota_{E_{dyn}}(\tau) \overset{(9)}{\Longrightarrow} \tag{22}$$

$$R(\tau) \leq \max_{\tau' \in pred(\tau)} R(\tau') + C_{\mathcal{R}_\tau} + C_{TA_\mathcal{R}} + C_\tau + C_{\mathcal{N}_\tau} + \iota_{E_{dyn}}(\tau) \tag{23}$$

Since $pred(\tau) = \emptyset$, the response time of the predecessors is zero. Also no adaptation can occur since $\beta(\tau) = 0$, thus no additional interference is introduced, i.e. $\iota_{E_{dyn}}(\tau) = 0$:

$$R(\tau) \leq 0 + C_{\mathcal{R}_\tau} + C_{TA_\mathcal{R}} + C_\tau + C_{\mathcal{N}_\tau} + 0 \overset{(17)}{\Longrightarrow} \tag{24}$$

$$R(\tau) \leq \epsilon(\tau) - \beta(\tau) \overset{\beta(\tau)=0}{\Longrightarrow} R(\tau) \leq \epsilon(\tau) \tag{25}$$

Induction step: Suppose that (20) holds for all predecessors of task τ. Starting from equation (23) and through equation (6) we establish:

$$R(\tau) \leq \beta(\tau) - \sigma_\tau + C_{\mathcal{R}_\tau} + C_{TA_\mathcal{R}} + C_\tau + C_{\mathcal{N}_\tau} + \iota_{E_{dyn}}(\tau) \overset{(17)}{\Longrightarrow} \tag{26}$$

$$R(\tau) \leq \epsilon(\tau) - \sigma_\tau + \iota_{E_{dyn}}(\tau) \tag{27}$$

By Property 2 we know that $\iota_{E_{dyn}}(\tau) - \sigma_\tau \leq 0$, therefore, concluding the proof. ◀

We have therefore established that *is*RA-DYN is timely safe, and relaxes the dependency relation when enough global slack exists to accommodate the additional interference. Furthermore, the execution is work-conserving, w.r.t. E_{dyn}, which improves run-time performance, as shown in Section 5.

5 Experimental Evaluation

5.1 Experimental Setup

Platform. A real multi-core COTS platform, i.e., the TMS320C6678 chip (TMS in short) of Texas Instrument [25] is used for the experiments. The platform characteristics are depicted in Table 2. The *is*RA-DYN mechanism is implemented as a bare-metal library, with low-level functions for the controller phases using TMS hardware semaphores. The *is*RA-DYN semaphore implementation is applicable to any platform, since semaphores are a fundamental building block. In the rare case that no such hardware support exists, a software implementation can be used. However, the *is*RA-DYN approach can be implemented by other protection mechanisms.

Benchmarks. To experimentally evaluate our *is*RA-DYN approach, we have conducted experiments using three different applications with respect to the number of tasks, WCET, and WCRA taken from the StreamIT benchmarks [26]: i) Discrete Cosine Transformation (DCT), ii) Mergesort (MERGE), and iii) Fast Fourier Transformation (FFT).

Table 2 Benchmark and platform characteristics.

(a) Benchmark.

	No. tasks	Seq. WCET (cycles)	No. WCRA
DCT	32	981,120	69,808
MERGE	47	669,026	55,415
FFT	47	275,891	41,981

(b) TMS platform.

DSP char.	Instr.	Freq.	L1P	L1D	L2
	8	1 GHz	32 KB	32 KB	512 KB
DSPs	8	**NoC**	TeraNet (Delay: 11 cycles)		
L3	4 MB	**DDR3**	512 MB	**Sem.**	32 cycles

WCET and WCRA acquisition. Since no existing static WCET analysis tool supports the TMS platform, a measurement-based approach has been used to acquire the WCET of each task. Obtaining safe and context-independent measurements requires to eliminate the sources of timing variability [6], by disabling data-caches, removing interferences (i.e., the task is executed alone on one core) and providing input data to enforce the worst-case path. To perform our measurements on the real platform, we used the local timer of the core. To increase the reliability of the measurements, we have followed the approach of multiple executions. Each task has been executed 50 times, which has been shown to provide a small standard deviation [14], and maintained the largest observed value. The application has been compiled with -O0, i.e., no optimizations, in order to obtain the WCRA of each task by the produced binary. Table 2 depicts the overall WCET, WCRA and number of tasks of each benchmark, used to obtain the offline near-optimal solutions.

Data-placement. The controller data are placed on the on-chip Multicore Shared SRAM Memory (MSM), while application data are placed in the off-chip main memory (DDR3), ensuring that the *is*RA-DYN does not interfere with the task's execution.

Comparison. The evaluation of the proposed approach can be achieved by comparing run-time execution time of the tasks allocated on each core (a.k.a. makespan) obtained by the proposed *dynamic* approach (*is*RA-DYN) and the *static* run-time approach (*is*RA-LOCK) that enforces the partial order of tasks [22]. The offline *is*WCET schedule has been generated by [24] and it is used as an input to both approaches. To attribute the observed gains to isRA-DYN, any system parameter, that may lead to timing variability at run-time, should be controlled and explored independently, whenever possible. These parameters are mainly the interferences, the different execution paths of the benchmarks and the impact of caches. Therefore, we initially explore the timing variability that each benchmark can have, when executed on TMS platform and a single system parameter is tuned each time. Then, we present the gains provided by *is*RA-DYN and *is*RA-LOCK by comparing the makespan under different variations at the timing variability of benchmarks. Last, but not least, we compare the overhead of *is*RA-DYN and *is*RA-LOCK approaches.

5.2 Evaluation results

Characterization of timing variability. The main system parameters that can alter the execution time are the occurring interferences, the diverse execution paths of the benchmarks and the caches. In this first experimental section, we tune each of these parameters independently in order to characterize its impact to the timing variability per benchmark. To compute the timing variability, the execution time of the best observed case and the worst observed case are compared. Table 3 shows the timing variability due to caches and

Table 3 Benchmark timing variability.

(a) Interference variability

Caches	Path	DCT	MERGE	FFT
Disabled	**Best-Path**	32.13%	46.67%	55.03%
	Worst-Path	44.80%	43.78%	52.70%
Enabled	**Best-Path**	0.43%	0.91%	3.58%
	Worst-Path	0.32%	0.91%	3.29%

(b) Cache variability (No interferences)

Path	DCT	MERGE	FFT
Best-Path	73.83%	69.03%	69.40%
Worst-Path	76.57%	68.60%	69.38%

(c) Path variability (No interferences)

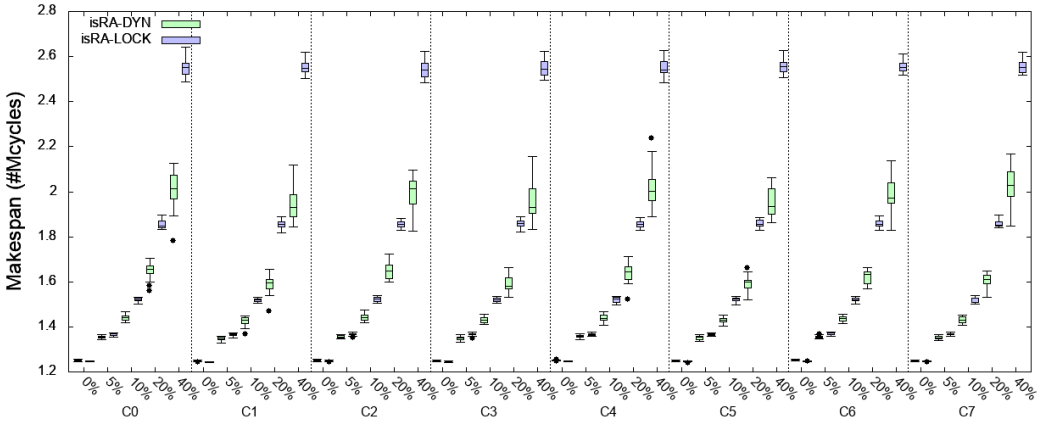
Caches	DCT	MERGE	FFT
Disabled	46.65%	12.84%	0.15%
Enabled	40.51%	14.69%	0.46%

diverse paths (computed without any interferences, i.e., running the benchmark alone on a core), and the timing variability due to interferences, when all cores are running the same benchmark. We observe that even when the application is executed in isolation, the impact of caches in execution time is quite high for all applications, with 71.14% on average. The impact of different execution paths depends on the application type, thus it is higher for the DCT, since it is an application that has several execution paths and, much smaller for FFT, which is a single-path application. Last but not least, we observe an important impact of the interferences, with 45.85% on average, with disabled caches. When caches are enabled the interference impact is reduced, since the cache sizes are large enough to keep the data locally.

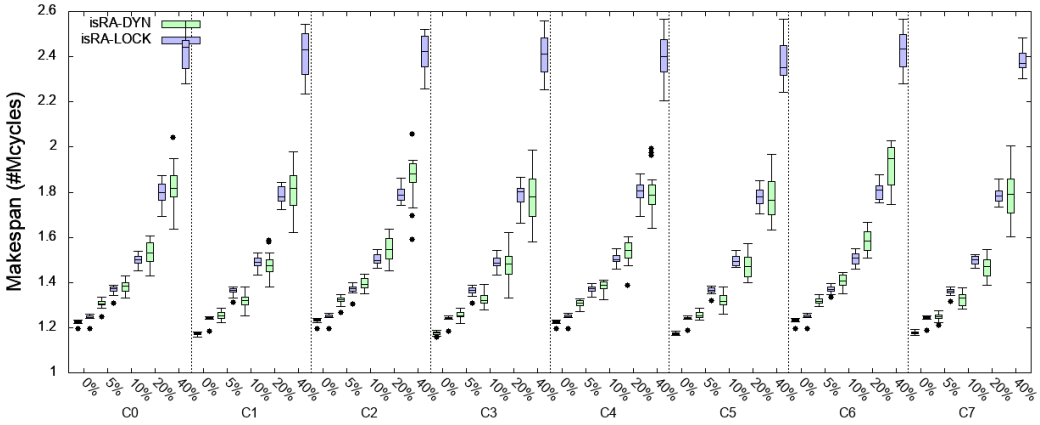
Makespan comparison. We perform an exhaustive set of experiments to explore and quantify the behavior of the proposed approach. We have used three configurations, i.e., two, four and eight application instances on two, four and eight cores, respectively. In addition, in order to explore the behavior of *is*RA-DYN with respect to the timing variability, due to interferences, caches, and multiple execution paths of the application, we have performed experiments, where we insert at each task a timing variability from 0% up to 40%, on average. Each experiment has been executed twenty consecutive iterations. During the execution, we observed no timing violations according to the offline solution. Due to page limitations, we only present the measured makespan of each core for the eight core configuration in Figure 4, in the form of box plot. However, we thoroughly analyze the behavior of our approach by providing the gains of the proposed *is*RA-DYN compared to *is*RA-LOCK for all experiments. The gain is given by computing the makespan gain, i.e., $\frac{(isRA-LOCK)-(isRA-DYN)}{(isRA-LOCK)}$, for each core. Tables 5, 4 and 6 depict the average makespan gain per core and the average makespan gain across all cores, for all configurations.

a) General observations: The first and important observation is that the behavior of *is*RA-LOCK is similar, in terms of minimum, maximum and average makespan, for all cores for all benchmarks, under any timing variability. This behavior of *is*RA-LOCK is due to the fixed partial task order. This behaviour motivates the use of a dynamic approach that can explore the variability occurring at run-time. Compared to the *is*RA-LOCK, the makespan distribution of the *is*RA-DYN among cores is varying. As *is*RA-DYN performs partial order relaxations, allows earlier task execution and additional interference to occur, which varies the core's makespan. When the variability is increased, this variation becomes more important.

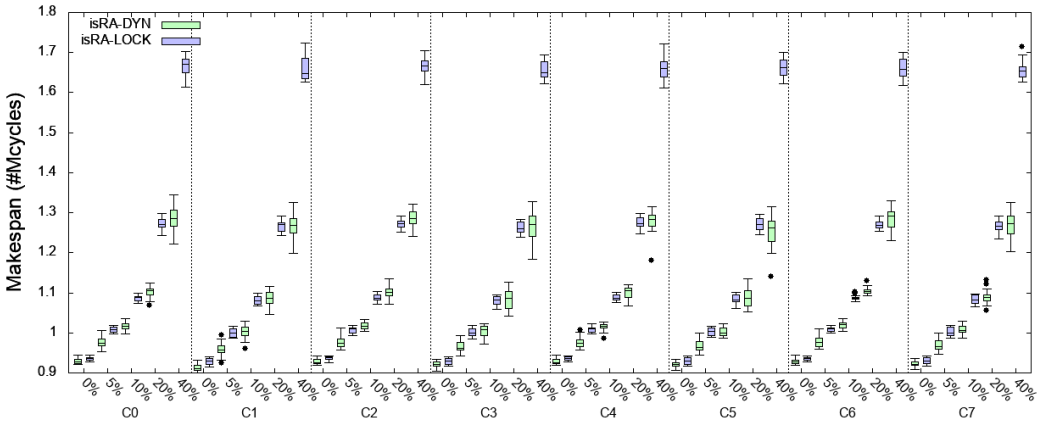
b) Minimum timing variability (0%): This experimental set-up is the worst for the proposed approach, since the timing variability of the benchmarks is eliminated as much as possible.



(a) DCT.



(b) MERGE.



(c) FFT.

Figure 4 Eight core configuration: *is*RA-DYN and *is*RA-LOCK makespan per core.

To achieve that, the same execution path is used among executions and caches are disabled. However, it is not possible to eliminate the interferences occurring from the parallel execution of tasks. For all the experiments, we observe that the behaviour of *is*RA-DYN improves over the behavior of *is*RA-LOCK, in all cores, as the number of cores increases. More precisely, for the configuration with two cores and 0% variability, *is*RA-DYN provides a small gain (from 0.08% for MERGE up to 0.22% for FFT, with an average of 0.145% among all applications). As the number of cores is increased, the gains are also increased, especially for MERGE. Compared to the two core configuration, the gain is increased on average by a factor of x3.11 for DCT, x36.63 for MERGE and x1.93 for FFT, when four cores are used, and by x2.95 for DCT, x44.64 for MERGE and x4.28 for FFT. The high gain factor of the MERGE benchmark is due to the low gain when only two cores are used. The lower gain, when only two cores are used, is attributed to the small number of interferences occurring during execution in combination with a bit higher run-time overhead, due to the relax phase, compared to *is*RA-LOCK. However, as the number of cores is increased, the number of occurring interference is increased, and thus, the gain is higher. As the only source of timing variability is the interferences in this experimental, the achieved gain of *is*RA-DYN verifies that the proposed approach is capable of exploring the occurring interferences during execution, compared to *is*RA-LOCK.

c) Tuning timing variability (from 5% to 40%): To quantitatively characterize the behavior of the proposed approach, when other sources of timing variability occur on top of the interferences, we insert an average variability of 5%, 10%, 20% and 40% in the WCET of the tasks (WCRA remains unchanged). For all the experiments, we observe that as task variability is inserted, *is*RA-LOCK fails to take advantage of it during execution, due to its fixed partial order policy. On the other hand, as the variability is increased, the proposed approach provides higher gains. More precisely, we observe that with the configuration with two cores (which is the configuration with the minimum possible interferences), the gains of *is*RA-DYN are significant compared to *is*RA-LOCK. In particular, with an increasing timing variability of 5%, the average gains are increased to 1.125% for DCT, 0.935% for MERGE, and 0.490% for FFT (with an average of 0.85% for all benchmarks). Tables 5, 4 and 6 show that with timing variability increasing, the gains are increased. Considering all

Table 4 MERGE: Average gains (%) per core (C) and among all cores (A).

MERGE									
Timing	**2 cores**			**4 cores**					
Var. (%)	**CO**	**C1**	**A**	**CO**	**C1**	**C2**	**C3**	**A**	
0	0.08	0.08	0.08	1.90	6.17	1.81	1.84	2.930	
5	1.03	0.84	0.935	2.10	7.40	2.46	2.56	3.630	
10	2.29	2.09	2.190	4.04	9.37	4.48	4.63	5.630	
20	4.84	4.76	4.800	9.59	13.06	9.59	10.09	10.583	
40	9.35	9.29	9.32	16.80	19.85	17.96	17.50	18.028	
Timing	**8 cores**								
Var. (%)	**CO**	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**A**
0	1.96	5.42	1.64	5.30	2.06	5.37	1.69	5.13	3.571
5	4.58	8.01	3.62	7.84	4.56	8.07	3.76	8.22	6.083
10	7.78	11.43	7.19	11.15	7.98	11.60	6.40	11.52	9.381
20	14.90	17.27	13.70	17.69	14.95	13.40	15.25	17.77	15.616
40	24.49	24.92	23.04	26.11	25.31	25.17	20.91	24.78	24.341

Table 5 DCT: Average gains (%) per core (C) and among all cores (A)

DCT									
Timing	**2 cores**			**4 cores**					
Var. (%)	**C0**	**C1**	**A**	**C0**	**C1**	**C2**	**C3**	**A**	
0	0.14	0.13	0.135	0.46	0.41	0.40	0.41	0.42	
5	1.10	1.15	1.125	1.51	1.69	1.56	1.66	1.605	
10	2.56	2.63	2.595	3.67	4.95	4.04	3.83	4.123	
20	5.94	5.58	5.760	6.28	10.77	7.95	7.61	8.153	
40	11.35	10.48	10.915	13.85	17.23	16.48	14.54	15.525	
Timing	**8 cores**								
Var. (%)	**C0**	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**A**
0	0.36	0.44	0.33	0.42	0.36	0.46	0.42	0.40	0.398
5	0.64	1.16	0.75	1.06	0.70	1.06	0.71	1.05	0.891
10	3.77	6.17	4.37	5.78	3.87	5.73	5.76	5.49	5.118
20	11.68	14.56	11.90	14.21	12.21	14.51	13.08	13.86	13.251
40	21.30	23.49	21.87	23.44	21.46	23.64	21.78	21.33	22.288

benchmarks, we observe an average gains of 2.09% (10%), 4.95% (20%), and 9.33% (40%). The maximum gain for 40% variability is 11.35% observed for C0 running DCT. As the number of cores is increased, the gains are also increased. This occurs due to the fact that the proposed approach is able to take advantage of both the inserted timing variability and the occurring interferences. When four cores are used, the average gain over all benchmarks is 1.89%, 3.82%, 8.46% and 15.86%, for 5%, 10%, 30% and 40% variability, respectively. The maximum gain for 40% variability is 19.85% observed for C1 running MERGE. When eight cores are used, the gains are even higher, i.e., with an average gain over all benchmarks equal to 3.45%, 7.11%, 14.22% and 23.26%, for 5%, 10%, 30% and 40% variability, respectively. The maximum gain for 40% variability is 25.31% observed for C4 running MERGE.

5.3 Controller cost

Table 7 depicts the corresponding WCET values for the *is*RA-DYN and *is*RA-LOCK approaches. Due to the additional relax phase, the overhead of the *is*RA-DYN controller is higher than *is*RA-LOCK controller. Despite the increased overhead, *is*RA-DYN can provide further performance improvements, as it has been shown in the previous paragraphs.

6 Related Work

The run-time mechanisms are categorized whether: i) the considered tasks are *only time-critical* or *also best-effort*, ii) the WCET is *pessimistic* or *interference-sensitive*, and iii) the adaptation is *static* or *dynamic*. A detailed survey of the state-of-the-art is available in [4].

The run-time mechanisms considering only time-critical tasks must guarantee the timely execution of the complete task set. The mechanisms that consider the pessimistic WCET are typically based on static decisions, i.e., the execution of a new task can start as soon as a task finishes earlier than its pessimistic WCET. Typical examples of such approaches come from scheduling theory, e.g. [1, 5]. However, the use of pessimistic WCET over-approximates the interferences having a negative impact in performance and in schedulability. To tackle with over-approximated WCETs due to interferences, several approaches incorporate interference

Table 6 FFT: Average gains (%) per core (C) and among all cores (A).

FFT									
Timing	**2 cores**			**4 cores**					
Var. (%)	**CO**	**C1**	**A**	**CO**	**C1**	**C2**	**C3**	**A**	
0	0.22	0.22	0.220	0.51	0.39	0.42	0.38	0.425	
5	0.50	0.48	0.490	0.40	0.40	0.44	0.36	0.400	
10	1.49	1.48	1.485	1.67	1.78	1.67	1.68	1.700	
20	3.75	4.80	4.275	6.70	7.24	7.10	6.75	6.948	
40	7.45	8.05	7.750	13.76	14.81	13.61	14.00	14.045	

Timing	**8 cores**								
Var. (%)	**CO**	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**A**
0	0.68	1.69	0.85	0.88	0.82	0.98	0.73	0.90	0.942
5	3.04	4.15	3.04	3.58	3.22	3.56	3.03	3.31	3.366
10	6.44	7.38	6.40	7.09	6.53	7.53	6.35	6.99	6.839
20	13.33	14.23	13.45	14.28	13.65	14.41	13.02	14.00	13.797
40	22.79	23.59	22.90	23.64	22.97	24.65	22.53	22.23	23.163

Table 7 Control phases WCET.

	***is*RA-LOCK WCET**		***is*RA-DYN WCET**	
	Fixed Cost	**Cost/Edge**	**Fixed Cost**	**Cost/Edge**
$C_{\mathcal{R}_\tau}$	355	251	355	251
$C_{\mathcal{N}_\tau}$	260	263	260	263
$C_{\mathcal{S}_\tau}$	-	-	183	201
$C_{TA_\mathcal{R}}$	45	-	45	-

analysis and provide *interference-sensitive* WCETs, such as [17, 19, 24]. In general, these approaches provide improved timing guarantees, since they compute a context-dependent upper-bound of the interferences for a particular schedule. To improve the provided upper-bounds, some approaches take into account the length of task overlapping, e.g. [17], or the precise timing of the requests, e.g. [20], or even provide contention-free schedules, e.g. [19]; a detailed survey of such approaches can be found in [12]. To further reduce the impact of the inherent pessimism in any kind of WCET estimations, several run-time mechanisms have been proposed. In [21, 22, 24], the authors provide a run-time approach suitable for interference-sensitive WCET. However, these approaches act upon static decisions, being unable to modify the partial order of tasks, created offline during the interference-sensitive scheduling. In contrast, the proposed *is*RA-DYN approach embraces dynamic decisions allowing safe modification of the *is*WCET scheduling, leading to performance improvements.

The run-time mechanisms for time-critical and best-effort tasks assume the timely execution of time-critical tasks, when they run in isolation. Based on this assumption, they decide the best-effort tasks execution, so as to still guarantee the timely execution of time-critical tasks. Such approaches use different confidence levels in the WCET estimation [3], compute the remaining WCET in isolation for the time-critical tasks [9–11], use resource usage capacities [15, 16] and partitioning of the memory accesses [27]. The *is*RA-DYN approach is orthogonal to these approaches, since it focuses on providing timing guarantees for the time-critical tasks.

7 Conclusion

In this work, we propose a dynamic interference-sensitive run-time adaptation technique *is*RA-DYN that alleviates the limitations of the existing *is*RA-LOCK, since it allows to safely relax the partial order of *is*WCET schedule solutions, whenever this is possible. We have presented the corresponding RTA for our technique and have formally argued regarding its safety, under any execution. The obtained results show that *is*RA-DYN outperforms *is*RA-LOCK as it can exploit variability in actual execution of tasks. When using two cores, the interferences are few and without any variability, the *is*RA-DYN provides small gains. However, with increasing variability, even with under few interferences, *is*RA-DYN provide significant gains. As the number of cores is increased, *is*RA-DYN provides better gains.

References

1 Marko Bertogna. *Real-time scheduling analysis for multiprocessor platforms.* PhD thesis, Scuola Superiore Sant'Anna, Pisa, 2008.

2 Alessandro Biondi and Marco Di Natale. Achieving predictable multicore execution of automotive applications using the let paradigm. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 240–250. IEEE, April 2018. doi:10.1109/RTAS.2018.00032.

3 Alan Burns and Sanjoy K. Baruah. Timing faults and mixed criticality systems. In CliffB. Jones and JohnL. Lloyd, editors, *Dependable and Historic Computing*, volume 6875 of *Lecture Notes in Computer Science*, pages 147–166. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-24541-1_12.

4 Alan Burns and Robert I. Davis. A survey of research into mixed criticality systems. *ACM Comput. Surv. (CSUR)*, 50(6):82:1–82:37, 2018. doi:10.1145/3131347.

5 Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv. (CSUR)*, 43(4):35, 2011. doi:10.1145/1978802.1978814.

6 Jean-François Deverge and Isabelle Puaut. Safe measurement-based WCET estimation. In Reinhard Wilhelm, editor, *5th International Workshop on Worst-Case Execution Time Analysis (WCET'05)*, volume 1 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2007. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2005.808.

7 Alexandre Esper, Geoffrey Nelissen, Vincent Nélis, and Eduardo Tovar. An industrial view on the common academic understanding of mixed-criticality systems. *Real-Time Systems*, 54(3):745–795, 2018.

8 Namhoon Kim, Bryan C. Ward, Micaiah Chisholm, Cheng-Yang Fu, James H. Anderson, and F. Donelson Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12. IEEE, 2016.

9 Angeliki Kritikakou, Olivier Baldellon, Claire Pagetti, Christine Rochange, and Matthieu Roy. Run-time control to increase task parallelism in mixed-critical systems. In *ECRTS*, 2014.

10 Angeliki Kritikakou, Thibaut Marty, and Matthieu Roy. DYNASCORE: dynamic software controller to increase resource utilization in mixed-critical systems. *ACM Trans. Design Autom. Electr. Syst. (TODAES)*, 23(2):13:1–13:26, 2018. doi:10.1145/3110222.

11 Angeliki Kritikakou, Christine Rochange, Madeleine Faugère, Claire Pagetti, Matthieu Roy, Sylvain Girbal, and Daniel Gracia Pérez. Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems. In *International Conference on Real-Time Networks and Systems (RTNS)*, pages 139–148. ACM, 2014. doi:10.1145/2659787.2659799.

12 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 (CSUR)*, 52(3):56, 2019.

13 Sébastien Martinez, Damien Hardy, and Isabelle Puaut. Quantifying wcet reduction of parallel applications by introducing slack time to limit resource contention. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 188–197. ACM, 2017.

14 C. Moreno and S. Fischmeister. Accurate measurement of small execution times—getting around measurement errors. *IEEE Embedded Systems Letters*, 9(1):17–20, March 2017.

15 Jan Nowotsch and Michael Paulitsch. Quality of service capabilities for hard real-time applications on multi-core processors. In *International Conference on Real-Time Networks and Systems (RTNS)*, pages 151–160. ACM, 2013. doi:10.1145/2516821.2516826.

16 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),*, pages 109–118. IEEE, 2014.

17 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 24th International Conference on Real-Time Networks and Systems*, RTNS '16, pages 67–76, New York, NY, USA, 2016. ACM. doi:10.1145/2997465.2997472.

18 Benjamin Rouxel, Steven Derrien, and Isabelle Puaut. Tightening contention delays while scheduling parallel applications on multi-core architectures. *ACM Trans. Embed. Comput. Syst. (TECS)*, 16(5s):164:1–164:20, September 2017. doi:10.1145/3126496.

19 Benjamin Rouxel, Stefanos Skalistis, Steven Derrien, and Isabelle Puaut. Hiding communication delays in contention-free execution for spm-based multi-core architectures. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

20 Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele. Timing analysis for TDMA arbitration in resource sharing systems. In *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 215–224. IEEE, 2010.

21 Stefanos Skalistis, Federico Angiolini, Alena Simalatsar, and Giovanni De Micheli. Safe and efficient deployment of data-parallelizable applications on many-core platforms: Theory and practice. *IEEE Design & Test*, 35(4):7–15, 2018.

22 Stefanos Skalistis and Angeliki Kritikakou. Timely fine-grained interference-sensitive run-time adaptation of time-triggered schedules. In *Real-Time Systems Symposium (RTSS)*. IEEE, 2019.

23 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 (FORMATS)*, pages 211–227. Springer, 2016.

24 Stefanos Skalistis and Alena Simalatsar. Near-optimal deployment of dataflow applications on many-core platforms with real-time guarantees. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 752–757. IEEE, 2017.

25 Texas Instruments. TMS320C6678 Multicore fixed and floating-point digital signal processor. Technical Report SPRS691D, TI, 2013.

26 William Thies and Saman Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 365–376. ACM, 2010.

27 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 *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013.

Improving the Accuracy of Cache-Aware Response Time Analysis Using Preemption Partitioning

Filip Marković
Mälardalen University, Västerås, Sweden
filip.markovic@mdh.se

Jan Carlson
Mälardalen University, Västerås, Sweden
jan.carlson@mdh.se

Sebastian Altmeyer
University of Augsburg, Germany
altmeyer@informatik.uni-augsburg.de

Radu Dobrin
Mälardalen University, Västerås, Sweden
radu.dobrin@mdh.se

Abstract

Schedulability analyses for preemptive real-time systems need to take into account cache-related preemption delays (CRPD) caused by preemptions between the tasks. The estimation of the CRPD values must be sound, i.e. it must not be lower than the worst-case CRPD that may occur at runtime, but also should minimise the pessimism of estimation. The existing methods over-approximate the computed CRPD upper bounds by accounting for multiple preemption combinations which cannot occur simultaneously during runtime. This over-approximation may further lead to the over-approximation of the worst-case response times of the tasks, and therefore a false-negative estimation of the system's schedulability. In this paper, we propose a more precise cache-aware response time analysis for sporadic real-time systems under fully-preemptive fixed priority scheduling. The evaluation shows a significant improvement over the existing state of the art approaches.

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

Keywords and phrases Real-time systems, Fixed-Priority Preemptive Scheduling, Preemption delay

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.5

Acknowledgements We want to thank our colleagues Sebastian Hahn, Jan Reineke, and Darshit Shah, who provided us with the evaluation data derived from the code-level analysis of benchmark programs. Also, we are very grateful to Davor Čirkinagić who borrowed his powerful computing system for performing schedulability evaluation.

1 Introduction

Fully-preemptive scheduling is used in many real-time embedded systems in order to e.g., overcome the limitations of non-preemptive scheduling which can introduce significant blocking on high priority tasks from lower priority ones. Fully-preemptive scheduling allows for an interruption (preemption) of the task's execution whenever a task with a higher priority is released. However, as shown by Pellizzoni et al. [21], a preemption can introduce a significant preemption related delay, even up to 33% of the task's worst-case execution time.

In embedded systems employing a cache-based architecture, one of the major causes of preemption delay is cache-related preemption delay (CRPD), as shown by Bastoni et al. [7]. CRPD represents the longest time needed by a resuming task to reload the memory cache blocks which it had loaded prior to the preemption. Since CRPD may significantly increase

the worst-case execution time of a task, its tight estimation is very important and therefore a new field of timing analysis, called cache-related preemption delay analysis, has emerged in the research of real-time systems.

In the context of CRPD analysis for fixed-priority fully-preemptive scheduling, many different approaches have been proposed in the last few decades, of which we describe a selection of the most recent ones. Tomiyama et al. [31], and Busquets-Mataix et al. [11] proposed analyses which are based on the over-approximation that a single preemption causes the CRPD equal to the time needed for reloading all the evicting cache blocks from a preempting task. These analyses neglected the fact that not every eviction results in a cache block reload. Contrary to this, Lee et al. [20] proposed the analysis that bounds the CRPD by accounting for the cache blocks which may be reused at some later point in a task, called the useful cache blocks. However, their analysis did not account for the fact that although useful, some cache block cannot be evicted, and thus cannot result in a cache block reload. These two opposite approaches defined the two main branches in CRPD analysis: ECB-based CRPD and UCB-based CRPD.

Later, Tan and Mooney [30] proposed the UCB-union approach, which accounted for the limitations of the above-described approaches. In this approach, CRPD is computed using the information about all possibly affected useful cache blocks along with the evicting cache blocks from the tasks which may evict them. Opposite to that, Altmeyer et al. [3] proposed the ECB-union approach, where the all possibly evicting cache blocks are analysed along with the useful cache blocks from the tasks that may be preempted.

The latest and in overall the most precise CRPD approaches are proposed by Altmeyer et al. [4], called ECB-union multiset and UCB-union multiset approaches. Those approaches are improvements over the UCB-union and ECB-union because they account for a more precise estimation of the nested preemptions. The multiset approach was also used by Staschulat et al. [28] for the periodic task systems where they accounted that each additionally accounted preemption of a single preempting task may result in a smaller CRPD value compared to the previous preemptions. In the context of periodic systems, Ramaprasad and Mueller [23, 22] investigated the possibility of tightening the CRPD bounds using preemption patterns. However, in this paper, we consider a sporadic task model which constrains such analysis.

Furthermore, in recent years, several cache-aware analysis were proposed in the contexts of: cache partitioning by Altmeyer et al. [26, 5], cache-persistence by Rashid et al. [25, 24] and Stock et al. [29], write-back cache by Davis et al. [12] and Blaß et al. [9].

In all of the above-mentioned CRPD analyses, the resulting upper bounds are overly pessimistic mainly because they account for CRPD obtained from preemption combinations which cannot occur simultaneously during runtime.

In this paper, we propose a cache-aware response-time analysis that accounts for the above-mentioned source of pessimism, and a few more, in the context of fixed priority fully-preemptive scheduling (FPPS). The evaluation shows a significant improvement over the existing state of the art approaches.

In the remainder of the paper, we first define the system model in Section 2. In Section 3, we overview the existing SOTA UCB-union and ECB-union based methods, including the multiset variants. In Section 4, we discuss the pessimism in the current state of the art cache-aware analyses. The proposed analysis is defined in Section 5, and the evaluation results are shown in Section 6. The paper is concluded in Section 7.

2 Task Model, Terminology and Notation

In this paper, we consider a sporadic task model, with preassigned fixed task priorities, under fully-preemptive scheduling. A taskset Γ consists of n tasks sorted in a decreasing priority order, where each task τ_i generates an infinite number of jobs, characterised with the following task parameters $\langle P_i, C_i, T_i, D_i \rangle$. Task priority is denoted with P_i and we assume disjunct priorities among the tasks. The worst-case execution time without accounted preemption delays is denoted with C_i. T_i denotes the minimum inter-arrival time between the two consecutive jobs of τ_i, and the relative deadline is denoted with D_i.

We also consider single-core systems with single-level direct-mapped caches, extending the task model by accounting for detailed knowledge about the cache usage. In addition, we describe a possible adjustment in subsection 5.8, thus also considering LRU set-associative caches. For each task τ_i, the information about the accessed cache blocks within the task's execution is assumed as derived, and based on that we define the following cache block sets:
ECB_i – a set of *evicting cache blocks* of τ_i, such that cache-set s is in ECB_i if and only if a memory block from s may be accessed during the execution of τ_i.
UCB_i – a set of all *useful cache blocks* throughout the execution of τ_i. As proposed by Lee et al. [20], and superseded by Altmeyer et al. [2], a cache-set s is in UCB_i, if and only if τ_i accesses a memory block m in s such that: a) m must be cached at some program point $\mathcal{P}$ in the execution of τ_i, and b) m may be reused on at least one control flow path starting from $\mathcal{P}$ without the eviction of m on this path.

In the remainder of the paper, we use the following notations for different sets of tasks:

- $hp(i)$ A set of tasks with priorities higher than τ_i
- $hpe(i)$ A set of tasks with priorities higher than τ_i, including τ_i, i.e. $hpe(i) = hp(i) \cup \{\tau_i\}$
- $lp(i)$ A set of tasks with priorities lower than τ_i
- $aff(i,h)$ A set of tasks with priorities higher than or equal to τ_i and lower than τ_h, i.e. $aff(i,h) = hpe(i) \cup lp(h)$

3 Background

Cache-related preemption delay is computed as the upper bound on the number of cache block reloads that can be caused due to preemptions and potential evictions of memory contents that are used by the preempted tasks. In this paper, CRPD is denoted with γ and is computed as the multiplication of the upper bound on cache block reloads with the constant BRT, which is the longest time needed for a single memory block to be reloaded into cache memory, i.e. block reload time. The general formula is $\gamma = \#reloads \times BRT$.

In this section, we briefly describe the most relevant CRPD-aware analyses for understanding the contributions of this paper. We describe *UCB-union* approach [30], *ECB-union* approach [3], and their multiset variants [4].

UCB-union and *ECB-union* approaches are computed as the least-fixed points of the following equation for the worst-case response time:

$$R_i^{(l+1)} = C_i + \sum_{\tau_h \in hp(i)} \lceil R_i^{(l)} / T_h \rceil \times (C_h + \gamma_{i,h}) \tag{1}$$

In Equation 1, $\gamma_{i,h}$ represents the CRPD due to a single job of a higher priority task τ_h executing within the worst-case response time of task τ_i. This term is computed differently for each of the CRPD approaches.

UCB-Union approach computes $\gamma_{i,h}^{ucbu}$ with the following equation, accounting that a job of τ_h causes a reload of each cache block which it may access and which is useful during the execution of at least one of the tasks from the range $[\tau_{h+1}, \tau_{h+2}, ..., \tau_i]$.

$$\gamma_{i,h}^{ucbu} = \Big|\big(\bigcup_{\tau_k \in aff(i,h)} UCB_k\big) \cap ECB_h\Big| \times BRT \tag{2}$$

ECB-Union approach computes $\gamma_{i,h}^{ecbu}$ with the following equation, accounting that a job of τ_h is preempted by all of the tasks with higher priority than τ_h, after the job directly preempted one of the tasks from the range $[\tau_{h+1}, \tau_{h+2}, ..., \tau_i]$. In this case, the preemption resulting in the highest number of evicted useful cache blocks is considered.

$$\gamma_{i,h}^{ecbu} = \max_{\tau_k \in aff(i,h)} \big\{\big|\big(\bigcup_{\tau_{h'} \in hpe(h)} ECB_{h'}\big) \cap UCB_k\big|\big\} \times BRT \tag{3}$$

Improving the above two approaches, Altmeyer et al. [4] introduced *UCB-Union multiset* and *ECB-Union multiset* which are computed as the least-fixed points of the following equation:

$$R_i^{(l+1)} = C_i + \sum_{\tau_h \in hp(i)} (\lceil R_i^{(l)}/T_h \rceil \times C_h + \gamma_{i,h}) \tag{4}$$

where $\gamma_{i,h}$ represents the CRPD due to each job of a higher priority task τ_h executing within the the worst-case response time of task τ_i.

ECB-Multiset approach computes $\gamma_{i,h}^{ecbum}$ accounting that τ_h can preempt each task $\tau_k \mid h < k \leq i$ the maximum number of times a single job of τ_k can be preempted by jobs of τ_h, for each job of τ_k that can be released within R_i, i.e. $\lceil R_k/T_h \rceil \times \lceil R_i/T_k \rceil$ times. This is accounted by the multiset $M_{i,h}$, which consists of the maximum CRPDs from jobs of τ_h on each preemptable job which can be released within R_i ($\uplus$ represents multiset union):

$$M_{i,h} = \biguplus_{\tau_k \in aff(i,h)} \Big(\biguplus_{\lceil R_k/T_h \rceil \times \lceil R_i/T_k \rceil} \big| UCB_k \cap \big(\bigcup_{\tau_{h'} \in hpe(h)} ECB_{h'}\big)\big| \Big) \tag{5}$$

Based on the above multiset, $\gamma_{i,h}^{ecbum}$ is computed as the sum of the maximum $\lceil R_i/T_h \rceil$ values from $M_{i,h}$, accounting that only $\lceil R_i/T_h \rceil$ jobs of τ_h can directly preempt and cause CRPD on the preemptable jobs accounted in $M_{i,h}$.

UCB-Multiset approach computes $\gamma_{i,h}^{ucbum}$ by first computing the multiset $M_{i,h}^{ucb}$ which consists of all possibly useful cache blocks from jobs which can be released within R_i, and have priority higher than or equal to τ_i, and lower than τ_h.

$$M_{i,h}^{ucb} = \biguplus_{\tau_k \in aff(i,h)} \Big(\biguplus_{\lceil R_k/T_h \rceil \times \lceil R_i/T_k \rceil} UCB_k \Big) \tag{6}$$

Next, this approach computes the multiset $M_{i,h}^{ecb}$ which consists of all possibly evicting cache blocks within jobs of τ_h that can be released within R_i. The following equation includes an instance of evicting cache block from τ_h for each job of τ_h that can be released within R_i:

$$M_{i,h}^{ecb} = \biguplus_{\lceil R_i/T_h \rceil} \Big(ECB_h \Big) \tag{7}$$

The upper bound on CRPD from jobs of τ_h preempting all jobs from τ_{h+1} to τ_i is equal to the size of intersection of those multisets, with accounted block reload time:

$$\gamma_{i,h}^{ucbum} = |M_{i,h}^{ucb} \cap M_{i,h}^{ecb}| \times BRT \tag{8}$$

The Combined-Multiset approach first computes the worst-case response time R_i^{ecbum} using Equation 4 and $\gamma_{i,h}^{ecbum}$, and similarly does with UCB-Union multiset, using Equation 4 and $\gamma_{i,h}^{ucbum}$ thus deriving R_i^{ucbum}. Then, the final result is computed as $\min(R_i^{ecbum}, R_i^{ucbum})$.

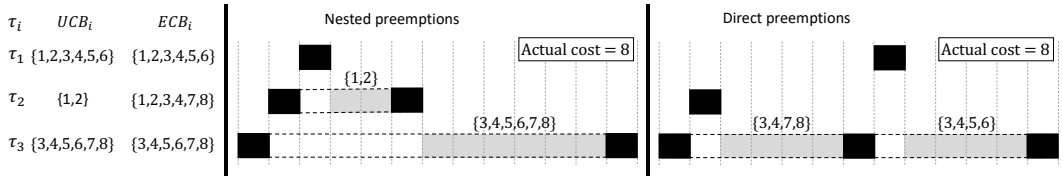


Figure 1 Example of the pessimistic CRPD estimation in both, UCB- and ECB-union based approaches. Notice that the worst-case execution time (black rectangles) is in reality significantly larger than CRPD, but the focus of the figure is rather on preemptions and CRPD depiction.

4 Pessimism in CRPD analyses based on UCB- and ECB-union approaches

In this section, we present the identified problems considering CRPD over-approximation when using UCB- and ECB-based approaches, including the multiset variants.

▶ **Problem 1.** *Combined approach over-approximates the CRPD bounds because all preemptions that may occur within a response time R_i are treated the same, with at most one method at a time. However, within all preemptions that may occur within R_i, different preemption sub-groups may be analysed with different analyses, thus the CRPD may be further reduced by computing the bounds for different preemption sub-groups individually instead of computing it with one method at a time for a single group of all preemptions.*

▶ **Problem 2.** *Combined approach accounts for CRPD from many different preemption combinations, which cannot occur together. This is presented with the following example. In Figure 1, we present three tasks τ_1, τ_2, and τ_3 with their respective sets of evicting and useful cache blocks. In the example, it is assumed that tasks τ_1 and τ_2 can be released at most once during the execution of τ_3 and that block reload time is equal to 1. Based on this, only two preemption combinations which result in the worst-case CRPD bound are possible: 1) A job of τ_2 directly preempts a job of τ_3, and a job of τ_1 directly preempts a job of τ_2 (nested preemptions), 2) A job of τ_2 directly preempts a job of τ_3, and a job of τ_1 directly preempts a job of τ_3 (direct preemptions).*

For each task, black rectangles in the figures represent the worst-case execution time, grey rectangles represent CRPD, whereas the sets of integer values above the grey rectangles represent the cache sets whose reloads must be accounted.

Considering the given cache block sets, the actual worst-case CRPD, based on the separately analysed preemption combinations, is:

- 8 (nested preemption): This is the case because τ_2 evicts cache blocks 3, 4, 7, and 8 which are then reloaded during the post-preemption execution of τ_3. After that, τ_1 evicts blocks 1 and 2 when preempting τ_2, which are reloaded during the post-preemption execution of τ_2, and also τ_1 evicts cache blocks 5 and 6 which are reloaded during the post-preemption execution of τ_3. Notice that although τ_1 also potentially evicts blocks 3, and 4 from τ_3, they are accounted as reloads only once within τ_3, because τ_3 is interrupted once and thus only one reload of each useful cache block within remaining execution of τ_3 is possible.
- 8 (direct preemptions): This is the case because τ_2 evicts cache blocks 3, 4, 7, and 8 from τ_3, and τ_1 evicts cache blocks 3, 4, 5, and 6 from τ_3.

Since any other preemption combination can be derived only by removing one of the preemptions accounted in the two above, the worst-case CRPD is equal to 8. However,

UCB-union based approaches (including the multiset variant) compute the following CRPD:

$$\gamma_{i,h}^{ucbu} = \left|\left(\bigcup_{\tau_k \in aff(i,h)} UCB_k\right) \cap ECB_h\right|$$
$$\gamma_{3,1}^{ucbu} = \left|\left(UCB_3 \cup UCB_2\right) \cap ECB_1\right| = 6 \ , \ \gamma_{3,2}^{ucbu} = \left|UCB_3 \cap ECB_2\right| = 4$$
$$\gamma_{3,1}^{ucbu} + \gamma_{3,2}^{ucbu} = 4 + 6 = 10 \quad , \text{ accounted reloads for blocks: } 1, 2, 3, 3, 4, 4, 5, 6, 7, 8$$

UCB-union based approaches compute CRPD upper bound of 10 reloads, thus approximating two block reloads over the safe upper bound (8 reloads) illustrated in Figure 1. Compared to the leftmost case from Figure 1, the accounted infeasible reloads are for blocks 3 and 4. Compared to the rightmost case, the accounted infeasible reloads are for blocks 1 and 2.

ECB-union based approaches (including the multiset variant) compute the CRPD upper-bound as follows:

$$\gamma_{i,h}^{ecbu} = \max_{\tau_k \in aff(i,h)} \left\{\left|\left(\bigcup_{\tau'_h \in hpe(h)} ECB_{h'}\right) \cap UCB_k\right|\right\}$$
$$\gamma_{3,1}^{ecbu} = \max_{\tau_k \in \{2,3\}} \left\{\left|(ECB_1) \cap UCB_k\right|\right\} = \max\left\{|ECB_1 \cap UCB_2|, |ECB_1 \cap UCB_3|\right\} = 4$$
$$\gamma_{3,2}^{ecbu} = \max\left\{\left|\left(ECB_1 \cup ECB_2\right) \cap UCB_3\right|\right\} = 6$$
$$\gamma_{3,1}^{ecbu} + \gamma_{3,2}^{ecbu} = 4 + 6 = 10$$

Similarly to UCB-union based approaches, ECB-union based approaches compute CRPD upper bound of 10 reloads, thus approximating two cache block reloads over the safe bound.

Even when the lowest bound of the two is selected, CRPD bound is over-approximated by accounting for two cache block reloads which cannot occur in a single combination of preemptions during runtime. CRPD over-approximation is further increased when multiple jobs of each task are introduced. In this paper, we propose a novel method for computing the CRPD and the worst-case response time, accounting for the above-described problems.

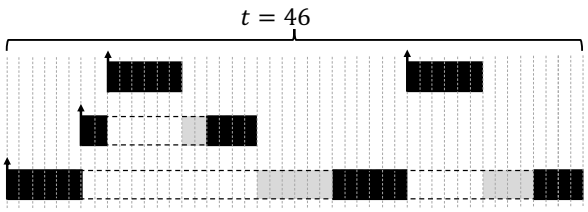
5 CRPD-aware Response-Time Analysis

In the remainder of the paper, when we refer to the term *preemption* we consider both, indirect (nested) and direct preemptions. We start with defining a cache-aware worst-case response time equation, slightly different than the existing ones. Formally, the response time analysis is defined as the least fixed-point of the following equation:

$$R_i^{(l+1)} = C_i + \gamma(i, R_i^{(l)}) + \sum_{\tau_h \in hp(i)} \left\lceil \frac{R_i^{(l)}}{T_h} \right\rceil C_h \tag{9}$$

Notice that unlike in the existing approaches, Equation 9 computes the CRPD upper bound $\gamma(i,t)$, which is a function that implicitly accounts for all preemptions that can occur within duration t, between the first i tasks of Γ. A CRPD upper bound on all preemptions that can occur within duration t can be computed more accurately by applying the following four steps that we describe in more detail in the remainder of this section:

1. Derive all possible preemptions which can occur within duration t, between the jobs of the first i tasks of Γ, (described in Subsection 5.1).
2. Divide the possible preemptions into partitions such that each partition accounts for single-job preemptions between the tasks, (described in Subsection 5.2)
3. Compute the CRPD bounds for each partition individually, (described in Subsection 5.3).
4. Sum the CRPD bounds of all partitions to obtain the cumulative CRPD bound on all possible preemptions within duration t, (described in Subsection 5.4).



τ_i	ECB_i	UCB_i	C_i	T_i	D_i
τ_1	{1,2,3,4,5,6}	{1,2,3,4,5,6}	6	26	7
τ_2	{1,2,3,4,7,8}	{1,2}	6	49	20
τ_3	{3,4,5,6,7,8}	{3,4,5,6,7,8}	16	50	46

Figure 2 Worst-case preemptions for τ_3 during the time duration $t = 46$.

To show an overview of how the proposed analysis works, we provide the running example from Figure 2, and we compute the upper bound on CRPD within the time duration $t = 46$. The analysis computes the bound as follows:

1. Derive all possible preemptions which can occur within duration t, between the jobs of the first i tasks of Γ.
   Example: Given the tasks from Figure 2, during the 46 time units, task τ_1 can preempt τ_3 at most two times, and it can preempt τ_2 at most once. Also, τ_2 can preempt τ_3 at most once. More formally, a single preemption from a job of τ_h on a job of τ_j is represented with an ordered pair (τ_h, τ_j). Thus, all possible preemptions, within 46 time units, can be represented by the following multiset of ordered preemption pairs:
   $$\Big\{(\tau_1,\tau_3),(\tau_1,\tau_3),(\tau_1,\tau_2),(\tau_2,\tau_3)\Big\} \tag{10}$$
2. Divide the possible preemptions into partitions such that each partition accounts for single-job preemptions between the tasks.
   Example: To represent the partitions, we generate the multiset Λ which consists of all possible preemptions, divided into partitions that account for single-job preemptions between the tasks. Given the possible preemptions from the multiset derived in the previous step, the multiset Λ of all partitions is:
   $$\Lambda = \Big\{ \underset{\downarrow}{\{(\tau_1,\tau_2),(\tau_1,\tau_3),(\tau_2,\tau_3)\}} \;,\; \underset{\downarrow}{\{(\tau_1,\tau_3)\}} \Big\}$$
   $$\text{Partition } 1 \qquad\qquad \text{Partition } 2$$
   The multiset Λ consists of two partitions (each represented as a set of preemptions), such that the first partition consists of the following preemptions $\{(\tau_1,\tau_2),(\tau_1,\tau_3),(\tau_2,\tau_3)\}$, meaning that it is possible that τ_1 preempts τ_2, that τ_1 preempts τ_3, and that τ_2 preempts τ_3. Jointly, the preemptions consist of all possible preemptions among the three tasks within a duration of 46 time units.
3. Compute the CRPD bounds for each partition individually.
   Example: As we showed in the previous section, when a single job of each task may preempt the other jobs with lower priority, the upper bound on CRPD is 8 time units. This is the upper bound on all preemptions accounted in *Partition* 1. Considering *Partition* 2, it consists of a single preemption, from a job of τ_1 on τ_3, and in this case the upper bound is 4 time units since the preemption may lead to the reloads of cache blocks $3, 4, 5$ and 6.
4. Sum the CRPD bounds of all partitions to obtain the cumulative CRPD bound on all possible preemptions within duration t.
   Example: The sum of upper bounds for *Partition* 1 and *Partition* 2 is $8 + 4 = 12$, which is the upper bound on all preemptions within 46 time units of the three shown tasks.

In the remainder of this section, we formally define the introduced terms, and prove that the proposed analysis results in a safe CRPD upper bound. The running example remains and it serves for better understanding on how the above values are computed and what they formally represent. This section is divided into the following subsections: 5.1 – describes the computation of upper bounds on the number of preemptions, 5.2 – describes the preemption partitioning, 5.3 – computation of CRPD bound for single partition, 5.4 – computation of CRPD bound for all preemptions, 5.5 – correctness proof for the computation of the worst-case response time, 5.6 – time complexity, and 5.7 – the additional computation for CRPD bound for single partition, based on finding the worst-case preemption combination.

5.1 Computing the upper bounds on the number of preemptions

▶ **Definition 1.** *An upper bound $E_j^h(t)$ on times a task τ_h can preempt τ_j $(h < j)$ within duration t is defined with the following equation:*

$$E_j^h(t) = \begin{cases} \left\lceil \frac{t}{T_h} \right\rceil & , \left\lceil \frac{t}{T_h} \right\rceil \leq \left\lceil \frac{t}{T_j} \right\rceil \\ \left\lceil \frac{t}{T_j} \right\rceil \times \left\lceil \frac{R_j}{T_h} \right\rceil & , \left\lceil \frac{t}{T_h} \right\rceil > \left\lceil \frac{t}{T_j} \right\rceil \end{cases} \tag{11}$$

▶ **Proposition 2.** *$E_j^h(t)$ is an upper bound on number of possible preemptions from τ_h on τ_j within duration t.*

Proof. Let us consider the following cases:
$\lceil \frac{t}{T_h} \rceil \leq \lceil \frac{t}{T_j} \rceil$: Each job of τ_h can preempt τ_j at most once, therefore the number of τ_h jobs which can be released within duration t is a safe bound on the number of preemptions from τ_h on τ_j within t.
$\lceil \frac{t}{T_h} \rceil > \lceil \frac{t}{T_j} \rceil$: An upper bound on preemptions from jobs of τ_h on a single job of τ_j is equal to $\lceil \frac{R_j}{T_h} \rceil$ since it is also an upper bound on number of times that jobs of τ_h can be released within the worst-case response time R_j of a single job. Since Equation 11 applies the bound $\lceil \frac{R_j}{T_h} \rceil$ on each job of τ_j which can be released within t, the proposition holds. ◀

5.2 Preemption partitioning

Once the all possible preemptions which can occur within duration t are identified, we divide them into partitions, such that no partition accounts for the same preemption pair of the first i tasks in Γ, and such that all partitions jointly account for all possible preemptions.

▶ **Definition 3.** *A multiset $\Lambda_{i,t}$ of partitions consisting of all possible preemptions that can occur within duration t, between the jobs of the first i tasks of Γ.*

$$\Lambda_{i,t} = \{\lambda_1, \lambda_2, ..., \lambda_z\} \text{ such that } \lambda_r = \{(\tau_h, \tau_j) \mid r \leq E_j^h(t)\} \tag{12}$$

In Equation 12, $\Lambda_{i,t}$ is defined as a multiset of of sets (partitions). Each set λ_r consists of possible preemptions and each preemption is represented as an ordered pair (τ_h, τ_j) where the first element represents the preempting, and the second element represents the preempted task. The multiset Λ is formed of exactly z partitions, where $z = \max\{E_j^h(t) \mid 1 \leq h < j \leq i\}$. Each partition consists of disjunct preemptions, meaning that no partition contains two same preemptioin pairs.
Example: Given the taskset from the running example in Figure 2, the multiset $\Lambda_{3,46}$ is computed as follows.

$$\Lambda_{3,46} = \{\lambda_1, \lambda_2\} \text{ where } \lambda_1 = \{(\tau_1, \tau_2), (\tau_1, \tau_3), (\tau_2, \tau_3)\} \text{ and } \lambda_2 = \{(\tau_1, \tau_3)\} \tag{13}$$

It is important to notice that the multiset union ($\uplus$) of all partitions in $\Lambda_{3,46}$ results in the multiset of all possible preemptions, e.g.,

$$\lambda_1 \uplus \lambda_2 = \{(\tau_1, \tau_2), (\tau_1, \tau_3), (\tau_2, \tau_3)\} \uplus \{(\tau_1, \tau_3)\} = \{(\tau_1, \tau_2), (\tau_1, \tau_3), (\tau_2, \tau_3), (\tau_1, \tau_3)\}$$

▶ **Proposition 4.** *Multiset $\Lambda_{i,t}$ consists of all possible preemptions that may occur within duration t, between the jobs of the first i tasks of Γ.*

Proof. Directly follows from Proposition 2 and Equation 12 since Equation 12 includes each possible preemption, occurable within duration t between the first i tasks of Γ, in one of the partitions of $\Lambda_{i,t}$. ◀

5.3 CRPD bound on preemptions from a single partition

As suggested in Section 3, considering the *Problem 1* of CRPD over-approximation, computing a bound for different preemption partitions individually, instead of computing it for all preemptions at once, may result in more precise CRPD estimations.

To achieve this, once the multiset $\Lambda_{i,t}$ of preemption partitions is computed, an upper bound on CRPD resulting from preemptions of a single partition $\lambda_r \in \Lambda_{i,t}$ can be computed by selecting the minimum CRPD bound among the results from UCB-Union and ECB-Union approaches. Here, we describe the improvements and adjustments on those approaches to compute CRPD bound from preemptions contained within a partition.

In the following equations, $aff(i, h, \lambda_r)$ represents a set of tasks with priorities higher than or equal to τ_i and lower than τ_h which can be preempted by τ_h according to preemptions represented in λ_r. Formally: $aff(i, h, \lambda_r) = \{\tau_k \mid (\tau_h, \tau_k) \in \lambda_r \ \wedge \ \tau_k \in hpe(i)\}$. Also, with $hp(i, \lambda_r)$ we denote a set of tasks with priorities higher than τ_i such that for each $\tau_h \in hp(i, \lambda_r)$ there is $(\tau_h, \tau_i) \in \lambda_r$.

First, we improve and adjust the ECB-Union approach, proposed by Altmeyer et al. [4]. In that approach, for a job of τ_h, it is accounted that it directly preempts one of the tasks from $aff(i, h, \lambda_r)$ set such that the maximum possible number of UCBs are evicted. In order for this approach to be correct, it is also accounted that tasks that can preempt τ_h also contribute to the evictions of useful cache blocks of the preempted task. This scenario is represented by a CRPD bound $\gamma^{ecbp}_{i,h}$. We further improve this formulation by accounting that a preemption from a single job of τ_h on any job of τ_k from $aff(i, h, \lambda_r)$ cannot cause more cache-block reloads than the maximum number of UCBs that can be evicted at a single preemption point of τ_k. The maximum number of UCBs at a single preemption point of τ_k is represented by ucb^{max}_k. The above translates to Equation 14.

$$\gamma^{ecbp}_{i,h}(\lambda_r) = \max_{\tau_k \in aff(i,h,\lambda_r)} \left\{ \min \left(\left| \left(\bigcup_{\tau_{h'} \in hp(h,\lambda_r) \cup \{\tau_h\}} ECB_{h'} \right) \cap UCB_k \right| , \ ucb^{max}_k \right) \right\} \quad (14)$$

We build the correctness of the proposed computation on the correctness of the standard ECB-Union method [4].

▶ **Proposition 5.** *$\gamma^{ecbp}_{i,h}(\lambda_r)$ is an upper bound on number of reloads that may be imposed by a direct preemption from τ_h on one of the tasks within $aff(i, h, \lambda_r)$ set.*

Proof. A direct preemption from τ_h on one of the tasks within $aff(i, h, \lambda_r)$ set cannot cause more reloads than the maximum number of UCBs of a preemptable task, which can be evicted by τ_h and all the tasks that may preempt τ_h. Also, such bound cannot be greater than the maximum number of useful cache blocks ucb^{max}_k that may be present at a single preemption point within a preemptable task, which concludes the proof. ◀

The ECB-Union based upper bound on all preemptions from λ_r is computed by summing all $\gamma_{i,h}^{ecbp}$ terms for each possibly preempting task, from τ_1 to τ_{i-1}:

$$\gamma_i^{ecbp}(\lambda_r) = \sum_{h=1}^{i-1} \gamma_{i,h}^{ecbp}(\lambda_r) \tag{15}$$

▶ **Proposition 6.** *$\gamma_i^{ecbp}(\lambda_r)$ is an upper bound on number of reloads that can be caused by preemptions from the partition λ_r.*

Proof. For each direct preemption from the preempting jobs of tasks from τ_1 to τ_{i-1} in λ_r, Equation 15 accounts that the upper-bounded number of cache-blocks is reloaded in one of the preemptable jobs, as follows from Proposition 5. Therefore, the proposition holds. ◀

Next, we adjust the UCB-Union approach, proposed by Tan and Mooney [30]. In this approach, for a job of τ_h, it is assumed that it can evict useful cache blocks from each task τ_k from the $aff(i,h,\lambda_r)$ set. However, since a single job of τ_h can directly or indirectly preempt each τ_k at only one of its preemption points, this cost can at most be equal to the sum of the number of maximum useful cache blocks at single preemption point of each task τ_k from $aff(i,h,\lambda_r)$. The above is formally represented with $\gamma_{i,h}^{ucbp}$ in the following equation:

$$\gamma_{i,h}^{ucbp}(\lambda_r) = \min\left(\left|\left(\bigcup_{\tau_k \in aff(i,h,\lambda_r)} UCB_k\right) \cap ECB_h\right|, \sum_{\tau_k \in aff(i,h,\lambda_r)} ucb_k^{max}\right) \tag{16}$$

We build the correctness of the proposed computation on the correctness of the standard UCB-Union method [30].

▶ **Proposition 7.** *$\gamma_{i,h}^{ucbp}(\lambda_r)$ is an upper bound on number of reloads within all tasks from $aff(i,h,\lambda_r)$, that may be imposed because of the cache-block accesses from a single job of τ_h.*

Proof. A job of τ_h cannot impose more than one cache block reload per cache-memory block m, such that $m \in ECB_h$, and $m \in UCB_k$ for any τ_k such that $\tau_k \in aff(i,h,\lambda_r)$, as follows from UCB-Union [30]. Also, since each task τ_k from $aff(i,h,\lambda_r)$ can be preempted by τ_h at only one of its preemption points, the maximum number of reloads from τ_h cannot be greater than the sum of the maximum numbers of useful cache blocks that may be present at a preemption point within each such task. This concludes the proof. ◀

The UCB-Union based upper bound on all preemptions from λ_r is also computed by summing all $\gamma_{i,h}^{ucbp}$ terms for each possibly preempting task from τ_1 to τ_{i-1}:

$$\gamma_i^{ucbp}(\lambda_r) = \sum_{h=1}^{i-1} \gamma_{i,h}^{ucbp}(\lambda_r) \tag{17}$$

▶ **Proposition 8.** *$\gamma_i^{ucbp}(\lambda_r)$ is an upper bound on number of reloads that can be caused by preemptions from the partition λ_r.*

Proof. For each possibly preempting job from τ_1 to τ_{i-1} in λ_r, Equation 17 accounts that the job leads to upper-bounded number of cache-block reloads in its possibly preemptable jobs, as follows from Proposition 7. Therefore, the proposition holds. ◀

The final upper bound $\gamma_i(\lambda_r)$ on CRPD from possible preemptions given in λ_r, between single jobs of the first i tasks from Γ, is defined as the least bound of the two.

$$\gamma_i(\lambda_r) = \min\left(\gamma_i^{ecbp}(\lambda_r),\ \gamma_i^{ucbp}(\lambda_r)\right) \times BRT \tag{18}$$

▶ **Proposition 9.** *$\gamma_i(\lambda_r)$ is an upper bound on CRPD from possible preemptions given in the partition λ_r.*

Proof. Follows from Propositions 6 and 8 since Equation 18 results in the least bound. ◀

5.4 CRPD bound on all preemptions within a time interval

Now, we define a computation for the CRPD bound on all preemptions which can occur within duration t, between the first i tasks of Γ.

▶ **Definition 10.** *An upper bound $\gamma(i,t)$ on CRPD of all preemptions, which can occur within duration t between the first i tasks of Γ, is defined with the following equation:*

$$\gamma(i,t) = \sum_{k=1}^{|\Lambda_{i,t}|} \gamma_i(\lambda_r) \tag{19}$$

▶ **Proposition 11.** *$\gamma(i,t)$ is an upper bound on CRPD of all preemptions which can occur within duration t between the first i tasks of Γ.*

Proof. Directly follows from Propositions 4 and 9, since Equation 19 is a sum of CRPD upper bounds of preemption partitions that jointly consist of all preemptions within t. ◀

5.5 Worst-case response time

In this subsection, we prove that the computed worst-case response time is an upper bound.

▶ **Theorem 12.** *R_i is an upper bound on worst-case response time of τ_i.*

Proof. By induction, over the tasks in Γ in a decreasing priority order.
Base case: $R_1 = C_1$, because $hp(i) = \emptyset$. Since C_i is the worst-case execution time of τ_1 the proposition holds.
Inductive hypothesis: Assume that for all $\tau_h \in hp(i)$, R_h is an upper bound on worst-case response time of τ_h.
Inductive step: We show that Equation 9 computes the worst-case response time of τ_i. Consider the least fixed point of Equation 9, for which $R_i = R_i^{(l)} = R_i^{(l+1)}$. At this point, the equation accounts for the following upper bounds and worst-case execution times:

- ◇ C_i, which is the worst-case execution time of τ_i, assumed by the system model.
- ◇ $\gamma(i, R_i)$, which is proved by Proposition 11 to be an upper bound on CRPD of all jobs which can be released within duration R_i, and have higher than or equal priority to τ_i.
- ◇ $\sum_{\forall \tau_h \in hp(i)} \lceil R_i/T_h \rceil C_h$, which is the worst-case interference caused by execution of all jobs of tasks with higher priority than τ_i without CRPD. Since we proved for all the factors which can prolong the response time of τ_i that they are accounted as the respective execution and CRPD upper bounds in Equation 9, then their sum results in an upper bound. ◀

5.6 Time complexity

The time complexity of the proposed analysis can be improved since in its current form Equation 12 explicitly creates all partitions which can lead to re-computation of CRPD bounds for many identical partitions. For this reason, we first define the matrix $A_{i,t}$, from which it is possible to identify how many repeated partitions there are, and compute CRPD bound for each distinct partition only once, as introduced in Algorithm 1.

▶ **Definition 13.** *A matrix $A_{i,t}$ of upper bounds on number of preemptions between each pair of tasks with higher than or equal priority to P_i which can occur within duration of t, is defined with the following equation:*

$$A_{i,t} = \ (a_{j,h}) \in \mathbb{N}^{i \times i} \mid a_{j,h} = \begin{cases} 0 & , j \leq h \\ E_j^h(t) & , j > h \end{cases} \tag{20}$$

▶ **Proposition 14.** *$A_{i,t}$ stores an upper-bounded number of preemptions, which can occur within t, between each pair of tasks with higher than or equal priority to P_i.*

Proof. Proposition 14 follows directly from Proposition 2 and the fact that τ_j cannot preempt any task τ_h of higher priority, or τ_j itself ($j \leq h$). ◀

Equation 20 defines a square matrix $A_{i,t}$ such that the number of rows and columns is equal to i, and each entry of the matrix represents the maximum number of preemptions from a task τ_h on τ_j within duration t.
Example: Given the taskset from Figure 2, a matrix of preemptions during 46 time units looks as follows:

$$A_{3,46} = \begin{array}{c} \begin{matrix} \tau_1 & \tau_2 & \tau_3 \end{matrix} \\ \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 2 & 1 & 0 \end{bmatrix} \end{array} \begin{matrix} \\ \tau_1 \\ \tau_2 \\ \tau_3 \end{matrix} \tag{21}$$

The element $a(2,1) = 1$ represents the maximum number of preemptions from τ_1 on τ_2 during 46 time units (note $R_2 = 14$).

Algorithm explanation: In a matrix $A_{i,t}$, there are at most $\frac{n*(n-1)}{2}$ values representing different numbers of possible preemptions among the tasks. Therefore, there are at most $\frac{n*(n-1)}{2}$ distinct partitions to be generated. In Algorithm 1, we define the procedure that first generates $A_{i,t}$ (line 3), and then generates distinct partitions one by one (lines 4 – 10). For each distinct partition, we compute the number sp of times a partition is repeated, then

Algorithm 1 Algorithm for computing the cumulative CRPD during a time interval of length t.

Data: Time duration t, task index i, Taskset Γ
Result: CRPD upper bound ξ on all jobs with priority higher than or equal to P_i, which can be released within duration t.

```
1  fn γ(i, t)
2      ξ ⟵ 0
3      A_{i,t} ⟵ generate the matrix of maximum preemption counts between the tasks
         (Equation 20)
4      while A_{i,t} ≠ 0_{i×i} do
5          sp ⟵ minimum value from A_{i,t}, greater than zero
6          λ ⟵ {(τ_h, τ_j) | sp ≤ a_{j,h}}
7          γ_i(λ) ⟵ compute the CRPD upper bound from the preemptions in λ
8          ξ ⟵ ξ + sp × γ_i(λ)
9          A_{i,t} ⟵ decrease all values, greater than zero, by sp
10     end
11     return ξ
```

compute the partition (line 6), and account for its CRPD bound *sp* times in the cumulative CRPD bound ξ that is updated for each distinct partition (lines 7 and 8). After this, the partitioned preemptions are removed (line 9), and the next distinct partition is computed until no more preemptions are left to be partitioned. Formally, termination criteria is satisfied when $A_{i,t}$ equals to the zero matrix $0_{i\times i}$. At the end, the algorithm results in the same CRPD bound as Equation 19. Using this algorithm, the time complexity is $\mathcal{O}(n^3 * x)$, where the complexity of computation at line 6 is $\mathcal{O}(x)$.

5.7 CRPD computation using preemption scenarios

In this section, we propose an alternative computation for the upper bound $\gamma_i(\lambda)$ on CRPD of preemptions in the partition λ. The goal is to compute the CRPD bound from a single worst-case preemption combination among the preemptions from λ, addressing *Problem 2* from Section 4. To achieve this, we first formally define the following terms:

- *Preemption scenario* (τ_i, PT), and its CRPD upper bound $\gamma(\tau_i, PT)$,
- *Preemption combination* Π^c_λ, and its CRPD upper bound $\gamma(\Pi^c_\lambda)$.

Informally, we define a preemption combination as a set of feasible preemptions where only one job of each task is involved, such that all accounted preemptions are present in a partition λ. Before being able to formally define a preemption combination, we first formally define a preemption scenario.

▶ **Definition 15** (Preemption scenario (τ_i, PT))**.** *A preemption scenario represents a single interruption due to preemption of a task and it is defined as an ordered pair (τ_i, PT), where τ_i represents the preempted (interrupted) task, and PT is a set of preempting tasks which execute after the interruption at τ_i and before the immediate resumption of τ_i. Formally, for each preemption scenario (τ_i, PT) it holds that $PT \subseteq hp(i)$.*

Example: Given the example from Fig. 2, the first preemption scenario in τ_3 is $(\tau_3\ ,\ \{\tau_1, \tau_2\})$, and the second preemption scenario is $(\tau_3\ ,\ \{\tau_1\})$. Also, in the same figure, τ_2 is preempted once and this preemption scenario is $(\tau_2\ ,\ \{\tau_1\})$.

In order to compute the upper bound on CRPD on τ_i, resulting from one interruption scenario, the ordering of the preempting tasks is not important. All of them are equally capable of evicting cache blocks of τ_i between its preemption and resumption, regardless of their ordering.

▶ **Definition 16** (CRPD of a preemption scenario $\gamma(\tau_i, PT)$)**.** *An upper bound $\gamma(\tau_i, PT)$ on the CRPD of a preempted task τ_i resulting from a preemption scenario (τ_i, PT) is:*

$$\gamma(\tau_i, PT) = |\, UCB_i \cap \bigcup_{\tau_h \in PT} ECB_h| \times BRT \tag{22}$$

▶ **Proposition 17.** *$\gamma(\tau_i, PT)$ is an upper bound on the CRPD of a preempted task τ_i resulting from a preemption scenario (τ_i, PT).*

Proof. Since Equation 22 accounts that each UCB from τ_i is definitely reloaded with the worst-case block reload time if there is a corresponding evicting cache block from any of the preempting tasks within a scenario, the proposition holds. ◀

Example: Given the preemption scenario $(\tau_3\ ,\ \{\tau_1, \tau_2\})$, the upper bound on CRPD of τ_3 is:

$$\gamma(\tau_3\ ,\ \{\tau_1, \tau_2\}) = |\ UCB_3 \cap \{ECB_1 \cup ECB_2\}\ | = 6$$

A safe upper bound on CRPD of τ_i resulting from a preemption scenario (τ_i, PT) can be tightened even more if the low-level task analysis can provide more detailed information on UCBs and ECBs at different program points. In such case, the bound is computed as the maximum intersection of UCBs from a single point within τ_i and the evicting cache blocks of tasks in PT. This is the case because Equation 22 considers a single preempted point and tasks which may evict cache blocks before preempted task resumes to execute. On the other hand, many existing approaches consider multiple preemption scenarios at once, and therefore this improvement is not applicable in their case, as shown by Shah et al. [27].

Given the formal definition of preemption scenarios, now we can define a preemption combination. With this definition, we need to insure that a preemption combination consists only of preemption scenarios which account for interactions between a single job of each task. Therefore, we need to insure that a preemption combination does not include two preemption scenarios which are mutually exclusive, given the constraint of using only single jobs.

▶ **Definition 18** (Preemption combination Π^c)**.** *A preemption combination Π^c is defined as a set of disjoint non-empty preemption scenarios between single jobs of tasks in Γ such that:*

1) If there are two preemption scenarios (τ_j, PT^j) and (τ_l, PT^l) in Π^c such that $\tau_h \in PT^j \cap PT^l$ and $P_l < P_j$, it implies that $\tau_j \in PT^l$.

2) Each preempting task $\tau_h \in PT$, where $(\tau_j, PT) \in \Pi^c$, can be in at most one preemption scenario imposed on τ_j.

The first constraint refers to a case: If a single job of task τ_h preempts single jobs of tasks τ_j and τ_l, where $P_j > P_l$, then that job of τ_l is definitely preempted by the τ_j job.
The second constraint accounts that an additional preemption scenario with τ_h implies that Π^c accounted for two jobs of τ_h preempting a job of τ_j, while the definition accounts for at most one job of each task.
Example: Given a definition of a preemption combination, one possible combination is: $\{(\tau_3\ ,\ \{\tau_1\})\ ,\ (\tau_3\ ,\ \{\tau_2\})\}$ which describes the preemption scenario where τ_1 directly preempts τ_3 at one preemption point, while τ_2 directly preempts τ_3 at another preemption point. However, the set $\{(\tau_3\ ,\ \{\tau_1\})\ ,\ (\tau_2\ ,\ \{\tau_1\})\}$ is not a preemption combination, because it describes the case where a job of τ_3 is preempted by a job of τ_1, and a job of τ_2 is preempted by a job of τ_1, while a job of τ_2 does not preempt a job of τ_3. Since this is the case, more than two jobs of τ_1 are accounted, which violates Definition 18.

▶ **Definition 19** (Preemption combination consistent with λ)**.** *We say that Π^c is a preemption combination consistent with the preemption partition λ iff for any preemption scenario $(\tau_j, PT) \in \Pi^c$ the preemptions captured by the scenario are possible, i.e. present in λ. Formally: $\forall(\tau_j, PT) \in \Pi^c_i : \forall\tau_h \in PT : (\tau_j, \tau_h) \in \lambda$.*

Example: Given the preemption partition $\lambda = \{(\tau_1, \tau_2), (\tau_1, \tau_3), (\tau_2, \tau_3)\}$, a preemption combination consistent with λ is $\{(\tau_3\ ,\ \{\tau_1\})\ ,\ (\tau_3\ ,\ \{\tau_2\})\}$ since it describes preemption scenarios made of preemptions that are possible, i.e. present in λ.

▶ **Definition 20** (CRPD $\gamma(\Pi^c)$ of a preemption combination)**.** *An upper bound $\gamma(\Pi^c)$ on the CRPD of a single preemption combination Π^c is defined as the sum of upper bounds of all preemption scenarios in Π^c. Formally, it is defined as:*

$$\gamma(\Pi^c) = \sum_{(\tau_k, PT) \in \Pi^c} \gamma(\tau_k, PT) \tag{23}$$

▶ **Proposition 21.** *$\gamma(\Pi^c)$ is an upper bound on CRPD of preemptions accounted within Π^c.*

Proof. A combination consists of a number of preemption scenarios representing the preemptions from preempting tasks on different preemption points. Following from Proposition 17, a sum of CRPD upper bounds of each task interruption in a combination is an upper bound on CRPD of all preemptions accounted in a combination, which concludes the proof. ◀

Example: Given the preemption combination of two direct preemptions from Figure 1, CRPD upper bound is: $\gamma\big(\{(\tau_3\ ,\ \{\tau_1\})\ ,\ (\tau_3\ ,\ \{\tau_2\})\}\big) = \gamma(\tau_3\ ,\ \{\tau_1\}) + \gamma(\tau_3\ ,\ \{\tau_2\}) = 4+4=8$. Given the preemption combination of a nested preemption from the figure, it is: $\gamma\big(\{(\tau_3\ ,\ \{\tau_1,\tau_2\})\ ,\ (\tau_2\ ,\ \{\tau_1\})\}\big) = \gamma(\tau_3\ ,\ \{\tau_1,\tau_2\}) + \gamma(\tau_2\ ,\ \{\tau_1\}) = 6+2=8$.

Now, we can imagine a set which consists of all possible preemption combinations which are consistent with preemptions enlisted in λ. Then, among all the generated preemption combinations, we find one which results in the worst-case CRPD and declare that as a safe upper-bound, since that is the maximum obtainable CRPD value among all the possible preemption combinations.

However, to generate such complete set of all possible preemption combinations is computationally inefficient. A potential solution can come from the fact that it is enough to compute a subset of the complete set of combinations, as long as we are sure that no greater CRPD value can be obtained in the remaining, unaccounted combinations. We show this with the following *example:* Given a set of possible preemptions $\lambda = \{(\tau_1,\tau_2),(\tau_1,\tau_3),(\tau_2,\tau_3)\}$ from Figure 1, let us consider the following set of two combinations:

$$\Pi_{3,\lambda} = \Big\{\ \ \Big\{(\tau_3\ ,\ \{\tau_1\})\ ,\ (\tau_3\ ,\ \{\tau_2\})\Big\}\ \ ,\ \ \Big\{(\tau_3\ ,\ \{\tau_1,\tau_2\})\ ,\ (\tau_2\ ,\ \{\tau_1\})\Big\}\ \ \Big\}$$

$\Pi_{3,\lambda}$ consists of: 1) a combination of direct preemption scenarios, and 2) a combination of nested preemption. Any other possible preemption combination, from preemptions in λ, can only be derived by omitting at least one preemption from a preemption scenario from one of the two combinations given in $\Pi_{3,\lambda}$. E.g. preemption combination $\{(\tau_3\ ,\ \{\tau_1,\tau_2\})\}$ is equal to and results in the same CRPD as $\{(\tau_3\ ,\ \{\tau_1,\tau_2\})\ ,\ (\tau_2\ ,\ \emptyset)\}$. Also, all of the preemption scenarios from $\{(\tau_3\ ,\ \{\tau_1,\tau_2\})\}$ are already included in the second combination. Therefore, all the other possible combinations cannot result in a greater CRPD value than those in $\Pi_{3,\lambda}$, meaning that this subset is sufficient to compute a safe CRPD.

A preemption combination which is constructed by adding a preemption scenario to any of the combinations in $\Pi_{3,\lambda}$ cannot be obtained. This is the case because in the first combination, it is accounted that τ_3 is preempted by both tasks, at two different points, accounting for all preemptions in λ where τ_3 can be preempted, i.e. (τ_1,τ_3) and (τ_2,τ_3). In this case, τ_2 cannot be preempted considering preemption $(\tau_1,\tau_2) \in \lambda$ as shown in Definition 18. In the second combination, τ_3 is interrupted once while both tasks preempt it, and τ_2 is preempted by τ_1, meaning that all preemptions from λ are accounted.

In order to define a safe set of combinations $\Pi_{i,\lambda}$, such that at least one of those combinations may result in the worst-case CRPD, we first introduce a term of **set partitioning**[1] in order to represent different ways one task may be preempted by the others. *Example:* Given the task τ_3, set partitions of a set $\{\tau_1,\tau_2\}$ of its potentially preempting tasks are: 1) $\{\{\tau_1,\tau_2\}\}$, and 2) $\{\{\tau_1\},\{\tau_2\}\}$.

Given a preemptable task τ_k, and a set of its possibly preempting tasks PT, all set partitions of PT represent all the ways τ_k may be preempted such that each task from PT preempts τ_k. This is the case because set partitions represent all the ways a set can

[1] Set partitioning is a mathematical concept sometimes also known as Bell partitioning [1, 18] named after Eric Temple Bell. There are many fast algorithms for generating set partitions, e.g. [13, 14].

be grouped in non-empty subsets, such that each set element is included in exactly one subset. Analogically, in this paper, each set partition is transformed into a preemption combination defining one way how a task (e.g. τ_3 above) can be preempted. Each set partition consists of subsets, and each subset represents a preemption scenario on the preempted task. This transformation is formally defined in function *generateCombs*(PT, τ_k) in Algorithm 2. *Example:* Considering different ways τ_3 can be preempted, set partition $\{\{\tau_1, \tau_2\}\}$ consists of a single subset, and forms a preemption combination $\{(\tau_3, \{\tau_1, \tau_2\})\}$, while set partition $\{\{\tau_1\}, \{\tau_2\}\}$ consists of two subsets and forms a preemption combination $\{(\tau_3, \{\tau_1\}), (\tau_3, \{\tau_2\})\}$ with two preemption scenarios: $(\tau_3, \{\tau_1\})$, and $(\tau_3, \{\tau_2\})$.

▶ **Proposition 22.** *Given a preemptable task τ_k, and a set of possibly preempting tasks PT, any combination of preemptions on τ_k will result in a less than or equal CRPD than any combination generated from generateCombs(PT, τ_k).*

Proof. By contradiction: Let us assume that there is a preemption combination Π_l^c representing the ways how τ_k can be preempted by tasks from PT, and that Π_k^c can result in a greater CRPD than any combination derived from *generateCombs*(PT, τ_k). The combinations generated from *generateCombs*(PT, τ_k) represent all the ways τ_k may be preempted such that each task from PT preempts τ_k since set partitions represent all the ways a set can be grouped in non-empty subsets, such that each element is included in exactly one subset. Thus, Π_k^c must omit at least one preemption, compared to at least one preemption combination derived from *generateCombs*(PT, τ_k). The initial assumption therefore contradicts Proposition 21 because Π_k^c cannot impose larger CRPD than the corresponding preemption combination from *generateCombs*(PT, τ_k), which accounts for the same preemptions as in preemption scenarios from Π_k^c and at least one additional preemption compared to Π_k^c. ◀

Using the concept of set partitioning to represent the ways a single task may be preempted, we generate a set $\Pi_{i,\lambda}$ of preemption combinations on how all tasks from τ_i to τ_1 can interact among each other:

▶ **Definition 23.** *By $\Pi_{i,\lambda}$ we denote the result from Algorithm 2, i.e. the set of preemption combinations between the first i tasks of Γ such that each combination is consistent with λ.*

We describe Algorithm 2 in more details and we use a running example from Figure 2 to show the algorithm walk-through in Figure 3. As stated before, for each τ_k, from τ_i to τ_1, the algorithm first generates possible combinations on how τ_k can be preempted, using set partitioning (line 3). This process is defined in function *generateCombs* (line 7) and it translates the set partitions of possibly preempting tasks on τ_k, into different ways τ_k can be preempted, which is represented with a set of preemption combinations (line 16). Then, for each of those combinations, the algorithm performs *extendCombs* (line 4), which is a function that updates the existing preemption combinations, with further preemption scenarios that are possible on the preempting tasks of τ_k. Take for example the preemption combination Π^c, given in Figure 3.

The combination represents the case where τ_4 is preempted at one preemption point, by all of its three possibly preempting tasks (τ_1, τ_2, τ_3). In the figure, this is represented by one arrow (standing for one preempted point of τ_4) and tasks preempting a point (above the arrow). Function *extendCombs*() further computes possible ways of preempting τ_3 since τ_3 is the lowest-priority preempting task from the preemption scenario $(\tau_4, \{\tau_1, \tau_2, \tau_3\})$. Those ways are represented with a set Π_3' of preemption combinations. After this, the function updates the preemption combinations with a Cartesian product of the two. Therefore, on the right side of the figure, you may notice that now we have two new combinations, updating the Π_c with different ways τ_3 can be preempted. The topmost combination can be updated further on, since preemption scenario $(\tau_3, \{\tau_1, \tau_2\})$ can be updated with additional scenario on how τ_2 can be preempted by τ_1. This is eventually computed within the algorithm because condition

in line 18 insures that all combinations are updated until no new preemption scenario can be added to any of the existing preemption combinations. More formally, this criteria is satisfied when for each preemption scenario (τ_x, PT) within any preemption combination from $\Pi_{i,\lambda}$, function $extended?((\tau_x, PT))$ yields true ($\top$), meaning that all preemption scenarios are extended.

▶ **Proposition 24.** *$\Pi_{i,\lambda}$ is a safe set of preemption combinations between the single jobs of the first i tasks in Γ, i.e. there is no preemption combination consistent with λ with a higher CRPD than the maximum CRPD of the combinations in $\Pi_{i,\lambda}$,*

Algorithm 2 Algorithm that generates a set $\Pi_{i,\lambda}$ of preemption combinations.

```
   Data: Set of possible preemption pairs λ,
         task index i
   Result: A set Π_{i,λ} of preemption
           combinations consistent with λ
1  Π_{i,λ} ⟵ ∅
2  for k ← i to 2 by −1 do
3  |  Π'_{k,λ} ⟵ generateCombs(hp(k) , τ_k)
4  |  Π_{i,λ} ⟵ Π_{i,λ} ∪ extendCombs(Π'_{k,λ})
5  end
6  return Π_{i,λ}
7  fn generateCombs(PT , τ_k)
8  |  Π_{k,λ} ⟵ ∅
9  |  PT_{k,λ} ⟵ remove those tasks from
   |    PT that cannot preempt τ_k according
   |    to λ
10 |  partitions(PT) ⟵ generate all
   |    possible partitions of a set PT_{k,λ},
   |    representing ways a job of τ_k can be
   |    preempted.
11 |  for each partition ∈ partitions(PT)
12 |  |  Π^c_k ⟵ ∅
13 |  |  for each subset ∈ partition
14 |  |  |  Π^c_k ⟵ Π^c_k ∪ {(τ_k, subset)}
15 |  |  Π_{k,λ} ⟵ Π_{k,λ} ∪ Π^c_k
16 |  return Π_{k,λ}
17 fn extendCombs(Π_{q,λ})
18 |  while ∃Π^c ∈ Π_{q,λ} : ∃(τ_r, PT) ∈ Π^c |
   |    extended?((τ_r, PT) , Π^c) = ⊥ do
19 |  |  τ_l ⟵ lowest-priority task in PT
20 |  |  Π'_l ⟵ generateCombs(PT \ τ_l , τ_l)
21 |  |  for each
   |  |    Π^c ∈ Π_{q,λ} | (τ_r, PT) ∈ Π^c
22 |  |  |  Π_{q,λ} ⟵ Π_{q,λ} ∪ (Π^c × Π'_l)
23 |  |  |  Π_{q,λ} ⟵ Π_{q,λ} \ Π^c
24 |  end
25 |  return Π_{q,λ}
26 fn extended?( (τ_r, PT) , Π^c )
27 |  τ_l ⟵ lowest-priority task in PT
28 |  if ∃(τ_x, PT') ∈ Π^c | τ_x = τ_l then
29 |  |  return ⊤
30 |  else return ⊥;
```

Data: $\lambda = \{(\tau_1,\tau_2),(\tau_1,\tau_3),(\tau_2,\tau_3)\}$, $i = 3$
Algorithm run:
$\Pi_{3,\lambda} \longleftarrow \emptyset$
for $k = 3$
$\Pi'_{3,\lambda} \longleftarrow generateCombs(hp(3),\tau_3)$
$\longleftarrow \{ \{(\tau_3,\{\tau_1\}) , (\tau_3,\{\tau_2\})\} , \{(\tau_3,\{\tau_1,\tau_2\})\} \}$
$\Pi_{3,\lambda} \longleftarrow \emptyset \cup extendCombs(\Pi'_{3,\lambda})$
$\longleftarrow \{ \{(\tau_3,\{\tau_1\}) , (\tau_3,\{\tau_2\})\} , \{(\tau_3,\{\tau_1,\tau_2\}) , (\tau_2,\{\tau_1\})\} \}$
for $k = 2$
$\Pi'_{2,\lambda} \longleftarrow generateCombs(hp(2),\tau_2)$
$\longleftarrow \{ \{(\tau_2,\{\tau_1\})\} \}$
$\Pi_{3,\lambda} \longleftarrow \{ \{(\tau_3,\{\tau_1\}) , (\tau_3,\{\tau_2\})\} , \{(\tau_3,\{\tau_1,\tau_2\}) , (\tau_2,\{\tau_1\})\} \} \cup \{ \{(\tau_2,\{\tau_1\})\} \}$
return $\Pi_{3,\lambda}$

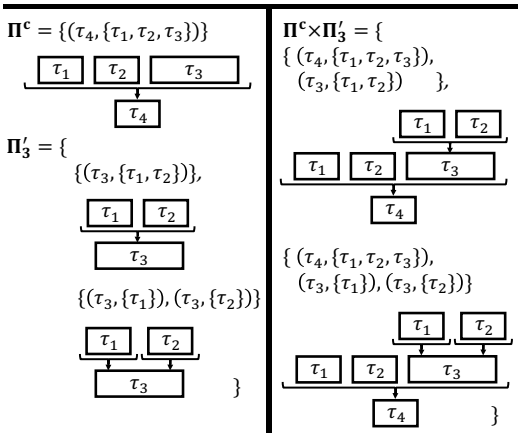


Figure 3 Top: Algorithm walktrough with an example from Figure 2. Bottom: Example for extending the combination Π^c of four tasks.

Proof. By contradiction: Let us assume that there is a preemption combination Π^c_o between the single jobs of the first i tasks, consistent with λ, which can result in a higher CRPD than any of the combinations in $\Pi_{i,\lambda}$. For each task τ_k such that $(1 < k \leq i)$, it is accounted that τ_k experiences the worst-case CRPD at one of the generated combinations, as follows from

Proposition 22 and line 3. Each such a combination is extended in line 4, accounting for further worst-case preemption scenarios on how all the preempting tasks can be preempted, and Algorithm 2 stops only when no more preemption scenarios can be generated and added to a set of preemption combinations $\Pi_{i,\lambda}$. This further implies that Π_o^c must omit at least one preemption from at least one of its preemption scenarios compared to any combination from $\Pi_{i,\lambda}$. Moreover, by construction of Algorithm 2, there is a preemption combination Π_w^c in $\Pi_{i,\lambda}$ which is a superset over the Π_o^c, i.e. there is a mapping of preemption scenarios between Π_w^c and Π_o^c such that each preemption scenario of Π_w^c includes same preemptions as the respective scenario in Π_o^c, but may also include additional ones not accounted by Π_o^c. As follows from Propositions 21 and 22, Π_o^c can only result in CRPD less than or equal to the one from Π_w^c. This contradicts the initial assumption. ◀

Finally, we can compute an upper bound on CRPD resulting from the worst-case preemption combination consisting of the preemptions in λ, with the following equation:

$$\gamma_i(\lambda) = \max_{\Pi^c \in \Pi_{i,\lambda}} \gamma(\Pi^c) \tag{24}$$

▶ **Proposition 25.** *$\gamma_i(\lambda)$ is an upper bound on CRPD from preemptions given in the partition λ, between the single jobs of the first i tasks from Γ.*

Proof. Equation 24 computes the maximum upper bound from all preemption combinations accounted by $\Pi_{i,\lambda}$. Then, following from Propositions 21 and 24, the proposition holds. ◀

Example: Given a set of possible preemptions $\lambda = \{(\tau_1, \tau_2), (\tau_1, \tau_3), (\tau_2, \tau_3)\}$ from Figure 1 and continuing from the example after Proposition 21, the upper bound on CRPD resulting from preemptions in λ is computed as $\gamma(\lambda) = \max(\{8, 8\}) = 8$.

5.8 Adjustment for LRU caches

The proposed methods can also be used for set-associative LRU caches with a single modification, as shown by Altmeyer et al. [4, 6]. In case of LRU set-associative cache, a cache-set may contain several useful cache blocks, e.g., $UCB_2 = \{1, 2, 2, 2\}$ means that τ_2 contains three cache blocks in cache-set 2, and one UCB in cache set 1. Upon preemption, one ECB of the pre-empting task may suffice to evict all UCBs of the same cache-set, meaning that $ECB_1 = \{1, 2\}$ may evict all cache blocks of τ_2. Therefore, the current notion of the ECBs and UCBs of a task may remain unchanged if a bound on CRPD due to preemption from τ_h on τ_i is defined as: $UCB_i \cap' ECB_h$ where the result is a multiset that contains each element from UCB_i if it is also in ECB_h, e.g. $UCB_2 \cap' ECB_1 = \{1, 2, 2, 2\} \cap' \{1, 2\} = \{1, 2, 2, 2\}$. In case of FIFO and PLRU cache replacement policies, the concepts of useful and evicting cache blocks cannot be applied, as shown by Burguiere et al. [10].

6 Evaluation

In this section, we show the evaluation results. The goal of the evaluation was to investigate to what extent the proposed method is able to identify schedulable tasksets upon the analysis of the cache-related preemption delays. We compared the state-of the art analyses for CRPD: (*ECB-Union Multiset*), (*UCB-Union Multiset*) methods, and (*Combined Multiset*), with two versions of the proposed method, i.e. (*Partitioning-ver1*) which computes CRPD according to Section 5.3, and the version (*Partitioning-ver2*) which computes CRPD from the worst-case preemption combination, presented in Section 5.7.

Table 1 Task characteristics obtained with LLVMTA [17] analysis tool used on Mälardalen [16] and TACLe [15] benchmark tasks.

Task (TACLe Bench.)	WCET	ECB	UCB	Max
app/lift	13592762	250	125	23
app/powerwindow	55842069	256	120	25
kernel/binarysearch	2860	43	19	18
kernel/bsort	3332496	42	30	29
kernel/complex_update	8190	36	28	27
kernel/countnegative	260303	78	45	45
kernel/fft	493123975	103	87	52
kernel/filterbank	38302875	164	151	66
kernel/fir2dim	86737	212	197	116
kernel/iir	3307	41	32	31
kernel/insertsort	16148	50	35	28
kernel/jfdctint	9043	115	107	54
kernel/lms	1758977	82	56	23
kernel/ludcmp	97908	173	137	44
kernel/matrix1	248058	48	43	42
kernel/md5	367421931	256	149	72
kernel/minver	67700	254	173	46
kernel/pm	141189221	256	247	45
kernel/prime	386343	80	54	41
kernel/sha	28380272	253	185	31
kernel/st	1763900	161	80	43
sequential/adpcm_dec	52530	233	145	59
sequential/adpcm_enc	58861	236	158	75
sequential/audiobeam	6434692	256	212	46
sequential/cjpeg_transupp	535718162	256	256	103
sequential/cjpeg_wrbmp	1610145	138	80	38
sequential/dijkstra	39781181581	151	80	46
sequential/epic	7423276281	256	256	107
sequential/g723_enc	22919200	256	154	81
sequential/gsm_dec	3744323	256	236	69
sequential/gsm_encode	2115350	256	256	118
sequential/h264_dec	24979237	256	166	29
sequential/huff_dec	9360435	254	144	44
sequential/mpeg2	130756234186	256	256	154
sequential/ndes	996427	253	167	39

...continuation	WCET	ECB	UCB	Max
sequential/petrinet	39951	256	92	2
sequential/ri._dec	1811372648	256	173	44
sequential/ri._enc	39467989	256	181	44
sequential/statemate	1949343	256	91	1
sequential/susan	2051176771	256	255	79
Task (Mälardalen Bench.)	**WCET**	**ECB**	**UCB**	**Max**
adpcm	82492494	256	230	103
bs	3052	43	23	20
bsort100	3146185	57	40	30
cnt	127558	123	58	44
compress	1090099	247	150	63
cover	74509	256	38	15
crc	1376054	121	62	30
edn	739866	256	222	123
expint	2161270	117	47	29
fdct	10258	126	113	62
fft1	271733	222	154	63
fibcall	8406	28	16	16
fir	12413071	94	42	21
insertsort	11291	29	16	15
janne_complex	33778	39	28	27
jfdctint	21742	132	122	54
lcdnum	6100	51	11	9
lms	10178805	242	134	38
ludcmp	116312	210	168	44
matmult	1447379	85	51	31
minver	67157	256	178	47
ndes	1050163	253	176	38
ns	126865	55	37	34
nsichneu	201969	256	183	2
prime	7782800	75	47	33
qsort.	163089	142	83	39
qurt	71655	130	40	26
select	6306	159	73	55
sqrt	22436	53	21	12
st	3701746	192	95	52
statemate	41579	256	105	1
ud	355318	194	151	39

As shown by Shah et al. [27], an evaluation of the CRPD-aware methods should consider task parameters derived by using the existing low-level analysis tools. Therefore, in this paper we use the suggested task parameters that are derived with LLVMTA analysis tool [17], used on Mälardalen [16] and TACLe [15] benchmark tasks. The derived task characteristics are shown in Table 1, and they are: worst-case execution time, expressed in terms of wall-clock time, set of evicting cache blocks, set of definitely useful cache blocks (shown in the table as the size of the respective sets – ECB and DC-UCB), and the maximum number (Max DC-UCB) of definitely useful cache blocks per any program point of a task. The characteristics were derived with assumed direct-mapped instruction cache and a data scratchpad. The assumed cache memory consists of 256 sets with line size equal to 8 bytes, while block reload time is equal to 22 cycles. For more details about the low-level analysis refer to [27].

Tasksets are generated by randomly selecting a subset of tasks from one of the two benchmarks, Mälardalen or TACLe, specified in each figure. We generated 1000 tasksets for each pair of selected utilisation and taskset size. Since the task binaries were analysed individually, they all start at the same address (mapping to cache set 0). In a multi-task scheduling situation this can hardly be a case because the ECB and UCB placement is determined by their respective locations in memory. We took this into account by randomly shifting the cache set indices, e.g. the ECB in cache set i is shifted to the cache line equal to $(i + random(256))$ *modulo* 256. Task utilisations were generated using U-Unifast algorithm, as proposed by Bini et al. [8]. Minimum inter-arrival times were then computed using equation $T_i = C_i/U_i$, while the deadlines are assumed to be implicit, i.e. $D_i = T_i$. Task priorities were assigned using deadline-monotonic order.

In Figure 4, on the leftmost plot, we show the schedulability results of an experiment where we generated tasksets of size 9, from the Mälardalen tasks (top left), and TACLe tasks (bottom left). For each generated taskset, its utilisation was varied from 0.5 to 1, by step of 0.01. The results show that *Partitioning-ver2* and *Partitioning-ver1* identify the highest number of schedulable tasksets, even up to 23% more for *Partitioning-ver2*, and 20% more for *Partitioning-ver1*, compared to *Combined multiset*.

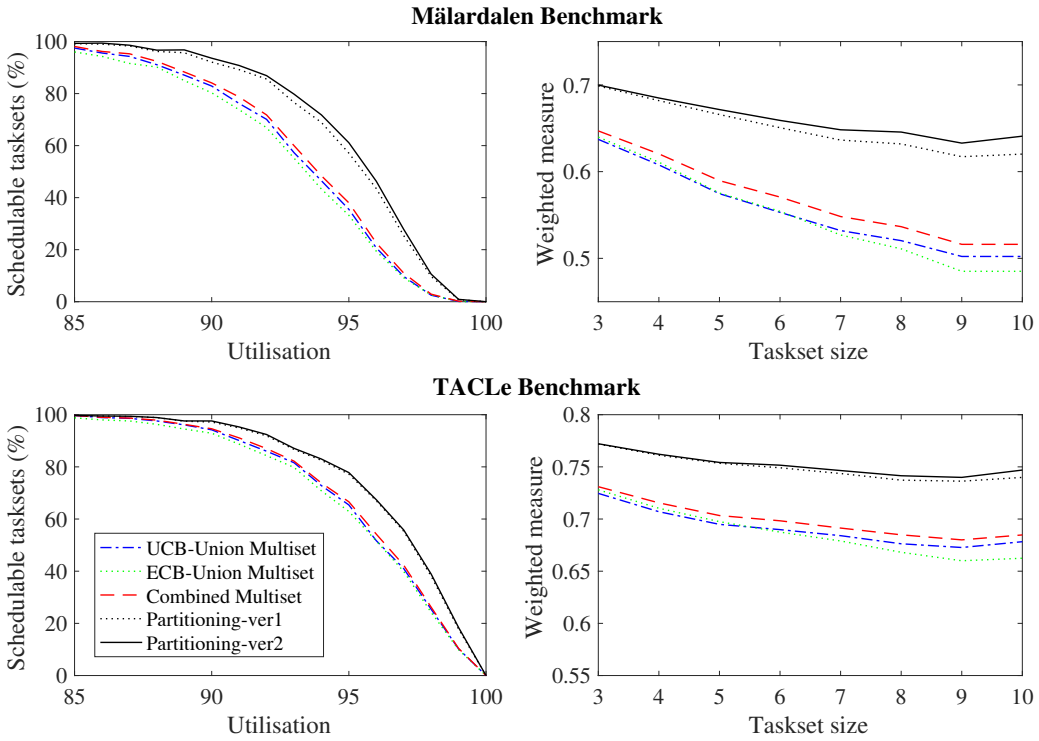


Figure 4 *Left*: Schedulability ratio at different taskset utilisation. *Right*: Weighted measure at different taskset size.

To increase the exhaustiveness of the performed evaluation and the respective results, for the rightmost plots from Figure 4 we used the weighted schedulability measure in order to show a 2-dimensional plot which would otherwise be a 3-dimensional plot, as proposed by Bastoni et al. [7]. In those figures, we show the weighted schedulability measure $W_y(|\Gamma|)$, for schedulability test y as a function of taskset size $|\Gamma|$. For each taskset size (in range from 3 to 10), this measure combines data for all of the tasksets generated for each utilisation level from 0.85 to 1, with step of 0.1, since for utilisation levels from 0 to 0.85 all of the compared methods deem almost all tasksets to be schedulable. For each taskset size $|\Gamma|$, the schedulability measure $W_y(|\Gamma|)$ is equal to $W_y(|\Gamma|) = \sum_{\forall\Gamma}(U_\Gamma \times B_y(\Gamma,|\Gamma|))/\sum_{\forall\Gamma} U_\Gamma$, where $B_y(\Gamma,|\Gamma|)$ is the binary result (1 if schedulable, 0 otherwise) of a schedulability test y for a taskset Γ and taskset size $|\Gamma|$. Weighting the schedulability results by taskset utilisation means that the method which succeeds to produce a higher weighted measure, compared to the others, is more prone to identify tasksets with higher utilisation as schedulable.

The results show that *Partitioning-ver2* is able to identify more schedulable tasksets compared to the others for any given taskset size, immediately followed by *Partitioning-ver1*. Also, as the taskset size increases, the multiset-based methods deteriorate more in identifying schedulable tasksets compared to the proposed methods. This means that partitioning-based methods are able to identify more tasksets as schedulable with an increase of the taskset size and utilisation.

Next, we report the worst-case computation time results since *Partitioning-ver2* uses set partitioning which is known to be a computation with quadratic/exponential complexity, depending on the algorithm type. The results, reported in the left-most plot in Figure 5,

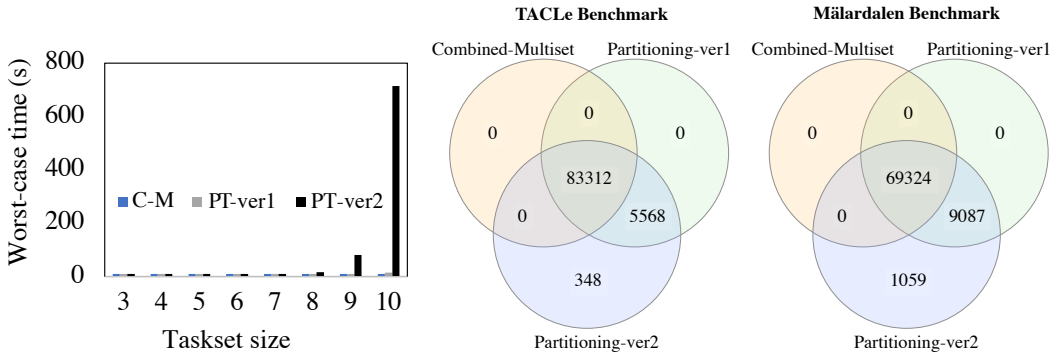


Figure 5 *Leftmost:* The worst-case measured analysis time per taskset, at different tasket size. *Center and rightmost:* Venn Diagrams[19] representing schedulability result relations between different methods, over 120000 analysed tasksets per each – TACLe Benchmark and Mälardalen Benchmark.

were computed on MacBook Pro (Retina, 13-inch, Early 2015) version, with Intel Core i5 processor of 2,9 GHz, and DDR3 RAM memory of 8 GB, and 1867 MHz. We used a sequential set partitioning algorithm, and as shown in the graph, in this case exponential complexity is evident for *Partitioning-ver2*. However, the proposed method is intended to be used offline, and its performance can be improved using the algorithm from [14], and even more with parallel computing, e.g. set partitioning algorithms proposed by Djokic et al. [13]. In contrast, *Partitioning-ver1* has a low worst-case time measured for each experiment for different taskset sizes, similar to the the *Combined-multiset* approach.

Finally, we show the relations between the results in Figure 5 (central and rightmost figures). In the central figure, it is evident that all tasksets from TACLe benchmark, that are identified as schedulable by *Combined-multiset*, are also identified as schedulable by *Partitioning-ver1* and *Partitioning-ver2*. However, partitioning-based approaches identify 5568 (and 9087) additional schedulable tasksets depending on the benchmark, while *Partitioning-ver2* identifies additional 348 (and 1059) schedulable tasksets compared to *Partitioning-ver1*. In conclusion of the evaluation, we notice that the proposed partitioning-based algorithms outperform existing state of the art *Combined-Multiset* approach. Also *Partitioning-ver2* outperforms *Partitioning-ver1*, however this comes with the expense of time complexity. The complexity of *Partitioning-ver2* can be further decreased by narrowing down the task interactions for which the preemption combinations should be generated. This remains as a part of the future work as well as the formal proof of the dominance relations between the methods. Finally, the proposed approaches allow for a hybrid, joint use of the two proposed algorithms, while *Partitioning-ver1* significantly outperforms the existing multiset approaches without the expense of time complexity.

7 Conclusions

In this paper, we proposed a partitioning based cache-aware schedulability analysis for precise and safe estimation of cache-related preemption delays in the context of fully-preemptive scheduling of real-time systems with sporadic tasks with fixed priorities. The proposed methods are based on a precise analysis of: 1) different preemption subgroups, and 2) different preemption combinations that may occur within a system, and therefore they are able to compute more precise cache-related preemption delay estimations compared to the state of the art approaches. The evaluation was performed using the realistic task parameters

from well-established benchmarks, obtained with a low-level analysis tool, and it showed that the proposed approaches manage to identify significantly more schedulable tasksets compared to the other preemption-cost aware approaches.

In future work, we will apply the proposed method in the context of limited preemptive scheduling since for such task model partitioning-based consideration of preemptions can lead to a more precise computation of cache-related preemption delay. We will also apply the proposed methods to other cache architectures and replacement protocols since many existing analyses inherit the overly pessimistic estimations which are identified in this paper. Finally, we will define a more precise static analysis on number of cache block reloads that are possible during the execution of a task since the existing useful cache block concept significantly over-approximates cache-block reloadability.

References

1 Martin Aigner. A characterization of the bell numbers. *Discrete mathematics*, 205(1-3):207–210, 1999.

2 Sebastian Altmeyer and Claire Burguiere. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *Real-Time Systems, 2009. ECRTS'09. 21st Euromicro Conference on*, pages 109–118. IEEE, 2009.

3 Sebastian Altmeyer, Robert I Davis, and Claire Maiza. Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 261–271. IEEE, 2011.

4 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, 2012.

5 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, 2016.

6 Sebastian Altmeyer, Claire Maiza, and Jan Reineke. Resilience analysis: tightening the CRPD bound for set-associative caches. In *ACM Sigplan Notices*, volume 45, pages 153–162. ACM, 2010.

7 Andrea Bastoni, Björn Brandenburg, and James Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. *Proceedings of OSPERT*, pages 33–44, 2010.

8 Enrico Bini and Giorgio C Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

9 Tobias Blaß, Sebastian Hahn, and Jan Reineke. Write-back caches in wcet analysis. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

10 Claire Burguière, Jan Reineke, and Sebastian Altmeyer. Cache-related preemption delay computation for set-associative caches-pitfalls and solutions. In *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

11 José V Busquets-Mataix, Juan José Serrano, Rafael Ors, Pedro Gil, and Andy Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Real-Time Technology and Applications Symposium, 1996. Proceedings., 1996 IEEE*, pages 204–212. IEEE, 1996.

12 Robert I Davis, Sebastian Altmeyer, and Jan Reineke. Response-time analysis for fixed-priority systems with a write-back cache. *Real-Time Systems*, 54(4):912–963, 2018.

13 Borivoje Djokić, Masahiro Miyakawa, Satoshi Sekiguchi, Ichiro Semba, and Ivan Stojmenović. Parallel algorithms for generating subsets and set partitions. In *International Symposium on Algorithms*, pages 76–85. Springer, 1990.

14 MC Er. A fast algorithm for generating set partitions. *The Computer Journal*, 31(3):283–284, 1988.

15 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. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2016.

16 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen wcet benchmarks: Past, present and future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010 .

17 Sebastian Hahn, Michael Jacobs, and Jan Reineke. Enabling compositionality for multicore timing analysis. In *Proceedings of the 24th international conference on real-time networks and systems*, pages 299–308. ACM, 2016.

18 Paul R Halmos. *Naive set theory*. Courier Dover Publications, 2017.

19 Henry Heberle, Gabriela Vaz Meirelles, Felipe R da Silva, Guilherme P Telles, and Rosane Minghim. Interactivenn: a web-based tool for the analysis of sets through venn diagrams. *BMC bioinformatics*, 16(1):169, 2015.

20 Chang-Gun Lee, Joosun Han, Yang-Min Seo, Sang Luyl Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sam Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

21 Rodolfo Pellizzoni, Bach D Bui, Marco Caccamo, and Lui Sha. Coscheduling of CPU and I/O transactions in COTS-based embedded systems. In *Real-Time Systems Symposium, 2008*, pages 221–231. IEEE, 2008.

22 Harini Ramaprasad and Frank Mueller. Bounding worst-case response time for tasks with non-preemptive regions. In *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 58–67. IEEE, 2008.

23 Harini Ramaprasad and Frank Mueller. Tightening the bounds on feasible preemptions. *ACM Transactions on Embedded Computing Systems (TECS)*, 10(2):27, 2010.

24 Syed Aftab Rashid, Geoffrey Nelissen, Sebastian Altmeyer, Robert I Davis, and Eduardo Tovar. Integrated analysis of cache related preemption delays and cache persistence reload overheads. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 188–198. IEEE, 2017.

25 Syed Aftab Rashid, Geoffrey Nelissen, Damien Hardy, Benny Akesson, Isabelle Puaut, and Eduardo Tovar. Cache-persistence-aware response-time analysis for fixed-priority preemptive systems. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 262–272. IEEE, 2016.

26 Altmeyer Sebastian, Douma Roeland, Lunniss Will, and I Davis Robert. Evaluation of cache partitioning for hard real-time systems. In *proceedings Euromicro Conference on Real-Time Systems (ECRTS)*, pages 15–26, 2014.

27 Darshit Shah, Sebastian Hahn, and Jan Reineke. Experimental evaluation of cache-related preemption delay aware timing analysis. In *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

28 Jan Staschulat, Simon Schliecker, and Rolf Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Real-Time Systems, 2005.(ECRTS 2005). Proceedings. 17th Euromicro Conference on*, pages 41–48. IEEE, 2005.

29 Gregory Stock, Sebastian Hahn, and Jan Reineke. Cache persistence analysis: Finally exact. In *Real-Time Systems Symposium (RTSS)*, December 2019.

30 Yudong Tan and Vincent Mooney. Timing analysis for preemptive multitasking real-time systems with caches. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(1):7, 2007.

31 Hiroyuki Tomiyama and Nikil 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*, pages 67–71. ACM, 2000.

Nested, but Separate: Isolating Unrelated Critical Sections in Real-Time Nested Locking

James Robb
Reykjavik University, Iceland
me@jamesrobb.ca

Björn B. Brandenburg
Max Planck Institute for Software Systems, Kaiserslautern, Germany
bbb@mpi-sws.org

Abstract

Prior work has produced multiprocessor real-time locking protocols that ensure asymptotically optimal bounds on priority inversion, that support fine-grained nesting of critical sections, *or* that are independence-preserving under clustered scheduling. However, while several protocols manage to come with two out of these three desirable features, no protocol to date accomplishes all three. Motivated by this gap in capabilities, this paper introduces the *Group Independence-Preserving Protocol* (GIPP), the first protocol to support fine-grained nested locking, guarantee a notion of independence preservation for fine-grained nested locking, *and* ensure asymptotically optimal priority-inversion bounds. As a stepping stone, this paper further presents the *Clustered k-Exclusion Independence-Preserving Protocol* (CKIP), the first asymptotically optimal independence-preserving k-exclusion lock for clustered scheduling. The GIPP and the CKIP rely on allocation inheritance (a.k.a. migratory priority inheritance) as a key mechanism to accomplish independence preservation.

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

Keywords and phrases multiprocessor real-time locking, nested locking, independence preservation, suspension-oblivious analysis, priority inversion, asymptotically optimal blocking, RNLP, OMIP

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.6

Related Version Extended paper with full results `https://www.mpi-sws.org/tr/2020-002.pdf`.

Funding This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 803111).

1 Introduction

From a practical point of view, any effective multiprocessor real-time locking protocol should avoid some obvious pitfalls:

1. Non-conflicting accesses to different resources should *not* be needlessly serialized.
2. Tasks should *not* be delayed due to contention for resources they do not even access.
3. A real-time locking protocol should *not* make it impossible to provision latency-sensitive tasks carefully designed to not require any shared resources (such as critical interrupt handlers with stringent sub-millisecond deadlines).
4. Worst-case blocking should *not* be exponential.

It is not difficult to see how a protocol that fails to meet these requirements would result in costly and inefficient over-provisioning. It may thus come as a surprise that *no multiprocessor real-time locking protocol in the published literature satisfies all four properties!*

The reason, however, is all the more understandable: these innocuous-looking requirements translate to well-known real-time locking protocol properties that are difficult to ensure by themselves, let alone *jointly* in a single protocol. In particular, Requirement 3 rules out any locking protocol that relies on the non-preemptive execution of critical sections, a

trait of virtually all spin-lock protocols [8]. Requirement 1 implies that a protocol must support *fine-grained nested locking* [3, 31, 32] – that is, tasks must be able to incrementally lock additional resources while already holding some other shared resources – because the alternative, namely coarse-grained *group locking* [4], serializes even trivially non-conflicting requests for resources in the same group. Fine-grained nested real-time locking, however, is a notoriously difficult problem [3, 8, 31], and easily gives rise to blocking bounds that are exponential in the number of simultaneously acquired resources [8, 19, 31]. In fact, it is a fundamental algorithmic challenge to ensure both Requirements 1 and Requirements 4 in a single protocol. The only known protocol to surmount this challenge is Ward and Anderson's *Real-Time Nested Locking Protocol* (RNLP) [32, 33], and actually does so with *asymptotically optimal* bounds on *priority inversion blocking* [10, 32].

The RNLP, in turn, does not satisfy Requirement 2. As we discuss in more detail in Section 2, the RNLP relies on a *token lock* that regulates contention for shared resources, an ingenious element of the RNLP's design that ensures its asymptotic optimality. However, in its configuration for suspension-based locking (under "suspension-oblivious analysis," see Section 2), this token lock becomes a global bottleneck that causes tasks to delay each other even if they do not share any resources.

To satisfy Requirements 2 and 3, a locking protocol must temporally isolate tasks from each other when they do not access the same resources, which is known as *independence preservation* [6], a concept we discuss in detail in Section 2. The only protocol to date to realize independence preservation for clustered scheduling is the $\mathcal{O}(m)$ *Independence-Preserving Protocol* (OMIP) [6]. However, the OMIP as originally proposed fails to satisfy Requirement 1 since it can realize nested locking only through group locks – and if the OMIP is extended to permit fine-grained locking, it fails to satisfy Requirement 4 due to its FIFO queuing structure, which gives rise to exponential worst-case blocking [31].

Seemingly, the satisfaction of one of the four requirements comes at the cost of another. Is this a fundamental limitation? Is it perhaps *impossible* to satisfy all four requirements at once? As we show in this paper, the answer to both questions is *no* – it is in fact possible to combine fine-grained nesting, independence preservation, and asymptotically optimal pi-blocking in a single protocol, which we demonstrate by constructing the first such protocol.

In **related work**, the *Priority Inheritance Protocol* (PIP) [16, 28, 30] provides independence preservation, but only on uniprocessors or globally-scheduled systems, and the multiprocessor variant [16, 37] does not support nested critical sections. The *Flexible Multiprocessor Locking Protocol* (FMLP) [4] likewise is independence-preserving only under global scheduling, and only supports group locks [4, 37]. The *Multiprocessor Bandwidth Inheritance Protocol* (MBWI) [18, 19] and the *Multiprocessor Resource Sharing Protocol* (MrsP) [14] both allow for fine-grained nested locking. Unfortunately, they are subject to the exponential blow-up in blocking times described by Takada and Sakamura [31]. Several variants of the RNLP [32, 33] have been introduced in recent years to enable reader-writer synchronization [34], to provide contention-sensitive pi-blocking bounds [23], and to reduce implementation overheads in the locking protocol itself by means of a fast path [26] and lock servers [25]. However, none of these variants removes the *algorithmic* bottleneck of a single, shared token lock. For further discussion of the larger area of multiprocessor real-time locking protocols, we refer the interested reader to a recent comprehensive survey [8].

The **contributions** of this paper are as follows. First, we examine what it means to be independence-preserving in the presence of nested locking (Section 3), and the ensuing implications on asymptotic pi-blocking bounds (Section 3.1). Our main contribution is the *Group Independence-Preserving Protocol* (GIPP), the first asymptotically optimal, independence-

preserving, real-time fine-grained nested locking protocol for clustered scheduling under suspension-oblivious analysis (Section 4). In other words, the GIPP is the first multiprocessor real-time locking protocol that meets all of the desirable Requirements 1–4. To realize the GIPP, we develop and analyze a novel *Clustered k-Exclusion Independence-Preserving Protocol* (CKIP), an asymptotically optimal independence-preserving k-exclusion lock for clustered scheduling (Section 4.1). Lastly, we provide a fine-grained pi-blocking analysis of the GIPP using a state-of-the-art blocking analysis method based on linear programming (Section 5), and present an empirical evaluation that shows the GIPP to perform favorably in comparison to both the OMIP and the RNLP across a wide range of workloads (Section 6).

2 Background and Definitions

We assume the sporadic task model with n tasks $\tau = \{T_1, \ldots, T_n\}$ scheduled on m identical processors. Tasks are executed as a series of *jobs*, and we use J_i to denote a job of T_i. Each T_i is characterized by a *worst-case execution time* (WCET) e_i, a *period* p_i (i.e., the minimum arrival separation between jobs), and a *relative deadline* d_i. We assume implicit deadlines in this work, i.e., $d_i = p_i$, but the derived results do not depend on this constraint.

A job is said to be *pending* from the time it arrives until the time it completes. While a job is pending, it can be in one of two states: a *ready* job can be scheduled on a processor, whereas a *suspended* job cannot be scheduled. We assume that jobs do not self-suspend, and that all suspensions are due to interactions with the locking protocol(s) of the system.

Shared Resources. Tasks compete for a set of q serially-reusable shared resources $\Gamma = \{\ell_1, \ldots, \ell_q\}$. Each task T_i accesses a possibly empty subset $\gamma_i \subseteq \Gamma$ of the shared resources in the system. A *locking protocol* arbitrates requests from tasks for the shared resources in Γ such that no shared resource is held at the same time by two different tasks.

We say J_i *requires* a shared resource ℓ_a at the first instant access to ℓ_a is required for the continued execution of J_i. Once J_i requires ℓ_a, it *issues* a request $\mathcal{R}$ to the locking protocol to *acquire* ℓ_a. $\mathcal{R}$ is said to be *unsatisfied* from the time it is issued until J_i acquires ℓ_a, at which point $\mathcal{R}$ is said to be *satisfied*. $\mathcal{R}$ becomes *complete* when J_i *releases* ℓ_a, which is also when J_i no longer requires ℓ_a. Conversely, we say that $\mathcal{R}$ is *incomplete* from the time it is issued until it is completed. While J_i waits to acquire ℓ_a, it is said to make *progress* if (one of) the job(s) that prevent(s) J_i from acquiring ℓ_a is scheduled. Any method employed by a locking protocol to ensure that a job makes progress is called a *progress mechanism*.

If a J_i issues a request $\mathcal{R}$ for a shared resource ℓ_a while holding no other shared resources, then $\mathcal{R}$ is said to be an *outermost request*. Conversely, if J_i issues $\mathcal{R}$ for a shared resource ℓ_b while holding ℓ_a, then $\mathcal{R}$ is said to be a *nested request*. We do not require that requests are properly nested; it is possible for J_i to acquire ℓ_a, and then ℓ_b via a nested request, but release ℓ_a before releasing ℓ_b, as seen in Figure 1.

We construct a strict (irreflexive) partial ordering $\succ$ on Γ from the behavior of the tasks in τ: let $\ell_a \succ \ell_b$ iff there exists a task that requests ℓ_b while holding ℓ_a. It follows that we do not permit workloads where $\ell_a \succ \ell_b$ and $\ell_b \succ \ell_a$ both hold (which precludes deadlock).

Consider Figure 1. Let t_1 be the time that J_i issues an outermost request for a shared resource ℓ_a, and t_4 be the next point in time where both the outermost request is complete, and J_i holds no other shared resources. We call the part of J_i's execution in the time interval $[t_1, t_4)$ an *outermost critical section*. We define the *critical section length* to be the time required to execute a critical section in the absence of any blocking, and use $L_{i,a}$ to denote the length of J_i's longest outermost critical section that begins with an outermost request for ℓ_a. Note

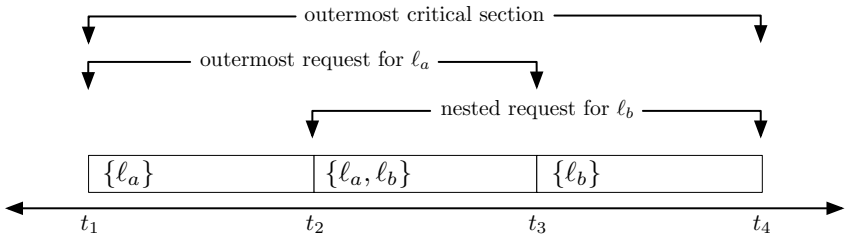


Figure 1 An outermost critical section spanning from time t_1 until time t_4. It begins with an outermost request for ℓ_a at time t_1, which completes at time t_3. The nested request for ℓ_b begins at time t_2 and completes at time t_4.

that $L_{i,a} \leq t_4 - t_1$, as J_i can experience scheduling or blocking delays during the execution of its outermost critical section. The maximal critical section length of J_i is $L_i^{\max} = \max_a L_{i,a}$ and the maximal critical section length among all tasks is $L^{\max} = \max_i L_i^{\max}$.

Scheduling. We target *clustered scheduling* in this work. Under clustered scheduling, the m processors in the system are grouped into disjoint subsets of size c called *clusters*. For simplicity, we assume $m = k \cdot c$ where $k \in \mathbb{Z}^+$. Each task is statically assigned a *home cluster*, denoted by $C(T_i)$, which it will be scheduled in. The tasks in each cluster are scheduled by a global scheduling algorithm applied to the processors in that cluster. Global and partitioned scheduling are special cases of clustered scheduling where $c = m$ and $c = 1$, respectively.

We assume the use of *Job-Level Fixed Priority* (JLFP) scheduling algorithms such as *Earliest-Deadline First* (EDF) scheduling or *Fixed-Priority* (FP) Scheduling. The priorities assigned to jobs are assumed to be *unique* in each cluster, with any ties broken in favor of lower-indexed jobs. A job J_i has both an *effective priority* and a *base priority*. J_i's base priority, which is determined by the scheduling algorithm, never changes, but its effective priority may change due to interactions with a locking protocol; the scheduler uses J_i's current effective priority when scheduling jobs. We let $\mathrm{H}(J_i, t)$ be a predicate that indicates whether J_i is among the c highest effective-priority pending jobs at time t in its home cluster.

Priority Inversion. Intuitively, a *priority inversion* is said to occur when the execution of a higher-priority job is delayed due to the execution of a lower-priority job [28, 30]. A typical example of this occurs when a lower-priority job holds a shared resource that a higher-priority job is requesting, and so the higher-priority job's execution is delayed until the resource is released. We refer to this type of blocking as *priority inversion blocking* (pi-blocking).

Under *suspension-oblivious analysis* (s-oblivious analysis) it is assumed that tasks never self-suspend (even though they may), and any self-suspension is treated as execution time. Conversely, *suspension-aware analysis* (s-aware analysis) explicitly accounts for self-suspensions. These two methods of analysis yield different lower-bounds on the pi-blocking incurred by a job due to requests for shared resources, which are $\Omega(m)$ and $\Omega(n)$, respectively [10]. Locking protocols that ensure $\mathcal{O}(m)$ and $\mathcal{O}(n)$ per-job pi-blocking under s-oblivious analysis and s-aware analysis, respectively, are said to be *asymptotically optimal* [10].

We focus on s-oblivious analysis in this work, and adapt the definition for s-oblivious priority inversion under clustered scheduling from previous work [6].

▶ **Definition 1.** J_i incurs an *s-oblivious priority inversion* at time t iff J_i is not scheduled and its priority is among the top c priorities of pending jobs in cluster $C(T_i)$, i.e., if $\mathrm{H}(J_i, t)$.

Let b_i denote the maximum amount of s-oblivious pi-blocking that any job of T_i incurs. When deriving asymptotic bounds on b_i, we consider $L^{\max}$ to be a constant (i.e., not a function of m nor n). Following prior work [8, 10, 27, 28, 30], we consider pi-blocking to be *bounded* only if no b_i depends on any e_i (the e_i parameter is *not* considered to be a constant).

Independence Preservation. The high-level idea of independence preservation is that tasks are isolated from "unrelated" critical sections. This can be easily pictured for locking protocols that do not permit nested locking: if a task never requests a shared resource ℓ_a, then it incurs no pi-blocking as a result of requests by other tasks for ℓ_a. This is of particular importance when considering latency-sensitive tasks, defined as follows.

▶ **Definition 2.** A task T_i is said to be *latency-sensitive* if its *slack*, the difference between its relative deadline and WCET, is less than the length of the longest critical section of some other task, i.e., $d_i - e_i < L^{\max}$.

If critical sections are permitted to execute non-preemptively, then a job of a latency-sensitive task necessarily misses its deadline if its release coincides with a lower-priority task executing non-preemptively for $L^{\max}$ time units, thus providing clear motivation for independence preservation. Prior work introduced the notion of independence preservation [6] among tasks under the assumption that nested resource requests are never made. For clarity, and to build upon it later, we restate the definition here.

▶ **Definition 3.** Let $b_{i,a}$ denote the maximum pi-blocking incurred by J_i due to requests for a shared resource ℓ_a, and $N_{i,a}$ be the number of times J_i requests ℓ_a. Under s-oblivious analysis, a locking protocol is *non-nested independence-preserving* iff $N_{i,a} = 0$ implies $b_{i,a} = 0$.

Priority Donation. The progress mechanism *Replica-Request Priority Donation* (RRPD) [35] was introduced to realize the *Replica-Request Priority Donation Global Locking Protocol* (R^2DGLP) [35], a real-time k-exclusion lock for globally-scheduled systems. RRPD is a modification of the earlier *Job-Release Priority Donation* (JRPD) progress mechanism [11]; RRPD has a job donate its priority upon requesting a resource, whereas donation happens upon arrival (i.e., release) under JRPD. Unlike JRPD, which was designed for clustered systems, RRPD relies on the ability to compare priorities among all jobs. Thus, RRPD applies only to globally-scheduled systems, as analytically speaking, numeric priority values are incomparable across clusters. This trade-off allows the R^2DGLP to realize non-nested independence preservation as jobs only donate their priority upon resource request. We revisit RRPD further in Section 4, and the R^2DGLP in Section 4.1.

Nested Locking Protocols. The way locking protocols support nesting can be divided into two broad categories: *coarse-grained* locking and *fine-grained* locking. The FMLP [4] is an example of a real-time locking protocol that employs coarse-grained nested locking. Under the FMLP, resources are split into groups, and each group has a corresponding *group lock*. A job that holds a group lock has mutually-exclusive access to all the resources in the corresponding group; while simple, the loss in parallelism is clear.

In contrast to coarse-grained locking, fine-grained locking allows shared resources to be acquired incrementally. For example, the RNLP [32] allows jobs to issue nested requests for resources as they are needed, which provides more opportunities for parallelism when compared to simple group locks. However, the increased potential for parallelism comes at the cost of more complicated protocol rules and data structures when compared to group locks, as group locks can be realized with simpler non-nested locking protocols.

The RNLP [32] was a breakthrough in real-time nested locking, as it is the first, and to date only, asymptotically optimal fine-grained nested locking protocol for multiprocessor systems. It can be applied under clustered (and therefore global/partitioned) scheduling. The RNLP is in fact a "meta protocol" in the sense that it defines the properties that a *token lock* and a *request satisfaction mechanism* (RSM) must obey to realize an optimal nested locking protocol. The token lock restricts the number of jobs that can hold resources at any given time, and the RSM sequences requests of the token holders and ensures progress. The behavior of a particular *instantiation* (i.e., a token lock/RSM combination) of the RNLP is largely determined by the progress mechanisms that token lock and RSM employ. Ward and Anderson demonstrate a number of possible instantiations of the RNLP [32]. When instantiated for s-oblivious analysis, the token lock is configured to provide m tokens. We examine the rules, requirements, and structure of the RNLP further in Section 4.

Summary. If shared resources that will be held at the same time are protected by a group lock, then both the OMIP and R^{2}DGLP can realize asymptotically optimal coarse-grained nested locking under clustered and global scheduling, respectively, while still remaining non-nested independence-preserving.

Non-nested independence preservation is not trivially realized with the RNLP in the absence of nested locking. Fundamentally, the use of a token lock that arbitrates access to m tokens (and thus restricts the number of resource-holding jobs to m) precludes non-nested independence preservation; when all tokens in the system are held, a job must wait to acquire a token, even if the resource it requires is never accessed by any other task. In fact, to the best of our knowledge, prior work has not yet considered what it means to be independence-preserving in the context of fine-grained nested locking.

3 Nested Independence Preservation

The notion of independence preservation introduced in Definition 3 does not directly apply to nested locking, and there exists more than one way to generalize the notion in a conceptually analogous way, depending on when exactly resources involved in nesting are considered to be "related" (i.e., when they are considered "non-independent"). We consider two possible definitions in the following that we consider to be the most natural ways of expressing the idea.

3.1 Outer-Lock Independence Preservation

The core idea behind *outer-lock independence preservation* is that there is an asymmetric, transitive, and reflexive relation, which we call *dependence*, between a shared resource ℓ_a, and shared resources acquired via nested requests (with respect to an outer request for ℓ_a).

More precisely, for a shared resource $\ell_a \in \Gamma$, we say it *depends* on the set $[\ell_a]^{ol} = \{\ell_x \mid \ell_a \succ \ell_x\} \cup \{\ell_a\}$. Similarly, a task T_i depends on the set of shared resources $D_i^{ol} = \bigcup_{\ell_a \in \gamma_i} [\ell_a]^{ol}$. Based on this precise notion of dependence, we define outer-lock independence preservation.

▶ **Definition 4.** Let $b_{i,a}$ denote the maximum pi-blocking incurred by J_i due to requests by any task for a shared resource ℓ_a. Under s-oblivious analysis, a locking protocol is *outer-lock independence-preserving* iff $\ell_a \notin D_i^{ol}$ implies $b_{i,a} = 0$.

Outer-lock independence preservation as a notion for nested independence preservation has a fundamental impact on the pi-blocking incurred by a job under s-oblivious analysis. In fact, it turns out that for a large class of locking protocols (that arguably includes all "reasonable" locking protocols), the per-request pi-blocking bound is necessarily non-optimal

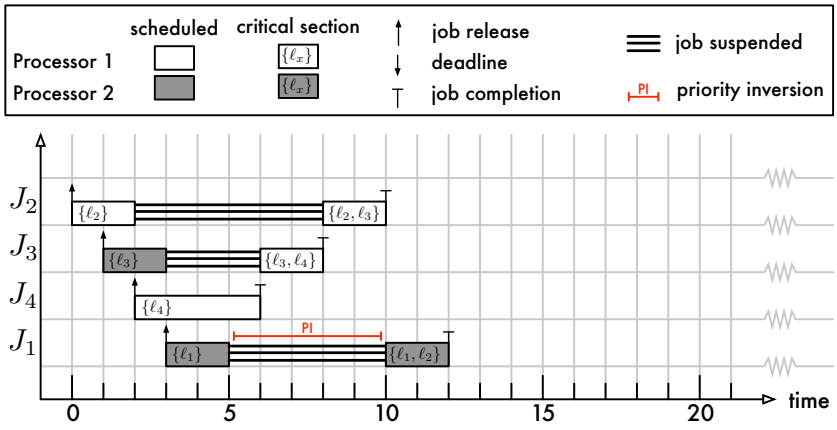


Figure 2 One possible G-FP schedule of $\tau^{ol}(4)$ on $m = 2$ processors. The jobs are in ascending priority from top to bottom (i.e., J_2 has the lowest priority, and J_1 has the highest priority).

under *rate monotonic* (RM), *deadline monotonic* (DM), or EDF scheduling (and s-oblivious analysis). Our proof of the non-optimality of outer-lock independence-preserving locking protocols requires the following seemingly obvious property.

▶ **Definition 5.** Let $\Gamma' \subseteq \Gamma$ denote the set of shared resources currently held by all tasks in the system. A locking protocol is *non-procrastinating* if any request for a shared resource (by one of the c highest-priority pending jobs in each cluster) is satisfied immediately if $|\Gamma'| = 0$.

We are not aware of *any* real-time locking protocol in the literature that does not satisfy a request $\mathcal{R}$ for a shared resource by one of the c highest-priority pending jobs when $|\Gamma'| = 0$. If we assume non-clairvoyance, and that tasks are sporadic (i.e., we cannot predict future job arrivals), then delaying the satisfaction of $\mathcal{R}$ is tantamount to willingly wasting CPU time; this is in direct contradiction to one of the most important goals of an effective real-time locking protocol. Non-procrastination is also a fairly weak property as it does not impose restrictions on how to arbitrate access to resources once contention is present.

We use a parameterized task set to lower-bound pi-blocking under RM, DM, and EDF.

▶ **Definition 6.** Let $\tau^{ol}(n) = \{T_1, \ldots, T_n\}$ be a task set of n tasks that share n resources $\{\ell_1, \ldots, \ell_n\}$, where $n \geq m \geq 2$, with the following properties: **(i)** $\ell_1 \succ \ell_2 \succ \ldots \succ \ell_{n-1} \succ \ell_n$; **(ii)** $\forall_{1 \leq i \leq n}\ e_i = 4$; **(iii)** $\forall_{1 \leq i \leq n}\ p_i = d_i = e_i \cdot n \cdot i$; **(iv)** $\forall_{1 \leq i < n}$ jobs of T_i require $\{\ell_i\}$ during the first two units of their execution, and then $\{\ell_i, \ell_{i+1}\}$ during the last two units of their execution; **(v)** jobs of T_n require $\{\ell_n\}$ throughout the four units of their execution.

▶ **Theorem 7.** *There exists an arrival sequence of $\tau^{ol}(n)$ such that $\max_{T_i \in \tau^{ol}(n)} b_i = \Omega(n)$ under s-oblivious analysis for any suspension-based incremental locking protocol that is non-procrastinating and outer-lock independence-preserving, when scheduled under RM, DM, or EDF scheduling (with respect to each cluster).*

Proof. Let $a_{i,1}$ denote the first arrival of T_i. Consider the arrival sequence of $\tau^{ol}(n)$ where $a_{i,1} = i - 2$ for $2 \leq i \leq n$ and $a_{1,1} = n - 1$. An example of $\tau^{ol}(4)$ with this arrival sequence is depicted in Figure 2. At time $t = 0$, J_2 requests and acquires ℓ_2, as we assume the use of a non-procrastinating locking protocol. At time $t = 1$, a request for ℓ_3 is made by J_3. If J_3 does not acquire ℓ_3 (and is therefore not scheduled) at $t = 1$, then the blocking that results from delaying J_3's request would result in a violation of outer-lock independence preservation

as we would then have $b_{3,2} > 0$ despite $\ell_2 \notin D_3^{ol}$. The same argument analogously applies to all jobs released up until $t = n - 1$ when J_1 arrives and issues a request $\mathcal{R}_1$ for ℓ_1. There are then two cases to consider: $\mathcal{R}_1$ is satisfied immediately, or it is satisfied at a later time.

In the first case $\mathcal{R}_1$ is satisfied immediately and J_1 issues a nested request $\mathcal{R}_2$ for ℓ_2 at time $t = n + 1$. The maximum number of units of execution completed for jobs $J_2, \ldots, J_n$ up to $t = n+1$ is $(n-1)\cdot 2+1 = 2n-1$ for any $m \geq 2$. This is because jobs $J_2, \ldots, J_{n-1}$ can execute for at most 2 units of time before suspending due to a nested request for an already held resource, and because J_n can execute for at most 3 units of time until $\mathcal{R}_2$ is issued. Therefore, at the time $\mathcal{R}_2$ is issued, there are $(\sum_{i=2}^{n} e_i) - (2n-1) = (n-1)\cdot 4 - (2n-1) = 2n-3$ units of execution left before jobs $J_2, \ldots, J_n$ complete and ℓ_2 becomes available for acquisition by J_1. Furthermore, as jobs $J_2, \ldots, J_{n-1}$ are all waiting for the job with the next-highest index to release a resource, their executions are serialized. Thus J_1 incurs at the very least $2n - 3 = \Omega(n)$ time units of s-oblivious pi-blocking while waiting to acquire ℓ_2.

In the second case, we assume $\mathcal{R}_1$ is satisfied at time $t = n - 1 + \epsilon$ where $\epsilon > 0$ (i.e., not immediately). Then, J_1 would begin to incur s-oblivious pi-blocking at $t = n - 1$, as it is the highest-priority job and not scheduled. This situation cannot result in a reduction of the s-oblivious pi-blocking that J_1 incurs in the first case because **(i)** J_1 is the highest-priority job by construction, and **(ii)** that the serialization of executions described in the first case enforces a minimum of $2n - 3$ units of execution before ℓ_2 is released. Therefore, the asymptotic lower-bound in the first case applies, as J_i still incurs a minimum of $2n - 3 = \Omega(n)$ units of s-oblivious pi-blocking while waiting to acquire ℓ_2. ◀

To conclude, under a standard set of assumptions and commonly used scheduling algorithms, any outer-lock independence-preserving protocol is necessarily non-optimal with respect to s-oblivious pi-blocking. In the rest of this paper, we focus on an alternative definition for nested independence preservation, which we call group independence preservation.

3.2 Group Independence Preservation

With group independence preservation, the relationships that exist among shared resources and tasks are defined by relaxing when resources are considered to be non-independent.

Let $\circ$ be a symmetric relation on the set of shared resources Γ. For any $\ell_a \in \Gamma$, let $\ell_a \circ \ell_a$, and for any $\ell_b, \ell_c \in \Gamma$ let $\ell_b \circ \ell_c$ if $\ell_b \succ \ell_c$ or $\ell_c \succ \ell_b$. The transitive closure of $\circ$ forms an equivalence relation on the resources in Γ, which we denote with $\sim$. Then the equivalence class $g(\ell_a) = \{\ell_x \in \Gamma \mid \ell_a \sim \ell_x\}$ is the set of resources that ℓ_a is *associated* with.

We refer to these equivalence classes as *groups*, and let $G = \{g_1, \ldots, g_r\}$ be the set of resource groups in the system. From the definition of a group, it naturally follows that the groups in G are disjoint, and that their union yields Γ. This definition of groups matches the notion of resource groups used in the FMLP [4]. Based on this notion of resource groups, we say that a task T_i is *associated* with the shared resources $D_i = \bigcup_{\ell_a \in \gamma_i} g(\ell_a)$.

▶ **Definition 8.** Let $b_{i,a}$ denote the pi-blocking incurred by J_i due to requests by any task for a shared resource ℓ_a. Under s-oblivious analysis, a locking protocol is *group independence-preserving* iff $\ell_a \notin D_i$ implies $b_{i,a} = 0$.

Stated differently, group independence is preserved if the overall s-oblivious pi-blocking $b_i = \sum_a b_{i,a}$ of each task does not depend on resources that the task is not associated with.

4 The Group Independence-Preserving Protocol

We show that group independence-preserving protocols do not necessarily suffer from the $\Omega(n)$ s-oblivious pi-blocking bound seen with outer-lock independence preservation. We

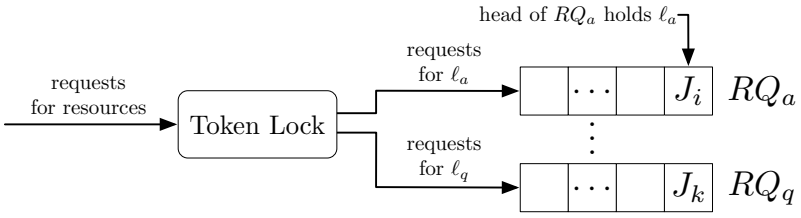


Figure 3 Under the RNLP, a job J_i that requests a resource ℓ_a first competes for a token. Once J_i acquires a token, a timestamp $ts(J_i)$ is recorded, and the request is enqueued into the priority queue RQ_a ordered by the token-acquisition timestamps. The job of the request that occupies the head of a queue holds the corresponding shared resource.

demonstrate this through the construction of group independence-preserving fine-grained nested locking protocol that is asymptotically optimal under s-oblivious analysis: the *Group Independence-Preserving Protocol* (GIPP). We review the necessary background of the RNLP and RRPD in this section required to realize the GIPP, and then give a high-level overview of the GIPP before constructing its components in subsequent sections.

RNLP. As a brief reminder, an instantiation of the RNLP consists of a token lock and an RSM. We speak in the context of a job J_i that requires a shared resource $\ell_a \in \Gamma$ when reviewing the following rules. Under the rules of the RNLP [32, Section 3], J_i first competes for a token λ in the token lock before it can actually issue a request for ℓ_a. Once J_i acquires λ, a timestamp $ts(J_i)$[1] of the current time is recorded. Conceptually, J_i now enters the RSM, and is placed in a priority queue RQ_a ordered by increasing timestamp[2]; such a priority queue exists for each shared resource in Γ. J_i waits until it becomes the head of RQ_a, and then it holds ℓ_a unless a job J_k holds a shared resource ℓ_b s.t. $\ell_b \succ \ell_a \wedge ts(J_k) < ts(J_i)$. Once J_i completes its request for ℓ_a it is dequeued from RQ_a and the new head of RQ_a (if any) acquires ℓ_a subject to the previous constraints. As requests need not be properly nested, J_i may continue to compete for shared resources in the RSM. Once J_i has completed its outermost critical section, it releases λ. This queuing structure is depicted in Figure 3.

The RNLP specifies properties [32] that a token lock and RSM must have to realize a valid instantiation of the RNLP. Of the following properties, **T1** and **T2** apply to the token lock, and **R1** applies to the RSM.

T1 There are at most c token-holding jobs per cluster at any time (and thus m system-wide).

T2 If a job is pi-blocked waiting for a token, then it makes progress (i.e., the token holder is scheduled).

R1 If a job is pi-blocked by the RSM, then the job makes progress.

Finally, we restate one of the RNLP's theorems, as we use it to prove properties of the GIPP later in this work. We refer to the original RNLP paper for its proof [32].

▶ **Theorem 9** ([32, Theorem 1], paraphrased). *The maximum amount of pi-blocking in the RSM, when waiting is realized by suspending, is* $(m-1) \cdot L^{max}$.

[1] The timestamps are assumed to be unique, and thus there is a total order on them.
[2] For clarity, the head of a queue RQ_a is the request with the oldest (i.e., earliest) timestamp.

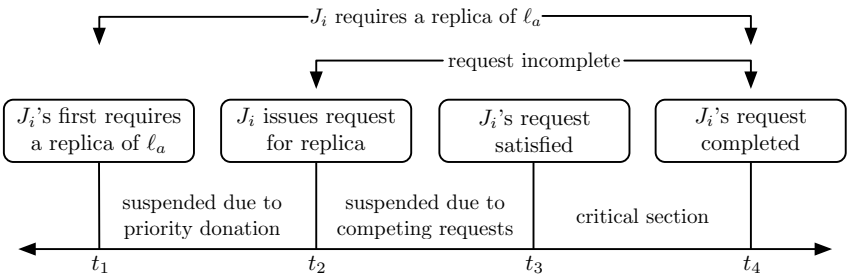


Figure 4 Life-cycle of a request under RRPD for a replica of a shared resource ℓ_a (adapted from [35]).

RRPD. Under the rules of RRPD [35, Section 3], a job donates its priority upon requesting a resource, and as such only becomes a priority donor at most once per outermost critical section. Thus if a job does not request a resource, it never donates its priority to jobs requesting said resource. We speak in the context of a job J_i that requires a shared resource $\ell_a \in \Gamma$ when reviewing the following rules. Furthermore, we discuss RRPD in the context of a single cluster despite it having been designed for globally-scheduled systems, as we assume throughout this paper that clusters function independently with respect to RRPD. We refer the reader to Figure 4 for a visual depiction of the life-cycle of an RRPD request, and the terminology used. Let $R(\ell_a, t)$ denote the (possibly empty) set of the c highest effective-priority jobs that require ℓ_a at time t. If J_i requires ℓ_a and $J_i \in R(\ell_a, t)$ then J_i may issue a request for ℓ_a, unless it becomes a priority donor first. J_i becomes the priority donor of a job J_d at time t_1 (see Figure 4) if **(i)** $J_i \in R(\ell_a, t)$, **(ii)** there are c jobs with an incomplete request for ℓ_a, and **(iii)** J_d is the lowest-effective priority job with an incomplete request for ℓ_a; while J_i donates its priority to J_d in the interval $[t_2, t_4)$ it is suspended and assumed to have no effective-priority. Should J_i be displaced from $R(\ell_a, t)$ while donating its priority to J_d by a job J_k, then J_i ceases to be a priority donor and J_k becomes J_d's priority donor. Finally, if J_i is J_d's priority donor when J_d completes its outermost critical section, then J_i ceases to be a priority donor.

We now restate two of the RRPD's lemmas as we use them later when constructing the components required to realize the GIPP. Their proofs are available in the original work [35].

▶ **Lemma 10** ([35, Lemma 2 (adapted for clustered scheduling)]). *There are at most c jobs per cluster with an incomplete request for a replica of a shared resource ℓ_a at any time.*

▶ **Lemma 11** ([35, Lemma 4]). *Under RRPD, if a job J_i that requires a replica of ℓ_a is pi-blocked waiting for a replica of ℓ_a it either has an incomplete request for a replica of ℓ_a or it is a priority donor.*

GIPP. At a very high level, the GIPP works as follows. For each group, we instantiate a separate instance of the RNLP. Crucially, the choice of token lock and RSM used to instantiate each instance of the RNLP must not violate group independence preservation, that is, any progress mechanisms employed must lend themselves to group independence preservation. Progress mechanisms like priority boosting that rely on elevating a job's priority can cause jobs that never request shared resources to incur release-blocking, which precludes the property of group independence preservation; this is highly undesirable in the presence of latency-sensitive tasks [6]. Furthermore, progress mechanisms that rely on the ability to directly compare priorities across clusters can result in unbounded pi-blocking (i.e., the

blocking depends on some other task's WCET) [6]. The challenges to realizing the GIPP are then to **(i)** construct an appropriate token lock and RSM, **(ii)** prove the token lock and RSM satisfy the required properties of the RNLP, **(iii)** prove the optimality of the GIPP under s-oblivious analysis, and **(iv)** prove that the GIPP is group independence-preserving.

We construct the token lock in Section 4.1, and the RSM in Section 4.2. We then show how to use these two components to realize the GIPP in Section 4.3.

4.1 An Independence-Preserving k-Exclusion Locking Protocol

To realize the GIPP we use a single token lock that is common to all the instantiations of the RNLP. If there are r groups (and therefore r instances of the RNLP), then a token lock that arbitrates access to r distinct token types, where each token type has m replicas, will suffice. However, as stated earlier, any such token lock must lend itself to independence preservation. To the best our knowledge, no such k-exclusion locking protocol (i.e., token lock) exists for clustered scheduling. To realize such a token lock, we generalize the R^{2}DGLP [35], which satisfies the requirements just described, with the exception that it was designed for globally-scheduled systems. Ward et al. use the term replica instead of token when discussing shared resources in the context of RRPD and R^{2}DGLP; we use the terms interchangeably.

As discussed in Section 2, the R^{2}DGLP uses RRPD as a progress mechanism. Thus, to generalize the R^{2}DGLP to clustered scheduling, the requirement that priorities across all jobs are comparable must be lifted. Additionally, RRPD alone is not enough to ensure progress [35], which means that replica-holders (i.e., token holders) are not guaranteed to be scheduled without the aid of an additional progress mechanism. The R^{2}DGLP solves this with priority inheritance, as the protocol targets globally-scheduled systems. However, the R^{2}DGLP does not strictly mandate the use of priority inheritance, instead, any locking protocol that utilizes RRPD must satisfy the following property [35, Section 3].

P1 A job J_i with an incomplete replica request makes progress (i.e., either J_i is scheduled itself or the replica-holding job that J_i is waiting for is scheduled) if J_i has sufficient priority to be scheduled in $C(T_i)$.

We introduce the *Clustered k-Exclusion Independence-Preserving Protocol* (CKIP) as a generalization of R^{2}DGLP that is non-nested independence preserving, asymptotically optimal under s-oblivious analysis, and employable under clustered scheduling. The CKIP is realized by having tasks compete amongst each other in their home clusters under the rules of RRPD, but not across clusters. This is possible as priorities can be directly compared within each cluster. However, this means that priority inheritance can no longer be used to ensure that replica holders make progress. To this end, we employ *allocation inheritance* [21, 22, 20] (sometimes referred to as *migratory priority inheritance* [6, 12]), an independence-preserving progress mechanism that works across clusters, and which is also used in the OMIP [6]. We define allocation inheritance in the context of the CKIP and GIPP as follows.

▶ **Definition 12** (allocation inheritance)**.** Let J_i be a job that holds a replica of a shared resource ℓ_a that has $k \geq 1$ replicas, and W_i be the set of jobs across all clusters waiting to a acquire a replica of ℓ_a. Under allocation inheritance (AI), if J_i is not scheduled and there exists a job $J_k \in W_i \cup \{J_i\}$ that has sufficient priority to be scheduled in $C(T_k)$, then J_i migrates to $C(T_k)$ (if necessary) and runs with J_k's priority. While J_i executes in $C(T_k)$ with J_k's priority, we call J_k an *allocation donor*. Once J_i releases ℓ_a, it migrates back to $C(T_i)$ (if necessary) and resumes execution when it has sufficient priority. Finally, J_i's allocation donor (if any) ceases to be an allocation donor when J_i releases ℓ_a.

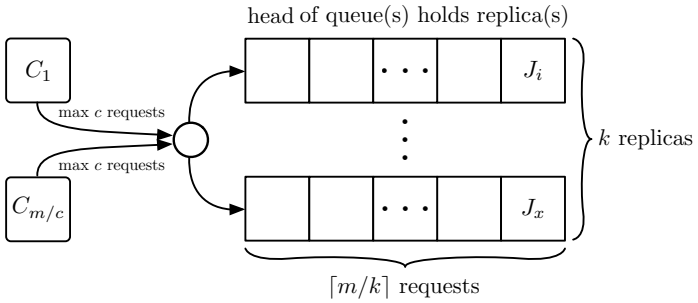


Figure 5 Queuing structure of the CKIP, which has been adapted from the R^{2}DGLP [35]. At any time, each of the m/c clusters has at most c incomplete requests for a replica of a given shared resource. Requests are enqueued into the replica queue with the least number of requests in it.

Now armed with an independence-preserving progress mechanism [6],[3] we can construct the CKIP by adapting the rules that define the R^{2}DGLP [35, Section 4]. The rules and structure of the CKIP differ enough from the R^{2}DGLP that its rules do not directly apply. Therefore, we present the modified rules and structure in full below.

Structure. Tasks compete for a set of q shared resources $\Gamma = \{\ell_1, \ldots, \ell_q\}$ where each resource ℓ_a has $k \geq 1$ replicas. Nested requests are not permitted. Each of the k replicas has an associated FIFO queue of size $\lceil m/k \rceil$ that jobs are placed in when requesting a replica; we use KQ_a to refer to any one of the queues for ℓ_a. The queuing structure of the CKIP closely resembles the R^{2}DGLP and is depicted in Figure 5. The following rules for CKIP focus on a single replicated resource $\ell_a \in \Gamma$, though they directly apply to all resources in Γ.

K1 Jobs issue requests subject to the rules of RRPD. When J_i requests ℓ_a, it is enqueued into a KQ_a with the fewest number of jobs in it, and suspends while it waits.

K2 J_i's request for ℓ_a is satisfied when it becomes the head of KQ_a, and thus becomes ready.

K3 While J_i is the head of KQ_a, it benefits from AI, but only with respect to the other jobs in the same KQ_a as J_i (i.e., W_i is comprised of the jobs in KQ_a that J_i is the head of).

K4 When J_i's request for ℓ_a is completed, it is dequeued from KQ_a and the new head (if any) acquires the replica. If J_i had benefited from AI, it returns to its home cluster and assumes its former (possibly donated) priority. If J_i has a priority donor due to RRPD in $C(T_i)$, the donor may now issue a request subject to the rules of RRPD.

▶ **Lemma 13.** *Rule **K3** ensures property **P1**.*

Proof. If J_i in KQ_a has sufficient priority to be scheduled in $C(T_i)$, then under AI, the head of KQ_a can migrate to $C(T_i)$ and execute with J_i's priority (if necessary). Therefore, the replica-holder is scheduled and J_i makes progress. ◀

▶ **Lemma 14.** *A job J_i that incurs pi-blocking while acting as a priority donor under the rules of RRPD makes progress.*

[3] One might wonder whether the CKIP and hence the GIPP could also be realized with a progress mechanism that does not result in inter-cluster migrations. Unfortunately, that is not the case: it is generally impossible for a protocol to ensure bounded pi-blocking, be independence-preserving, *and* avoid inter-cluster migrations at the same time even in the non-nested case [6].

Proof. Let J_x be the job that J_i donates its priority to. Then J_x has an incomplete request for a replica of a shared resource ℓ_a that both jobs require. Because J_i has sufficient priority to be scheduled in $C(T_i)$ – otherwise it would not incur s-oblivious pi-blocking – J_x does as well, as $C(T_i) = C(T_x)$ and since J_x receives its effective priority from J_i. Therefore, J_x makes progress by Lemma 13, which means J_i does as well. ◀

▶ **Lemma 15.** *The CKIP ensures property **T1** with respect to each replicated resource.*

Proof. RRPD is orchestrated on a per-cluster basis under the CKIP, and so we can reason about each cluster individually as if it were a lone globally-scheduled system with c processors. Then, by Lemma 10 there are at most c incomplete requests for a given replicated resource per cluster, and therefore at most m across a clustered system as $\frac{m}{c} \cdot c = m$. ◀

▶ **Lemma 16.** *The CKIP ensures property **T2**.*

Proof. By Lemma 11, a job J_i that requires a replica of a shared resource ℓ_a has an incomplete request, or is a priority donor. By Lemma 14, J_i makes progress while acting as a priority donor, and by Lemma 13, J_i makes progress while it has an incomplete request. Thus, J_i makes progress if it incurs pi-blocking while waiting for a token (i.e., replica of ℓ_a). ◀

▶ **Theorem 17.** *J_i incurs at most $(2\lceil m/k \rceil - 1) \cdot L^{max}$ s-oblivious pi-blocking while waiting to acquire a replica of a shared resource ℓ_a.*

We refer the reader to an online appendix [29] for the proof of Theorem 17 that establishes the CKIP's optimal $\mathcal{O}(m/k)$ bound on pi-blocking under s-oblivious analysis, as it follows analogously to the proof of the optimality of the R^{2}DGLP [35, Section 4].

▶ **Theorem 18.** *The CKIP is non-nested independence-preserving under any JFLP scheduler.*

Proof. Under the CKIP, requests for replicas of shared resources are arbitrated under the rules of RRPD in each cluster. The rules of RRPD are such that jobs do *not* incur pi-blocking for resources they do not access [35]. Thus, any pi-blocking J_i incurs due to requests for a resource $\ell_a \notin \gamma_i$ would need to be the result of the use of AI as a cross-cluster progress mechanism. However, any job that benefits from AI only executes with the priority of another job currently waiting on a replica of the same resource, which precludes J_i from incurring pi-blocking due to jobs inheriting allocations [6]. Thus, if $N_{i,a} = 0$ then $b_{i,a} = 0$. ◀

4.2 An Independence-Preserving RSM

The GIPP requires that its RSM lends itself to independence preservation, and no such suitable RSM for clustered scheduling has been proposed in prior work. Thus, we introduce the *Allocation Inheritance Resource Satisfaction Mechanism* (AI-RSM). The AI-RSM applies to clustered scheduling, and utilizes AI to ensure progress among jobs competing for shared resources. Let $ts(J_i)$ be the time that J_i acquired its token (and therefore entered the RSM), and let $sr(J_i, t)$ be the set of resources J_i holds at time t. Finally, we let $A_{i,a,t} = \{J_k \mid ts(J_k) < ts(J_i) \wedge (\ell_a \in sr(J_k, t) \vee \exists \ell_b \in sr(J_k, t) \text{ s.t. } \ell_b \succ \ell_a)\}$ denote the set of jobs that can prevent J_i from acquiring ℓ_a at time t, which follows from the rules of the RNLP.

A1 When the AI-RSM prevents J_i from acquiring a shared resource ℓ_a at time t, J_i donates its allocation to the job in $A_{i,a,t}$ with the earliest timestamp under the rules of AI.

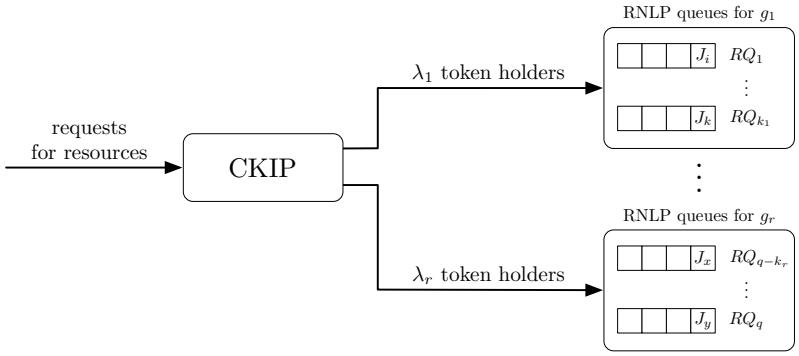


Figure 6 Queuing structure of the GIPP. A request for a token of a group is first arbitrated by the CKIP before the request is passed to the group's corresponding instance of the RNLP.

▶ **Lemma 19.** *The AI-RSM ensures property **R1** under clustered scheduling when waiting is realized by suspending.*

Proof. Let J_i be a job that is pi-blocked by the RSM at time t while it waits to acquire a shared resource ℓ_a. Then, there must exist some job $J_k \in A_{i,a,t}$ that prevents J_i from acquiring ℓ_a by the rules of the RNLP. By Rule **A1**, the job $J_k \in A_{i,a,t}$ with the earliest timestamp is eligible to inherit J_i's priority in J_i's home cluster. Since J_i incurs s-oblivious pi-blocking, it has one of the c highest priorities in its cluster, and hence the inherited priority enables J_k to be scheduled in J_i's home cluster. Thus, at least one job preventing J_i from acquiring ℓ_a is scheduled and J_i therefore makes progress. ◀

We now have a group independence-preserving token lock, and an RSM with an independence-preserving progress mechanism. We use these components to realize the GIPP.

4.3 Structure and Analysis of The GIPP

We next define the structure of the GIPP and then establish its asymptotic optimality with respect to s-oblivious pi-blocking, and that it is group independence-preserving.

There are m tokens for each group $g_x \subseteq \Gamma$; a token for g_x is denoted with λ_x. A single instance of the CKIP arbitrates access to the set $\Lambda = \{\lambda_1, \ldots, \lambda_r\}$ of replicated tokens, and an instance of the RNLP with the AI-RSM is instantiated for each group. The CKIP instance serves as a common token lock among all the instances of the RNLP. To execute an outermost critical section under the GIPP for resources in g_x a job must **(i)** compete for and acquire a token λ_x under the CKIP, **(ii)** compete in g_x's instance of the AI-RSM under the rules of the RNLP [32, Section 3], and **(iii)** release λ_x upon completing its outermost critical section and exiting the AI-RSM. The queuing structure of the GIPP is depicted in Figure 6.

▶ **Theorem 20.** *The maximum amount of s-oblivious pi-blocking incurred per outermost request under the GIPP is* $(2m-1) \cdot L^{max} = \mathcal{O}(m)$ *under any JLFP scheduler.*

Proof. The CKIP satisfies property **T1** by Lemma 15, and the AI-RSM satisfies property **R1** by Lemma 19. Therefore, the maximum amount of s-oblivious pi-blocking a job incurs while in the AI-RSM is $L^{\text{RSM}} = (m-1) \cdot L^{\max}$ by Theorem 9, as the corresponding RNLP proof generalizes to any protocol that satisfies these two properties.

Under the rules of the RNLP, a job holds a token for the entire duration it is in the RSM, and releases its token after completing its outermost critical section. The RNLP

proof of Theorem 9 establishes that a job is pi-blocked for at most $m-1$ outermost critical sections while in any RSM. Thus, after a job completes its outermost critical section, the maximum amount of time the job holds a token is $L^{\text{token}} = m \cdot L^{\max}$. By Theorem 17, a job waiting to acquire a token under the CKIP incurs at most $(2\lceil m/k \rceil - 1) \cdot L^{\text{token}}$ units of s-oblivious pi-blocking. Under the GIPP, there are m tokens for each group (i.e., $k = m$), so the pi-blocking incurred while waiting for a token simplifies to L^{token} as $(2\lceil m/m \rceil - 1) = 1$. The total s-oblivious pi-blocking a job occurs per outermost request is then the sum of the pi-blocking incurred while waiting to acquire a token and while competing in the AI-RSM, which is $L^{\text{token}} + L^{\text{RSM}} = m \cdot L^{\max} + (m-1) \cdot L^{\max} = (2m-1) \cdot L^{\max}$. ◀

▶ **Theorem 21.** *The GIPP is group independence-preserving under any JLFP scheduler.*

Proof. By the structure of the GIPP, a job interacts with the CKIP for the entire duration it interacts with the GIPP. Under the CKIP, nested requests are not permitted, so each shared resource (e.g., token type) forms its own group. When each group consists of a single resource, the definition of group independence preservation trivially reduces to non-nested independence preservation. Thus, it follows that the CKIP is group independence-preserving.

To prove the claim, it hence suffices to show that the GIPP remains group independence-preserving while token holders compete for shared resources in the AI-RSM. We prove this by contradiction: suppose a job J_i that does not request a token λ_x for a group g_x incurs pi-blocking due to a request for λ_x by a job J_k. Under the AI-RSM, J_k's effective priority is raised to that of another job competing for resources in g_x. Thus, if J_k's effective priority in $C(T_i)$ is greater than J_i's, there must be another job J_h in $C(T_i)$ that requires resources in g_x and has a higher base priority than J_i. As this argument applies to any such job J_k, and since AI establishes a one-to-one relationship among donors and recipients (i.e., donor priorities are not "duplicated"), it follows there are at least c higher-base-priority jobs in $C(T_i)$, and hence J_i does not incur pi-blocking according to Definition 1. Contradiction. ◀

It is worth noting that, in special cases, the GIPP emulates the behavior of the RNLP and the OMIP, respectively. When there is just a single group (i.e., $r = 1$), the GIPP effectively reduces to the RNLP in the sense that there is a single, global token lock. Conversely, when $r = |\Gamma|$, the GIPP behaves like the OMIP. These cases are examined further in Section 6.

5 Fine-Grained Pi-Blocking Analysis

We next introduce a fine-grained, non-asymptotic pi-blocking analysis for the GIPP, which we formulate as a *Linear Programming* (LP) problem as in prior work [7, 6, 36]. The asymptotic bound presented in Section 4.3 is coarse-grained as it does not reflect the exact resources each task requests, individual critical section lengths, nor the frequency of critical sections. The following analysis is fine-grained in the sense that it considers these workload-specific aspects to obtain a less pessimistic, but still safe upper-bound on s-oblivious pi-blocking.

In the following, we let T_i denote the task under analysis and let J_i denote an arbitrary job of T_i. For each other task T_x, we let θ_x^i denote a bound on the maximum number of jobs of T_x that overlap with J_i (i.e., that are pending while J_i is pending). Let r_i and r_x be the maximum response times of T_i and T_x, respectively. Then $\theta_x^i = \left\lceil \frac{r_i + r_x}{p_x} \right\rceil$ [5, 7].

We denote T_x's y^{th} outermost critical section as $O_{x,y}$, its length as $L_{x,y}^O$, the set of resources it accesses as $S_{x,y}$, and define $O_x(g) \triangleq \{O_{x,y} \mid S_{x,y} \subseteq g\}$. Note that the index y is used only for enumeration purposes and does *not* imply an order; each job of T_x may execute its outermost critical sections in any order. For each task $T_x \neq T_i$, each outermost request

$O_{x,y}$, and $v \in \{1, \ldots, \theta_x^i\}$, we introduce two real-valued variables $X_{x,y,v}^T$ and $X_{x,y,v}^R$, each with domain $[0, 1]$. These variables are called *blocking fractions* [7] and serve to encode the portion of T_x's v^{th} overlapping instance of $O_{x,y}$ that contributes to the pi-blocking that J_i incurs. We use $X_{x,a,v}^T$ and $X_{x,a,v}^R$ to respectively encode the *token* and *RSM blocking* that J_i incurs, where token blocking refers to the time spent waiting to acquire a token, and RSM blocking refers to time spent waiting for a resource within the AI-RSM.

With these definitions in place, the pi-blocking incurred by J_i can be stated as

$$b_i = \sum_{T_x \neq T_i} \sum_{O_{x,y}} \sum_{v=1}^{\theta_x^i} (X_{x,y,v}^T + X_{x,y,v}^R) \cdot L_{x,y}^O. \tag{1}$$

By interpreting Equation (1) as the **objective function** of an LP maximization problem, we obtain an upper bound b_i on the maximum pi-blocking incurred by any J_i [7, 6, 36]. To avoid excessive pessimism, we introduce in the following LP constraints that reflect both the invariants of the GIPP and properties of the specific task set under analysis.

To start, we prevent any blocking critical section from being counted twice.

▶ **Constraint 22.** $\forall T_x \neq T_i : \forall O_{x,y} : \forall v : X_{x,y,v}^T + X_{x,y,v}^R \leq 1$

Proof. A single critical section of T_x cannot cause J_i to experience token blocking and RSM blocking *simultaneously*: to wait for a resource within the AI-RSM, J_i must already hold a token, but while J_i competes for a token it cannot yet interact with the RSM. Thus, the combined token and RSM blocking induced by one of T_x's critical sections cannot exceed the length of the critical section (i.e., the blocking fractions sum to at most one). ◀

Next, we bound the maximum amount of token blocking that J_i incurs. In preparation, let τ_k be the set of tasks assigned to cluster C_k, and $\tau_k' = \tau_k \setminus \{T_i\}$. Furthermore, $\phi_{i,g} \triangleq |\{O_{i,y} \mid S_{i,y} \cap g \neq \emptyset\}|$ denotes the number of times J_i issues an outermost request for a resource in g, and $\beta_{k,g} \triangleq \left|\left\{T_x \;\middle|\; T_x \in \tau_k \wedge \bigcup_{O_{x,y}} S_{x,y} \cap g \neq \emptyset\right\}\right|$ is the number of tasks in C_k that request a resource in g. Based on these definitions, we state a bound on the number of times that J_i must wait for a token. In Equation (2), let k denote the index of $C(T_i)$.

$$W_{i,g} \triangleq \begin{cases} 0 & \beta_{k,g} \leq c \\ \min(\phi_{i,g}, \phi_{i,g}') & \text{otherwise} \end{cases} \qquad \text{where } \phi_{i,g}' \triangleq \left(\sum_{T_x \in \tau_k'} (\phi_{x,g} \cdot \theta_x^i)\right) - c + 1 \tag{2}$$

▶ **Lemma 23.** *$W_{i,g}$ upper-bounds the number of times J_i must wait for a token of group g.*

Proof. By case analysis. Let k denote the index of $C(T_i)$. First, if $\beta_{k,g} \leq c$, then there are at most c tasks in $C(T_i)$ that ever require a token for group g (including T_i). There are never more than c token holders per cluster under the CKIP, which effectively reserves c tokens for each cluster. Thus, whenever J_i requires a token for group g, one is always available, and J_i never needs to wait for a token: $W_{i,g} = 0$ if $\beta_{k,g} \leq c$.

Otherwise, if $\beta_{k,g} > c$, then J_i requires a token no more than $\phi_{i,g}$ times, and hence clearly $W_{i,g} \leq \phi_{i,g}$. To obtain $W_{i,g} \leq \phi_{i,g}'$, consider the number of times that other tasks require a token while J_i is pending, which is bounded by $\sum_{T_x \in \tau_k'} (\phi_{x,g} \cdot \theta_x^i)$. Since J_i must wait for a token only if all c tokens are currently held by other tasks, the worst case occurs when $c - 1$ tokens are held "indefinitely" (i.e., if they remain unavailable throughout the interval during which J_i is pending) and the remaining $\phi_{i,g}' = \left(\sum_{T_x \in \tau_k'} (\phi_{x,g} \cdot \theta_x^i)\right) - c + 1$ requests must all share a single token, and thus $W_{i,g} \leq \phi_{i,g}'$. ◀

We further restrict under which conditions J_i incurs token blocking at all.

▶ **Lemma 24.** *J_i incurs token blocking (i.e., it incurs pi-blocking while waiting to acquire a token for a group g) only if it is a priority donor under the rules of the RRPD.*

Proof. Recall that GIPP allocates group tokens using the CKIP, and that the CKIP employs RRPD. As there are $k = m$ tokens per group (i.e., replicas per token type), the CKIP's per-replica FIFO queues have length $\lceil \frac{m}{k} \rceil = 1$. Since by Rule K2 the head of each per-replica FIFO queue holds the replica (i.e., a token), and jobs enter a queue immediately when they issue a request (Rule K1), it follows that J_i can be waiting for a token only *before* it issues its request for a token, that is, while it serves as a priority donor under the rules of the RRPD in the time span between requiring a token and actually issuing a request (recall Figure 4). ◀

Lemmas 23 and 24 allow us to establish a constraint on token blocking due to each task.

▶ **Constraint 25.** $\forall g \in G : \forall T_x \neq T_i : \sum_{O_{x,y} \in O_x(g)} \sum_{v=1}^{\theta_x^i} X_{x,y,v}^T \leq W_{i,g}$

Proof. Suppose not. Then there exists a task T_x that token-blocks J_i with more than $W_{i,g}$ outermost critical sections (w.r.t. some group g). If $W_{i,g} = 0$, then by Lemma 23 J_i must never wait to acquire a token for group g, which immediately yields a contradiction. Hence assume $W_{i,g} > 0$. As J_i waits for a token for g at most $W_{i,g}$ times (Lemma 23), this implies that there exists an outermost critical section $O_{i,z}$ executed by J_i such that J_i is delayed, while waiting to acquire a token for g in preparation of $O_{i,z}$, by at least two outermost critical sections of T_x. By Lemma 24, J_i is a priority donor while it incurs token blocking. According to the rules of the RRPD (recall Section 2), J_i becomes a priority donor at most once per request, and only for a single other request: either immediately when J_i requires a token to commence $O_{i,z}$, or not at all. It follows that T_x must pi-block J_i with *two* distinct outermost critical sections while J_i continuously serves as the priority donor of some job J_l. Since under the rules of the RRPD J_i ceases to be a priority donor as soon as J_l finishes its outermost critical section (i.e., when J_l releases its token), J_l cannot be a job of T_x. Hence there remains only one other way for an outermost critical section of T_x to delay J_i, namely by delaying one or more requests of J_l within the RSM, which transitively causes J_i to incur pi-blocking. Consider the later of T_x's two outermost critical sections that cause J_i to incur pi-blocking while donating its priority to J_l. Since it is the second *outermost* critical section of T_x in this interval, T_x necessarily must have acquired a token for group g strictly after the beginning of the interval, when J_l was already holding its token. However, the RSM satisfies resource requests strictly in order of increasing token-acquisition timestamps, and thus T_x's second outermost critical section cannot delay J_l. Contradiction. ◀

We similarly bound the aggregate token blocking across all tasks in each cluster.

▶ **Constraint 26.** $\forall g \in G : \forall k \in \{1, \ldots, \frac{m}{c}\} : \sum_{T_x \in \tau'_k} \sum_{O_{x,y} \in O_x(g)} \sum_{v=1}^{\theta_x^i} X_{x,y,v}^T \leq W_{i,g} \cdot \min(c, \beta_{k,g})$

Proof. Again the case of $W_{i,g} = 0$ is trivial; hence assume $W_{i,g} > 0$ and suppose the invariant does not hold. Then analogously to the proof of Constraint 25, there exists a contiguous interval $[t_1, t_2)$ and a cluster C_k such that both **(i)** J_i serves as the priority donor of some job J_l throughout $[t_1, t_2)$, and **(ii)** J_i incurs pi-blocking during $[t_1, t_2)$ due to at least

$\min(c, \beta_{k,g}) + 1$ outermost critical sections executed by tasks in τ_k. Also analogously to the proof of Constraint 25, no task delays J_i with more than one outermost critical section during $[t_1, t_2)$. Because the RSM satisfies requests strictly in order of increasing token-acquisition timestamps, any job that delays J_l within the RSM (and hence transitively causes J_i to incur token blocking) must have acquired its token for group g before J_l did so, and hence no later than at time t_1. Furthermore, any such job necessarily releases its token only some time after t_1. At time t_1 there hence exist at least $\min(c, \beta_{k,g}) + 1$ token-holding jobs in cluster C_k. However, the CKIP ensures that no more than c jobs in C_k hold a token at any time, and by definition at most $\beta_{k,g}$ tasks in τ_k require a token for group g. Contradiction. ◀

This concludes the constraints on token blocking. We next constrain RSM blocking, that is, the pi-blocking incurred by J_i while it holds a token and waits for individual resources. First, we observe on a per-cluster basis that RSM blocking across all tasks is limited by the number of resource requests that J_i issues because the RSM serves jobs in timestamp order.

▶ **Constraint 27.** $\forall g \in G : \forall k \in \{1, \dots, \frac{m}{c}\}$:

$$\sum_{T_x \in \tau'_k} \sum_{O_{x,y} \in O_x(g)} \sum_{v=1}^{\theta_x^i} X^R_{x,y,v} \leq \begin{cases} \phi_{i,g} \cdot \min(c, \beta_{k,g}) & T_i \notin \tau_k \\ \phi_{i,g} \cdot \min(c-1, \beta_{k,g} - 1) & \text{otherwise} \end{cases}$$

Proof. If $\phi_{i,g} = 0$, then J_i does not access resources in group g and the invariant is trivial. Hence, suppose the invariant does not hold. First consider the case $T_i \notin \tau_k$: then there exists an interval $[t_1, t_2)$ during which J_i holds a token for group g such that J_i incurs RSM blocking due to more than $\min(c, \beta_{k,g})$ outermost critical sections executed by jobs in C_k. Analogously to the proof of Constraint 26, it follows that more than $\min(c, \beta_{k,g})$ jobs must hold a token for group g at time t_1, which is impossible. In the second case, if $T_i \in \tau_k$, then J_i necessarily holds one of the c available tokens (otherwise it could not interact with the RSM), so that there are only $c - 1$ tokens available to other tasks, and only $\beta_{k,g} - 1$ other tasks in τ_k that are also accessing resources in group g. ◀

Next we constrain RSM blocking in a more detailed fashion by considering which critical sections actually conflict within the RSM. The resulting constraint is essential to realizing the benefits of the increased parallelism in nested locking protocols (relative to coarse-grained group-locking) at analysis time, and not just at runtime. To this end, we require some further terminology. First, we say that a set of resources s is *possibly conflicting* with another set of resources s' if either **(i)** $s \cap s' \neq \emptyset$ or **(ii)** $\exists \ell_b \in s, \ell_a \in s'$ such that $\ell_a \succ \ell_b$. Second, we define $F_i(s) \triangleq |\{O_{i,y} \mid S_{i,y} \textit{ possibly conflicts with } s\}|$ to count the number of outermost critical sections of T_i in which it needs resources that the RSM may have to withhold due to other jobs holding resources in s. Finally, we let $S^i(g) \triangleq \{S_{x,y} \mid T_x \neq T_i \wedge S_{x,y} \cap g \neq \emptyset\}$ denote the set of all combinations of resources in group g acquired by other tasks. Based on these definitions, we constrain RSM blocking as follows.

▶ **Constraint 28.** $\forall g \in G : \forall s \in S^i(g) : \forall k \in \{1, \dots, \frac{m}{c}\}$:

$$\sum_{T_x \in \tau'_k} \sum_{\substack{O_{x,y} \text{ s.t.} \\ S_{x,y} \subseteq s}} \sum_{v=1}^{\theta_x^i} X^R_{x,y,v} \leq \begin{cases} F_i(s) \cdot \min(c, \beta_{k,g}) & T_i \notin \tau_k \\ F_i(s) \cdot \min(c-1, \beta_{k,g} - 1) & \text{otherwise} \end{cases}$$

Proof. Consider any group g, set of resources $s \in S^i(g)$, and cluster C_k. For J_i to incur RSM blocking when issuing a request for some resource $\ell_b \in g$, there must exist a job J_x with

an earlier token-acquisition time that either already holds ℓ_b, or that holds a resource $\ell_a \in g$ such that $\ell_a \succ \ell_b$. In other words, J_i incurs RSM blocking only if $\{\ell_b\}$ is possibly conflicting with the set of resources already held by jobs with earlier timestamps. Recall that $F_i(s)$ counts the number of outermost critical sections of T_i accessing resources that are possibly conflicting with s. It follows that J_i executes at most $F_i(s)$ outermost critical sections that may encounter RSM blocking due to outermost critical sections of tasks in τ'_k that access s or a subset of s (i.e., the requests represented on the left-hand side of the constraint). As in the proof of Constraint 27, it is easy to show that no more than $\min(c, \beta_{k,g})$ (respectively, $\min(c-1, \beta_{k,g}-1)$) outermost critical sections can cause RSM blocking per each outermost critical section of J_i if $T_i \notin \tau_k$ (respectively, $T_i \in \tau_k$). The bound follows. ◀

6 Schedulability Experiments

We compared the GIPP against the OMIP and the RNLP as **(i)** they are both asymptotically optimal with respect to s-oblivious pi-blocking, **(ii)** the OMIP [6] is the only prior independence-preserving locking protocol for clustered scheduling, and **(iii)** the RNLP [32] is the only prior fine-grained nested locking protocol that ensures asymptotically optimal pi-blocking bounds under clustered scheduling.

To conduct meaningful experiments, one requires a fine-grained (i.e., non-asymptotic) pi-blocking analysis. The OMIP has such analysis, which is also formulated as an LP [6], which we use in these experiments. However, for the RNLP, there are surprisingly no fine-grained bounds available in prior work. We therefore had to adapt our analysis of the GIPP (Section 5) to the RNLP. To this end, we created an instantiation of the RNLP called the CA-RNLP that uses the the CKIP as its token lock and AI-RSM as its RSM. As the RNLP and hence the CA-RNLP uses only a single, global token lock, our analysis in Section 5 is applicable to the CA-RNLP if one presumes that *all* resources are part of a *single* group.

We generated task sets with Emberson et al.'s method [17] via the SchedCAT [9] library and considered all combinations of the following parameter choices. Each task set consisted of $n \in \{2.0m, 3.0m\}$ tasks with total utilization $U \in \{0.4m, 0.6m\}$ to be scheduled on $m \in \{4, 8, 16\}$ processors under partitioned EDF scheduling ($c = 1$). There were $n^{\text{nl}} \in \{0.0m, 0.5m, 1.0m\}$ latency-sensitive tasks in the task set, and $n - n^{\text{nl}}$ non-latency-sensitive (i.e., regular) tasks. Periods were drawn uniformly at random from the set $\{1ms, 2ms, 4ms, 5ms, 8ms\}$ for latency-sensitive tasks, and from the set $\{10ms, 20ms, 25ms, 40ms, 50ms, 100ms, 125ms, 200ms, 250ms, 500ms, 1000ms\}$ for regular tasks; the specific period values were inspired by Kramer et al.'s work on producing real-world automotive benchmarks [24]. Latency-sensitive tasks shared three resources in a single group, and each issues one or two outermost requests for resources in their group at random. The regular tasks shared twelve resources split into equally sized groups of $g^{\text{size}} = \{1, 2, 3, 4\}$ resources. Each regular task accessed just one group, and issued from $\{1, \ldots, N^{\max}\}$ outermost requests for resources in that group at random, where $N^{\max} \in \{1, 2, 3\}$. The outermost critical section lengths of latency-sensitive tasks were drawn uniformly at random from $[1\mu s, 15\mu s]$, and from $[1\mu s, mcsl]$ for regular tasks, where $mcsl$ was varied across $[5\mu s, 1000\mu s]$ in increments of $5\mu s$.

Each resource group was of one of two types $g^{\text{type}} \in \{wide, deep\}$. More precisely, we say a shared resource $\ell_b \in \Gamma$ is a *top-level resource* if $\nexists \ell_a \in \Gamma$ s.t. $\ell_a \succ \ell_b$, and consider a group to be *wide* if at least half of its resources are top-level, and *deep* otherwise. Tasks were assigned a minimal set of requests to form the desired groups; afterwards tasks were assigned outermost requests for resources in their corresponding group at random until each made $N^{\max}$ requests (per job); each outermost request contained a nested request with probability 0.5.

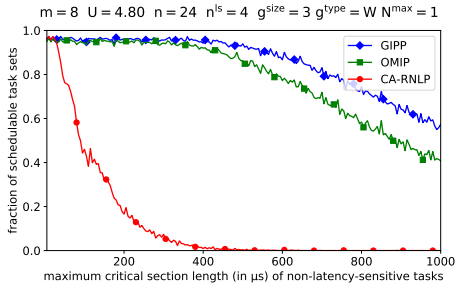


Figure 7 The presence of latency-sensitive tasks dominates schedulability.

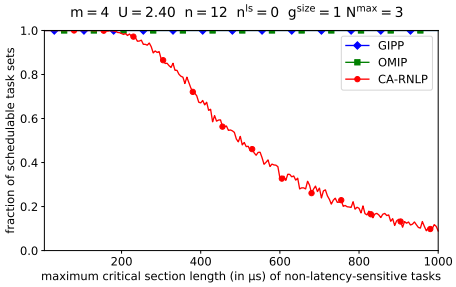


Figure 8 Token contention dominates schedulability.

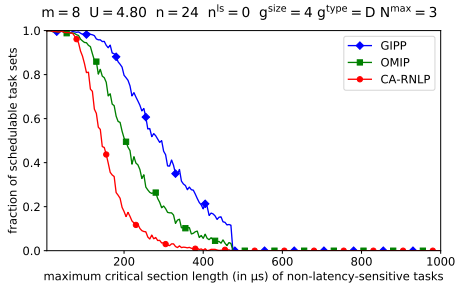


Figure 9 High contention among few groups dominates schedulability.

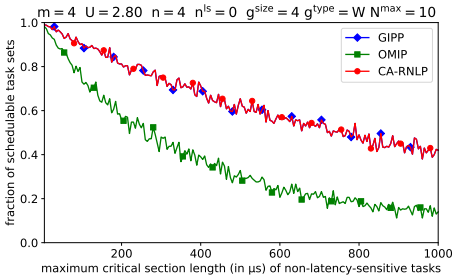


Figure 10 A single group lock limits schedulability for certain asymmetric access patterns.

There are 864 combinations of the varied parameters $(n, n^{\text{nl}}, U, m, N^{\max}, g^{\text{size}}, g^{\text{type}})$. Some combinations are duplicate in effect (i.e., wide and deep groups of size one are the same), or not possible to generate. After removing such combinations from consideration, there remain 522 combinations. For each combination, we generated 1000 task sets per *mcsl* value (i.e., per point on the x-axis), and then tested each task set for schedulability under the GIPP, OMIP, and the CA-RNLP. Figures 7 to 9 show three select scenarios out of the 522 combinations; all graphs are provided in an online appendix [29]. (Figure 10 further shows a hand-crafted example to highlight a specific behavior as discussed below.)

In our large-scale experiments, both the GIPP and the OMIP retained a high level of schedulability for most parameter configurations. In most cases, the CA-RNLP provided a substantially lower level of schedulability than the GIPP or the OMIP. We now outline some key observations drawn from the large-scale schedulability experiments. As our **first observation**, we notice that the GIPP performs noticeably better than the OMIP and the CA-RNLP in most of our experiments, and never worse. In corner cases, the performance of the GIPP approaches that of the OMIP when $g^{\text{size}} = 1$, and the CA-RNLP when $g^{\text{size}} = |\Gamma|$ (i.e., the total number of resources). As a result, the GIPP never performs worse than the better-performing of the two baselines. This is apparent in Figure 7 and Figure 8. Our **second observation** is that the isolation of latency-sensitive tasks greatly impacts schedulability. When latency-sensitive tasks must compete with regular tasks for the same set of tokens, schedulability quickly drops under the CA-RNLP as *mcsl* increases (see Figure 7). As our **third observation**, we note that even in the absence of latency-sensitive tasks, schedulability is greatly affected by the use of a single, global token lock (i.e., if tokens are not group-specific). As *mcsl* increases, schedulability under the CA-RNLP drops at a roughly linear rate in Figure 8, whereas task sets remain schedulable for the entire range of

mcsl under the GIPP and the OMIP. This demonstrates that a single token lock ultimately becomes a bottleneck for otherwise schedulable task sets. As a **fourth observation**, the benefits of (group) independence preservation diminish under high contention for all resources. This is shown in Figure 9, where roughly the same pattern of schedulability is seen under all three protocols. In contrast, the benefits of independence preservation are more clearly seen when there is a greater degree of isolation as in Figure 8.

This concludes our discussion of the large-scale schedulability study. As the workload generator with the chosen set of parameters did not generate a sufficient number of task sets that hit weak points of the OMIP, we further set up an experiment with a hand-crafted task set; the result is shown in Figure 10. The four tasks of the task set share a single wide group of size four. Three of the tasks access only top-level resources, while the fourth task makes the necessary requests to form the group. The rules and structure of the GIPP and the CA-RNLP allow for top-level resources to be acquired independently, which is not possible with the OMIP's group locks. Thus, as a final and **fifth observation**, we note that fine-grained nested locking offers a noticeable increase in schedulability when compared to group locks for heavily asymmetric resource access patterns.

7 Conclusion

We have examined the concept of independence preservation in the context of fine-grained nested locking. On the one hand, *outer-lock independence preservation* yields non-optimal bounds on s-oblivious pi-blocking. On the other hand, *group independence preservation* can be realized with asymptotically optimal pi-blocking bounds (under s-oblivious analysis), as demonstrated with the GIPP. To obtain the GIPP, we constructed the CKIP as a building block, which is noteworthy in itself as it is the first asymptotically optimal, non-nested independence-preserving, k-exclusion lock for clustered scheduling. Finally, we demonstrated with empirical experiments using a fined-grained pi-blocking analysis of the GIPP that it avoids the bottleneck imposed by the RNLP's single token lock (or group locks under the OMIP), thereby allowing latency-sensitive tasks to be accommodated.

In future work, it would be interesting to extend the GIPP to semi-partitioned scheduling [1, 2, 13]. It will also be necessary to study the real-world overheads (e.g., cache misses, TLB flushes, inter-processor interrupts, etc.), which the GIPP is particularly exposed to due to its use of allocation inheritance, in a practical system such as LITMUS$^{\mathrm{RT}}$ [5, 15].

References

1 James H. Anderson, Vasile Bud, and UmaMaheswari C. Devi. An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, 2005. doi:10.1109/ECRTS.2005.6.

2 Andrea Bastoni, Björn B. Brandenburg, and James H. Anderson. Is Semi-Partitioned Scheduling Practical? In *23rd Euromicro Conference on Real-Time Systems (ECRTS'11)*, 2011. doi:10.1109/ECRTS.2011.20.

3 Alessandro Biondi, Björn B. Brandenburg, and Alexander Wieder. A Blocking Bound for Nested FIFO Spin Locks. In *38th Real-Time Systems Symposium (RTSS'17)*, 2017. doi:10.1109/RTSS.2016.036.

4 Aaron Block, Hennadiy Leontyev, Björn B. Brandenburg, and James H Anderson. A Flexible Real-Time Locking Protocol for Multiprocessors. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'07)*. IEEE, 2007. doi:10.1109/RTCSA.2007.8.

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

6 Björn B. Brandenburg. A Fully Preemptive Multiprocessor Semaphore Protocol for Latency-Sensitive Real-Time Applications. In *25th Euromicro Conference on Real-Time Systems (ECRTS'13)*, 2013. doi:10.1109/ECRTS.2013.38.

7 Björn B. Brandenburg. Improved Analysis and Evaluation of Real-Time Semaphore Protocols for P-FP Scheduling. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'13)*, 2013. doi:10.1109/RTAS.2013.6531087.

8 Björn B. Brandenburg. Multiprocessor Real-Time Locking Protocols: A Systematic Review. *arXiv:1909.09600 [cs]*, 2019. arXiv:1909.09600.

9 Björn B. Brandenburg. SchedCAT: The schedulability test collection and toolkit, January 2020. URL: https://github.com/brandenburg/schedcat.

10 Björn B. Brandenburg and James H. Anderson. Optimality Results for Multiprocessor Real-Time Locking. In *31st IEEE Real-Time Systems Symposium (RTSS'10)*. IEEE, 2010. doi:10.1109/RTSS.2010.17.

11 Björn B. Brandenburg and James H. Anderson. Real-Time Resource-Sharing under Clustered Scheduling: Mutex, Reader-Writer, and k-Exclusion Locks. In *9th ACM International Conference on Embedded Software (EMSOFT'11)*, 2011. doi:10.1145/2038642.2038655.

12 Björn B. Brandenburg and Andrea Bastoni. The Case for Migratory Priority Inheritance in Linux: Bounded Priority Inversions on Multiprocessors. In *14th Real Time Linux Workshop (RTLWS'12)*, 2012.

13 Björn B. Brandenburg and Mahircan Gul. Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations. In *37th IEEE Real-Time Systems Symposium (RTSS'16)*. IEEE, 2016. doi:10.1109/RTSS.2016.019.

14 Alan Burns and Andy J. Wellings. A Schedulability Compatible Multiprocessor Resource Sharing Protocol - MrsP. In *25th Euromicro Conference on Real-Time Systems (ECRTS'13)*, 2013. doi:10.1109/ECRTS.2013.37.

15 John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. LITMUS^{RT} : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *27th IEEE Real-Time Systems Symposium (RTSS'06)*, 2006. doi:10.1109/RTSS.2006.27.

16 Arvind Easwaran and Björn Andersson. Resource Sharing in Global Fixed-Priority Preemptive Multiprocessor Scheduling. In *30th IEEE Real-Time Systems Symposium (RTSS'09)*, 2009. doi:10.1109/RTSS.2009.37.

17 Paul Emberson, Roger Stafford, and Robert I. Davis. Techniques for the Synthesis of Multiprocessor Tasksets. In *Proceedings of the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS'10)*, 2010.

18 Dario Faggioli, Giuseppe Lipari, and Tommaso Cucinotta. The Multiprocessor Bandwidth Inheritance Protocol. In *22nd Euromicro Conference on Real-Time Systems (ECRTS'10)*, 2010. doi:10.1109/ECRTS.2010.19.

19 Dario Faggioli, Giuseppe Lipari, and Tommaso Cucinotta. Analysis and Implementation of the Multiprocessor Bandwidth Inheritance Protocol. *Real-Time Systems*, 48(6):789–825, 2012. doi:10.1007/s11241-012-9162-0.

20 Philip Holman. *On the Implementation of Pfair-Scheduled Multiprocessor Systems*. PhD thesis, University of North Carolina at Chapel Hill, 2004.

21 Philip Holman and James H. Anderson. Object sharing in Pfair-Scheduled Multiprocessor Systems. In *14th Euromicro Conference on Real-Time Systems (ECRTS'02)*, 2002. doi:10.1109/EMRTS.2002.1019191.

22 Philip Holman and James H. Anderson. Locking Under Pfair Scheduling. *ACM Transactions on Computer Systems*, 24(2):140–174, 2006. doi:10.1145/1132026.1132028.

23 Catherine E. Jarrett, Bryan C. Ward, and James H. Anderson. A Contention-Sensitive Fine-Grained Locking Protocol for Multiprocessor Real-Time Systems. In *23rd International Conference on Real Time and Networks Systems (RTNS'15)*, 2015. doi:10.1145/2834848.2834874.

24 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'15)*, 2015.

25 Catherine E. Nemitz, Tanya Amert, and James H. Anderson. Using Lock Servers to Scale Real-Time Locking Protocols: Chasing Ever-Increasing Core Counts. In Sebastian Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems (ECRTS'18)*, 2018. doi:10.4230/LIPIcs.ECRTS.2018.25.

26 Catherine E. Nemitz, Tanya Amert, and James H. Anderson. Real-time Multiprocessor Locks with Nesting: Optimizing the Common Case. *Real-Time Systems*, 55(2):296–348, 2019. doi:10.1007/s11241-019-09328-w.

27 R. Rajkumar. Real-Time Synchronization Protocols for Shared Memory Multiprocessors. In *10th International Conference on Distributed Computing Systems (ICDCS'90)*, 1990. doi:10.1109/ICDCS.1990.89257.

28 Ragunathan Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, USA, 1991.

29 James Robb and Björn B. Brandenburg. Nested, but Separate: Isolating Unrelated Critical Sections in Real-Time Nested Locking (extended version). Technical Report MPI-SWS-2020-002, Max Planck Insitute for Software Systems, 2020. URL: https://www.mpi-sws.org/tr/2020-002.pdf.

30 Liu Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990. doi:10.1109/12.57058.

31 Hiroaki Takada and Ken Sakamura. Real-time Scalability of Nested Spin Locks. In *2nd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'95)*, 1995. doi:10.1109/rtcsa.1995.528766.

32 Bryan C. Ward and James H. Anderson. Supporting Nested Locking in Multiprocessor Real-Time Systems. In *24th Euromicro Conference on Real-Time Systems (ECRTS'12)*, 2012. doi:10.1109/ECRTS.2012.17.

33 Bryan C. Ward and James H. Anderson. Fine-Grained Multiprocessor Real-Time Locking with Improved Blocking. In *21st International Conference on Real Time and Networks Systems (RTNS'13)*, 2013. doi:10.1145/2516821.2516843.

34 Bryan C. Ward and James H. Anderson. Multi-Resource Real-Time Reader/Writer Locks for Multiprocessors. In *28th IEEE International Parallel and Distributed Processing Symposium (IPDPS'14)*, 2014. doi:10.1109/IPDPS.2014.29.

35 Bryan C. Ward, Glenn A. Elliott, and James H. Anderson. Replica-Request Priority Donation: A Real-Time Progress Mechanism for Global Locking Protocols. In *18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2012. doi:10.1109/RTCSA.2012.26.

36 Alexander Wieder and Björn B. Brandenburg. On Spin Locks in AUTOSAR: Blocking Analysis of FIFO, Unordered, and Priority-Ordered Spin Locks. In *34th IEEE Real-Time Systems Symposium (RTSS'13)*, 2013. doi:10.1109/RTSS.2013.13.

37 Maolin Yang, Alexander Wieder, and Björn B. Brandenburg. Global Real-Time Semaphore Protocols: A Survey, Unified Analysis, and Comparison. In *36th IEEE Real-Time Systems Symposium (RTSS'15)*, 2015. doi:10.1109/RTSS.2015.8.

The Safe and Effective Use of Learning-Enabled Components in Safety-Critical Systems

Kunal Agrawal
Washington University in Saint Louis, MO, USA
kunal@wustl.edu

Sanjoy Baruah
Washington University in Saint Louis, MO, USA
baruah@wustl.edu

Alan Burns
The University of York, UK
alan.burns@york.ac.uk

Abstract

Autonomous systems increasingly use components that incorporate machine learning and other AI-based techniques in order to achieve improved performance. The problem of assuring correctness in safety-critical systems that use such components is considered. A model is proposed in which components are characterized according to both their worst-case and their typical behaviors; it is argued that while safety must be assured under all circumstances, it is reasonable to be concerned with providing a high degree of performance for typical behaviors only. The problem of assuring safety while providing such improved performance is formulated as an optimization problem in which performance under typical circumstances is the objective function to be optimized while safety is a hard constraint that must be satisfied. Algorithmic techniques are applied to derive an optimal solution to this optimization problem. This optimal solution is compared with an alternative approach that optimizes for performance under worst-case conditions, as well as some common-sense heuristics, via simulation experiments on synthetically-generated workloads.

2012 ACM Subject Classification Computer systems organization → Embedded and cyber-physical systems; Computing methodologies → Machine learning; Software and its engineering → Real-time schedulability

Keywords and phrases Learning-enabled components (LECs), Safety-critical systems, Typical analysis, Performance optimization, Run-time monitoring

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.7

Funding *Kunal Agrawal*: National Science Foundation Grants CCF-1733873 and CCF-1439062.
Sanjoy Baruah: National Science Foundation Grants CNS-1814739 and CPS-1932530.
Alan Burns: EPSRC (UK) grants EP/K011626/1, EP/P003664/1, and EP/N023641/1.

1 Introduction

Many autonomous cyber-physical systems (CPS's) such as self-driving cars are safety-critical, and must have their safety properties verified before they can be considered for deployment. However approaches that have traditionally been used for the purposes of performing safety assurance in safety-critical systems do not seem directly applicable to modern autonomous CPS's due to multiple reasons, including the presence of complex and adaptive functionalities that are based upon the incorporation of machine learning techniques that are not well understood in the way that components traditionally used in safety-critical systems are. The importance, as well as the enormous complexity, of the problem of obtaining assurance for autonomous CPS's that incorporate machine learning has been widely recognized, and approaches for solving this problem are being actively sought – consider the following example initiatives:

- The *Assured Autonomy* Program [8] of the United States Defense Advanced Research Projects Agency (DARPA) has a goal of creating technology for establishing assurance of CPS's that contain "*Learning-Enabled Components*" (**LECs**), which are an abstraction defined in [8] that generalizes a wide variety of popular machine learning approaches.
- In a similar vein, the *Assuring Autonomy International Programme* [1] is an initiative funded by the international insurance company Lloyd's of London at the University of York (UK), motivated in part by a 2016 study by Lloyd's identifying assurance and regulation as being amongst the biggest obstacles to realising the benefits of robotics and autonomy.
- Yet another important example project in this space is the *Bounded Behavior Assurance* initiative [6], spearheaded by the major US defense contractor Northrop Grumman Corporation, which seeks to define processes for establishing assurance (and eventually, obtaining certification) that the behavior of unmanned aerial vehicles that use machine learning to make safety-critical and mission-critical decisions will always remain within pre-specified bounds.

It is widely accepted that *predictability* of run-time behavior [10] is very important for the purposes of assuring safety in safety-critical systems. Although most non-trivial safety-critical systems inevitably encounter some unpredictability in run-time behavior, safety-critical systems designers have developed a range of advanced and sophisticated techniques for dealing with inherent run-time unpredictability with regards to extra-functional properties such as timing (the duration required to complete execution) or energy consumption. However, safety-critical systems that make use of LECs tend to additionally not be predictable from the functional perspective: the precise "worth" or value of a computation performed by an LEC that incorporates deep learning or similar AI-based techniques is often not easily predicted beforehand. This aspect of run-time unpredictability has not been as widely studied in the safety-critical systems community: How should one deal with such functional uncertainty in safety-critical systems? In this paper, we continue our investigations, first reported in [3], of one possible approach for doing so for a particular form of computation involving LECs, that possess the following characteristics:

- The overall computation can be considered to be a multi-stage one, in which a series of functional blocks are to be executed in a specified sequence. For an execution of the computation to be considered *correct* (and hence safe), a specified minimum level of service must be obtained cumulatively over all the stages; we assume that this minimum level of service is quantified as a numerical target value.
- We have a choice of different alternative implementations for each stage of the computation, some or all of which may involve the use of LECs. Each implementation takes some *duration* to complete execution, and achieves an associated *value* – a quantitative measure of the quality of the computation that was achieved by executing that implementation.[1] We perform the complete end-to-end computation by selecting and executing exactly one of the implementation choices for each stage, in sequence. The total value obtained by the end-to-end computation is defined to be the sum of the values associated with the implementations that were selected for the individual stages.
- We can monitor the computation – determine certain aspects of system state – after each stage during run-time.

[1] It may be convenient to think of this value as a measure of the progress that will be made towards achieving the overall objective for the computation, if this implementation were selected for this stage of the computation.

For computations that can be characterized in this manner. we consider different approaches for scheduling the computation in ways that **guarantee safety** – i.e., guarantee that the computation will *achieve the specified target value of quality of service* – and **optimize for performance** – specifically, *reduce the overall duration* of the computation. We provide a precise formulation of the scheduling problem that needs to be solved as a constrained optimization problem (Section 2); explore a number of algorithms, ranging from provably optimal ones (Section 3) to simple heuristics that are efficiently implementable (Section 4), for solving this problem; and compare these different algorithms via simulation experiments on randomly-generated synthetic workloads (Section 4).

Organization. The reminder of this paper is organized in the following manner. In Section 2 we formally specify the problem that is studied in this paper, and present arguments making the case that this is indeed a relevant problem that merits further exploration by the real-time computing community. In Section 3 we present algorithms for solving this problem, and derive properties that demonstrate the correctness and characterize the efficacy of our algorithms. In Section 4 we explore some heuristic solution techniques: although these heuristics are easily shown to be non-optimal, some simulation-based experiments on synthetically generated workloads indicates that carefully chosen heuristics may be adequate for solving some specific classes of the problem in a near-optimal manner. We discuss some directions in which the problem we are studying here could be generalized in Section 5, and conclude in Section 6 by placing this work within a larger persepctive on the design and analysis of highly complex safety-critical cyber-physical systems.

2 Model and Problem Statement

In this section we motivate and formally define the model of LEC-enabled computations that we are considering in this paper. We highlight, via illustrative examples, some of the challenges that arise in the pre-runtime safety analysis of such systems that we seek to solve, and formally define the problem that is studied in the remaining sections of this paper.

As discussed in Section 1 above, we consider multi-stage computations in which a series of functional blocks are to be executed in a specified sequence, and we have a choice of several different implementations for each stage. Let n denote the number of stages, and m the maximum number of available alternative implementations for any stage. (An example multi-stage computation with $n = 2$ and $m = 3$ is depicted in Figure 1.) Let $\mathcal{V} \in \mathbb{N}$ denote a target value that must be obtained cumulatively across all stages of the computation. We will use the notation $\mathcal{I}_{i,j}$ to denote the j'th implementation choice for the i'th stage, $0 \leq i < n$ and $0 \leq j < m$. Let $V_{i,j} \in \mathbb{N}$ denote the value that is obtained by executing the implementation $\mathcal{I}_{i,j}$, and let $C_{i,j} \in \mathbb{N}$ denote the duration of this execution – we do not assume that the numerical values of these parameters are known prior to executing $\mathcal{I}_{i,j}$ (and indeed allow for the possibility that they may be different on different executions of $\mathcal{I}_{i,j}$). Consider some execution of the end-to-end computation, and let $\phi(i)$ denote the implementation of the i'th stage that is chosen (i.e., $\mathcal{I}_{i,\phi(i)}$ is the executed implementation) for each i, $0 \leq i < n$. Note that the function $\phi(\cdot)$ thus specifies the schedule for the computation – we will henceforth often refer to it as *the scheduling function*, or simply *the schedule*, for the computation. It is required that the scheduling function $\phi(\cdot)$ satisfy the constraint that $\sum_i V_{i,\phi(i)} \geq \mathcal{V}$; from amongst all the functions ϕ that do so, we seek one that minimizes $\sum_i C_{i,\phi(i)}$. That is, our CORRECTNESS CONSTRAINT is that the sum of the values returned across all n stages should equal (or exceed) the specified threshold value $\mathcal{V}$, while the PERFORMANCE OBJECTIVE is that the cumulative duration of the computation be minimized.

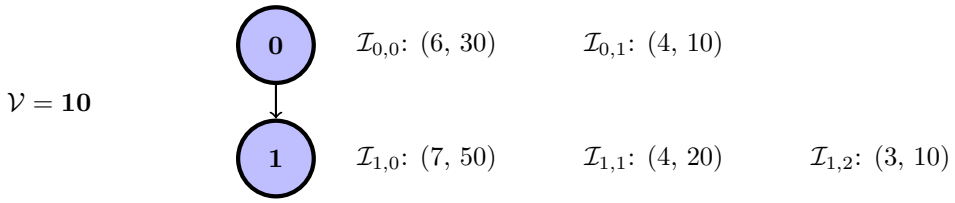


Figure 1 An instance discussed in Example 1: a 2-stage computation with a choice of two possible implementations for the first stage and three for the second, that must achieve a value of at least 10 ($\mathcal{V} = 10$). The ordered pairs represent the $(v_{i,j}, c_{i,j})$ parameters of the implementations.

As stated above, the C_{ij}, V_{ij} values are unknown prior to actually executing $\mathcal{I}_{i,j}$, and will in general take on different values each time $\mathcal{I}_{i,j}$ is executed. In order to be able to do pre-run-time verification, it is necessary that *worst-case bounds* be known on the values that these quantities may take. Let $c_{i,j}$ and $v_{i,j}$ denote safe worst-case bounds on the values of $C_{i,j}$ and $V_{i,j}$ respectively, that can be determined beforehand; by "safe," we mean that it is guaranteed that $C_{i,j} \leq c_{i,j}$ and $V_{i,j} \geq v_{i,j}$ for all executions of $\mathcal{I}_{i,j}$.

- The value of $c_{i,j}$ is what is commonly referred to in the real-time computing literature as the *worst-case execution time* (WCET) of the implementation $\mathcal{I}_{i,j}$, and may be determined using the wide range of tools, techniques, and methodologies for WCET-determination [11] that have been developed within the real-time computing community.[2]
- We require that similar tools, techniques, and methodologies be developed that enable us to determine lower bounds on the value of the computation that is performed by an LEC. While we recognize that this is a major "ask" that will require a large concerted effort on the part of the safety-critical systems community, we believe it is unavoidable – we don't really see any other path to enabling the safe and effective use of LECs in safety-critical systems.

If we are to be able to verify correctness of a given computation prior to run-time, it is evident that there should exist some implementation of each stage such that the worst-case value bounds of these implementations sum to at least the target value – this correctness requirement is formalized later (in Expression 1) as a *feasibility test*, and computations passing the feasibility test are said to be *feasible*. If a computation is deemed feasible, our approach, as briefly described in Section 1, will generate a schedule prior to run-time that can be verified for correctness, and shown to always have an acceptably small duration. What properties must such a schedule satisfy? Let us try to understand some of the issues involved via a simple example.

▶ **Example 1.** Consider a 2-stage computation ($n = 2$) with a choice of 2 implementations per stage ($m = 2$), for which correctness requires that a cumulative value of at least 10 be obtained ($\mathcal{V} = 10$) – see Figure 1. (As explained in the caption to the figure, each implementation is labeled here with an ordered pair denoting the minimum value that is

[2] **Note:** in much of the remainder of this paper we will make the simplifying assumption that the actual run-time of implementations does not vary much from their specified WCET's: $C_{i,j} \approx c_{i,j}$ for all (i, j). This simplifying assumption allows us to highlight the primary focus of this paper, which is that of dealing with the uncertainty that is inherent in a priori characterizing the *value* obtained from many LEC's. (This is also a reasonable assumption for the many Deep-Learning based LECs that are implemented as a known number of "layers" of matrix computations and therefore known to have very predictable and deterministic running times.) In Section 5 we briefly discuss how our work may be generalized by removing this simplifying assumption, and incorporating considerations of uncertainty along both the value and the timing dimensions.

guaranteed to be obtained by choosing to execute the implementation, and the maximum duration that the implementation may take to execute.) We note that since we have a choice of two implementations for the first stage and a choice of three implementations for the second stage, there is a total of $2 \times 3 = \mathbf{6}$ possible distinct schedules:

No.	Schedule	Min. cumulative value	Guaranteed Correct?	Max. delay
1.	$\langle \mathcal{I}_{0,0}; \mathcal{I}_{1,0} \rangle$	$6 + 7 = \mathbf{13}$	**Y**	$30 + 50 = \mathbf{80}$
2.	$\langle \mathcal{I}_{0,0}; \mathcal{I}_{1,1} \rangle$	$6 + 4 = \mathbf{10}$	**Y**	$30 + 20 = \mathbf{50}$
3.	$\langle \mathcal{I}_{0,0}; \mathcal{I}_{1,2} \rangle$	$6 + 3 = \mathbf{9}$	N	—
4.	$\langle \mathcal{I}_{0,1}; \mathcal{I}_{1,0} \rangle$	$4 + 7 = \mathbf{11}$	**Y**	$10 + 50 = \mathbf{60}$
5.	$\langle \mathcal{I}_{0,1}; \mathcal{I}_{1,1} \rangle$	$4 + 4 = \mathbf{8}$	N	—
6.	$\langle \mathcal{I}_{0,1}; \mathcal{I}_{1,2} \rangle$	$4 + 3 = \mathbf{7}$	N	—

Of these six, three can assure a value of at least 10 are are thus guaranteed correct; from amongst these correct schedules, the schedule $\langle \mathcal{I}_{0,0}; \mathcal{I}_{1,1} \rangle$ has the lowest associated delay bound of 50.

If a ***static*** schedule –one in which the choice of implementations is finalized prior to run-time– is desired then it is reasonable to choose the schedule $\langle \mathcal{I}_{0,0}; \mathcal{I}_{1,1} \rangle$ since it is the correct schedule that guarantees the best (smallest) maximum delay. Such a static schedule has the benefit of not requiring any run-time monitoring: under the assumption that the worst-case characterizations (of both the value obtained and the duration required) of the implementations is correct, this schedule is guaranteed to obtain the required value of $\mathcal{V} = 10$ and is hence correct; additionally, from among all such static schedules the selected one is, informally speaking, clearly the "best" choice.

Suppose now that the schedules are permitted to be ***adaptive***: they may be changed between stages in response to additional information that is revealed during run-time by monitoring the actual behavior (recall that the c_{ij} and v_{ij} values represent conservative worst-case estimates: the actual behavior experienced during run-time may well turn out to be superior to these worst-case estimates). Consider the following possible scenarios.

1. If while proceeding to execute the chosen schedule, $\langle \mathcal{I}_{0,0}; \mathcal{I}_{1,1} \rangle$, upon completing the first stage it is discovered that this stage actually returns a value ≥ 7. Now since the value remaining to be acquired is ≤ 3, it is safe to switch to the implementation $\mathcal{I}_{1,2}$ for the second stage; doing so further reduces the maximum delay bound to $30 + 10 = \mathbf{40}$.
2. If we had instead initially chosen the schedule $\langle \mathcal{I}_{0,1}; \mathcal{I}_{1,0} \rangle$ (which is also guaranteed to be correct, albeit with a larger maximum delay bound of 60) and upon executing the first stage discovered that the value returned is ≥ 6, it then becomes safe to switch to the implementation $\mathcal{I}_{1,1}$ for the second stage, and doing so results in a maximum delay bound of $10 + 20 = \mathbf{30}$.

In the second scenario above, we started out with a sub-optimal schedule (from the static perspective), but run-time monitoring enabled the adaptive schedule to achieve a smaller delay than was achieved in the first scenario by starting out with the optimal static schedule. So when adaptive scheduling is permitted, which schedule should we start out with? In the absence of additional information (for instance, in our example above how *likely* is it that the value returned by $\mathcal{I}_{0,1}$ will actually exceed 6?), we cannot think of any reason to go with an initial schedule other than the one with the best worst-case static guarantee (in our example, the schedule $\langle \mathcal{I}_{0,0}; \mathcal{I}_{1,1} \rangle$). However, it may be the case that additional information (over and above the worst-case bounds $c_{i,j}$ and $v_{i,j}$) regarding run-time behavior is available prior to run-time; if so, it may be possible to use such additional information to

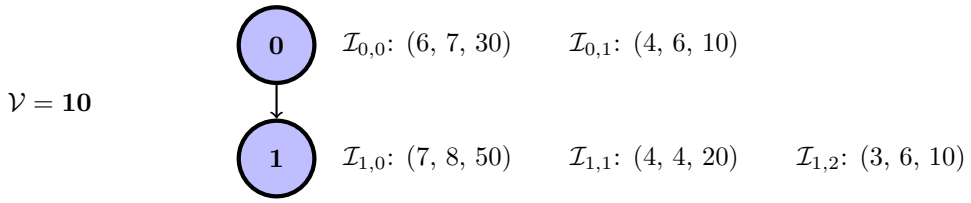


Figure 2 The example instance of Figure 1, with the typical values obtained by implementations also specified. The 3-tuples pairs represent the $(v_{i,j}, v^T_{i,j}, c_{i,j})$ parameters of the implementations.

further optimize the initial schedule *provided* we are able to do so without compromising the correctness guarantee. An example of such additional information that may be available, that we consider to be particularly interesting and useful, is suggested by Quinton et al. [9] via the concept of **typical analysis**. The idea behind typical analysis is that while a worst-case characterization of a system must encompass all possible behaviors of the system, a "typical" characterization excludes pathological behaviors that are extremely unlikely to occur in practice.[3] Let us suppose that our multi-stage computation is subjected to such typical-case analysis, and let the parameter $v^T_{i,j}$ denote the typical value obtained by the implementation $\mathcal{I}_{i,j}$; the interpretation being that implementation $\mathcal{I}_{i,j}$ will obtain a value no smaller than $v^T_{i,j}$ ($V_{i,j} \geq v^T_{i,j}$) in all non-pathological executions of the computation.

With this notion of typical-case analysis in mind, let us revisit the scenarios introduced above. We can now state that if $v^T_{0,1} \geq 6$ (in Figure 2, the example instance of Figure 1 is updated with the typical value parameters, the $v^T_{i,j}$'s, also specified), then choosing implementation $\mathcal{I}_{0,1}$ rather than $\mathcal{I}_{0,0}$ for the first stage

1. Assures *correctness* under all circumstances, and
2. *offers superior performance* – a smaller maximum delay – under typical circumstances.

Problem statement. We now summarize our workload model, and the problem we seek to solve. A problem *instance* is defined by specifying values for

- the number of stages n of the multi-stage computation;
- the maximum number of alternative implementations m for each stage;
- the target value $\mathcal{V}$ that is needed for correctness;
- the worst-case and typical values $v_{i,j}$ and $v^T_{i,j}$ for the value-obtained parameters for each implementation $\mathcal{I}_{i,j}$, $0 \leq i < n, 0 \leq j < m$; and
- the worst-case execution time $c_{i,j}$ of implementation $\mathcal{I}_{i,j}$, $0 \leq i < n, 0 \leq j < m$. (As previously noted –see footnote 2– we could, in principle, have worst-case and typical-case characterizations of the execution time parameters as well; we have chosen to not do so here in order to keep things simple. We will revisit and briefly discuss the implications of this choice in Section 6.)

To summarize, an ***instance*** I of our problem is characterized by

1. the $n \times m$ implementations $\mathcal{I}_{i,j}$, $(i,j) \in \{0,1,\ldots,(n-1)\} \times \{0,1,\ldots,(m-1\}$, with each $\mathcal{I}_{i,j}$ characterized by the 3-tuple $(v_{i,j}, v^T_{i,j}, c_{i,j})$; and
2. the target value $\mathcal{V}$.

[3] E.g., worst-case characterization of the value obtained by an implementation may be obtained by performing static analysis of the implementation, making worst-case (or pessimistic) assumptions and rigorously proving the value that will be obtained under these assumptions. In contrast, a typical characterization of this value may be obtained via extensive experimentation and measurement, executing the implementation under a wide range of "typical" conditions and using the smallest measured value that is obtained.

A *schedule* for such an instance is a function $\phi : [0, 1, \ldots n-1] \rightarrow [0, 1, \ldots, m-1]$, with $\phi(i)$ denoting the implementation of stage i that is selected for execution in the schedule. An instance is said to be *feasible* if it is possible to schedule it in a manner that guarantees that the cumulative value $\mathcal{V}$ is obtained for all possible actual run-time behaviors; it is obvious that *a necessary and sufficient feasibility condition* is that

$$\left(\sum_{i=0}^{n-1} \max_{j=0}^{(m-1)} \{v_{i,j}\} \right) \geq \mathcal{V} \tag{1}$$

Given a feasible instance, we seek a scheduling strategy for choosing an implementation $\phi(i)$ for each stage i of the instance. We require that our strategy only make *safe* choices: never choose an implementation that could result in an incorrect schedule. If several safe choices are available, then our choice is governed by our optimization objective of minimizing execution duration; in this paper we advocate for choosing an implementation that *guarantees the smallest overall delay under all typical circumstances*, and derive algorithms that enable us to do so. In the following example we illustrate some of the issues that arise in choosing to optimize for the typical case rather than the worst case.

▶ **Example 2.** Let us revisit the example of Figure 2. We note that

- If we were to choose implementation $\mathcal{I}_{0,0}$ for the first stage we are guaranteed to obtain a value of at least 6 units, which would leave us requiring the remaining 4 units of value from the second stage. Either of the implementations $\mathcal{I}_{1,0}$ or $\mathcal{I}_{1,1}$ is able to guarantee this; hence it is safe to choose implementation $\mathcal{I}_{0,0}$ for the first stage.
- If we were to instead choose to execute implementation $\mathcal{I}_{0,1}$ for the first stage, then we are guaranteed to obtain a value of at least 4 units. This would leave us requiring 6 units of value from the second stage. Since implementation $\mathcal{I}_{1,0}$ is able to guarantee this, we conclude that it is safe to choose implementation $\mathcal{I}_{0,0}$ for the first stage.

We have seen that from the perspective of safety we may therefore choose either implementation $\mathcal{I}_{0,0}$ or implementation $\mathcal{I}_{0,1}$ for the first stage. Let us examine how we would choose between the two implementations.

Worst-case analysis. We separately consider both choices of implementation for the first stage:

1. If we choose $\mathcal{I}_{0,0}$, then the lowest value that we would obtain is 6, and will then need an additional 4 units of value from the second stage. That requires that we choose $\mathcal{I}_{1,0}$ or $\mathcal{I}_{1,1}$ for the second stage; we prefer $\mathcal{I}_{1,1}$ since it has a lower delay, and hence end up with a bound of $30 + 20 = \mathbf{50}$ on the delay.
2. If, on the other hand, we were to choose $\mathcal{I}_{0,1}$ for the first stage, then the lowest value that we would obtain is 4. That would leave us 6 units of value short of the target, and we must choose implementation $\mathcal{I}_{1,0}$ for the second stage. Thus results in a cumulative bound of $10 + 50 = \mathbf{60}$ on the delay under typical circumstances.

Since $50 < 60$, we would choose the implementation $\mathcal{I}_{0,0}$ for the first stage, and are guaranteed a maximum cumulative delay of 50 time units.

Typical-case analysis. We again consider both choices for the first stage:

- If we choose $\mathcal{I}_{0,0}$, then the lowest value that we would obtain under typical conditions is 7. Since we will then need an additional 3 units of value from the second stage, we may choose implementation $\mathcal{I}_{1,2}$ for the second stage, for a bound of $30 + 10 = \mathbf{40}$ on the delay under typical circumstances.

- If, on the other hand, we were to choose $\mathcal{I}_{0,1}$ for the first stage, then the lowest value that we would obtain under typical conditions is 6. That would leave us 4 units of value short of the target; since these 4 units of value must be guaranteed to be obtained in the second stage, we must choose implementation $\mathcal{I}_{1,1}$ for the second stage.[4] This results in a cumulative bound of $10 + 20 = \mathbf{30}$ on the delay under typical circumstances.

Since the delay bound is smaller if we choose $\mathcal{I}_{0,1}$, we would prefer the implementation $\mathcal{I}_{0,1}$ for the first stage. By so doing, we are able to guarantee a maximum cumulative delay of 30 time units under typical conditions (as opposed to the 50 time units that was guaranteed above, when we chose implementation $\mathcal{I}_{0,0}$ based on worst-case analysis). However if we were to encounter atypical conditions whilst executing implementation $\mathcal{I}_{0,1}$ and obtain less than $v^T_{0,1} = 6$ units of value, we may end up with a shortfall of more than four units of value and therefore need to execute implementation $\mathcal{I}_{1,0}$ for the second stage, for a cumulative delay of $10 + 50 = \mathbf{60}$ time units.

Example 2 above illustrates the difference in the kinds of performance guarantees that are made by "traditional" worst-case analysis, and the typical-case analysis that we are proposing:

> *Typical analysis provides superior performance* (in this example, smaller cumulative delay) *under typical conditions, but may provide worse performance under atypical conditions. However, it does guarantee correctness* (in our example, to obtain a cumulative value of at least 10) *under all conditions, typical or atypical.*

3 Scheduling Algorithms and Analysis

In this section we will develop algorithms that solve the problem described in Section 2 above. We derive two separate algorithms: one for solving the (preëxisting) problem of optimizing performance for the worst case and the other, for the novel problem that is the major focus of this paper, of optimizing performance for the typical case (while assuring correctness under all cases, typical or not). It turns out (see Section 3.1 below) that both these optimization problems are NP-hard; hence, we should not expect to be able to find polynomial-time algorithms for solving them. We will however show that both our algorithms have *pseudo*-polynomial running time.

The remainder of this section is organized in the following manner. We show that our problems are unlikely to be solvable using polynomial-time algorithms in Section 3.1, develop pseudo-polynomial time algorithms for solving them in Sections 3.2–3.3, and formally specify their optimality properties in Section 3.4.

3.1 Computational Complexity

The problem we seek to solve is easily seen to be computationally intractable – NP-hard, by reduction from the Multiple-choice Knapsack Problem (MCKP) [7], a well-known NP-hard problem. The Multiple-choice Knapsack Problem (MCKP) may be defined in the following manner:

> Given k *classes* $N_1 \ldots, N_k$ of *items*, and a *knapsack* of capacity c. Each item $j \in N_i$ has a *profit* p_{ij} and a *weight* w_{ij}, and the problem is to choose one item from each class such that the profit sum exceeds p without having the weight sum exceed w.

[4] We could also have chosen $\mathcal{I}_{1,0}$, but that has a larger delay bound, and hence offers inferior performance.

To determine whether a given instance of the form described above $\in$ MCKP, we could reduce it to an instance of our problem that comprises k stages with a choice of $|N_i|$ implementations for the i'th stage for each $i, 1 \leq i \leq k$. The implementation $\mathcal{I}_{i,j}$ is assigned the parameters $v_{ij} = p_{ij}$, $v_{ij}^T = p_{ij}$, and $c_{ij} = w_{ij}$; the desired target value $\mathcal{V} = p$. It is evident that this instance of our problem can guarantee a delay not exceeding w if and only if the given instance $\in$ MCKP. Since determining whether an instance $\in$ MCKP is NP-hard, it follows that determining a schedule that optimizes for performance under typical conditions, while concurrently assuring correctness, is also NP-hard.

Observe that the instance of our problem obtained in the above reduction from MCKP has $v_{ij} = v_{ij}^T$ for all (i, j); hence, its optimal solution would be exactly the same regardless of whether one were optimizing for performance under worst-case assumptions or typical-case ones. This establishes that the problem of optimizing for the worst case while assuring correctness is also an NP-hard problem.

3.2 Algorithm Description

As stated earlier, our objective is to obtain a schedule that optimizes for performance (i.e., minimizes the duration of computation) while assuring correctness – i.e., obtaining a cumulative value no smaller than the specified target $\mathcal{V}$. Our approach toward achieving this is to construct, prior to run-time, a *lookup table* that will subsequently be used during run-time in a manner that is elaborated upon in Section 3.3.

Let us suppose that we have completed execution of the first $(i-1)$ stages during some execution of the computation, having obtained a cumulative value $\widehat{\mathcal{V}}$ by so doing, and wish to determine which implementation $\mathcal{I}_{i,j}$ of the i'th stage we should choose in order to optimize for performance whilst continuing to assure correctness (i.e., guaranteeing that we will be able to obtain a cumulative value $\mathcal{V}' \stackrel{\text{def}}{=} (\mathcal{V} - \widehat{\mathcal{V}})$ under all circumstances, typical or not). Below, we first describe how one can identify, for each stage i and each possible value for $\mathcal{V}'$, which implementations are safe to execute (in the sense of not compromising correctness) in Section 3.2.1. Once we have figured out how to identify the safe implementations, we separately discuss, in Section 3.2.2 how to choose amongst them in order to optimize for the two different performance criteria – optimizing for the worst case and for the typical case.

3.2.1 Identifying safe implementations

For each $i, 0 \leq i \leq n$, let Δ_i denote the largest value that we can guarantee to obtain over the remaining stages – stages $i, (i+1), \cdots, (n-1)$ – of the computation, based upon the characterizations of the implementations that are available to us. It is evident that

$$\begin{aligned} \Delta_n &= 0 \text{ (Since there is no stage } n) \\ \text{and } \Delta_i &= \Delta_{i+1} + \max_{j=0}^{m-1} \Big\{ v_{ij} \Big\} \end{aligned} \tag{2}$$

This computation is represented in pseudo-code form in Figure 3; since it comprises two nested loops, it is easily seen to take running time $\Theta(nm)$. (Notice that the feasibility condition of Expression 1 can be rewritten as $\mathcal{V} \geq \Delta_0$; i.e., the target value $\mathcal{V}$ needed for correctness is no smaller than the largest value that we can guarantee to obtain over [all] the stages $0, 1, \ldots, n-1$.)

Suppose now that that we are in the midst of executing the computation during some run – we have completed the stages $0, \cdots, (i-1)$ in a safe manner, and have an amount $\mathcal{V}'$ remaining of the target value to be acquired (implying that the stages $0, \cdots, (i-1)$ together

```
ComputeDeltas(I)
1  Δ_n = 0
2  for i = n − 1 downto 0
3        tmp = 0
4        for j = 0 to m − 1
5              if (v_{i,j} > tmp) then tmp = v_{i,j}
6        Δ_i = Δ_{i+1} + tmp
```

Figure 3 Pseudo-code for computing the Δ_i values – the maximum value that can be guaranteed over the stages $i, i+1, \cdots, (n-1)$ of the computation.

must have yielded a cumulative value $\mathcal{V} - \mathcal{V}'$). If we were to now choose the implementation $\mathcal{I}_{i,j}$ for the i'th stage and were to discover, upon having completed its execution, that doing so yielded the lowest (i.e., worst-case) value of $v_{i,j}$, we will need to obtain an additional amount $(\mathcal{V}' - v_{i,j})$ of value over the remaining stages $(i+1), \cdots, (n-1)$. Hence for it to be safe for us to choose the implementation $\mathcal{I}_{i,j}$ for the i'th stage, it is necessary (and sufficient) that

$$\left(v_{i,j} + \Delta_{i+1}\right) \geq \mathcal{V}' \tag{3}$$

Expression 3 above can be thought of as constituting a run-time safety check: it is safe to use the implementation $\mathcal{I}_{i,j}$ in order to obtain a remaining cumulative value $\mathcal{V}'$ if and only if Expression 3 evaluates to true.

3.2.2 Choosing an implementation to execute

Based upon the cumulative value that has been obtained over the first $(i-1)$ stages, we saw above how one can identify which of the provided implementations of the i'th stage are safe to execute. We now describe how we should choose from amongst these safe implementations in order to optimize for performance. We first consider, in Section 3.2.2.1, the case when we seek to optimize for performance in the worst case; subsequently in Section 3.2.2.2 we consider optimizing for the typical case.

3.2.2.1 Optimizing for the worst case

Conceptually speaking, we will build a table W with $(n+1)$ rows and $(\mathcal{V}+1)$ columns with the following interpretation. For any $i \in \{0, i, \ldots, n\}$ and any $\boldsymbol{v} \in \{0, 1, 2, \ldots, \mathcal{V}\}$, $W[i, \boldsymbol{v}]$ denotes the smallest delay bound that we can guarantee if we are required to obtain a target value of at least $\boldsymbol{v}$ over the stages $i, (i+1), \ldots, (n-1)$. The following recurrence defines the values for $W[i, \boldsymbol{v}]$:

$$\begin{aligned}
W[n, 0] &= 0 && \text{(Doing nothing, we trivially obtain a value 0)} \\
W[n, \boldsymbol{v}] &= \infty \text{ for all } \boldsymbol{v} > 0 && \text{(No positive value can be obtained over the non-existent } n\text{'th stage)} \\
W[i, \boldsymbol{v}] &= \min_{\{j \mid v_{ij} + \Delta_{i+1} \geq \boldsymbol{v}\}} \left\{ c_{ij} + W[i+1, \boldsymbol{v} - v_{ij}] \right\} && (4)
\end{aligned}$$

The third –recursive– equation above asserts that if we were to execute the j'th implementation, we would spend a duration c_{ij} and obtain a value v_{ij} in the worst case, thereby requiring an additional value $(\boldsymbol{v} - v_{ij})$ in the following stages. And the worst-case duration for this is, by definition, given by $W[i+1, \boldsymbol{v} - v_{ij}]$.

```
ComputeT-Table(I)
1  for i = (n − 1) downto 0 by (−1)
2      for v = 1 to V
3          T[i, v] = ∞// Initializing
4          for j = 0 to m − 1
5              if (v_{i,j} + Δ_{i+1} ≥ v)// I.e., implementation I_{i,j} is "safe"
6                  if (c_{i,j} + T[i + 1, v − v^T_{i,j}] < T[i, v])// Better than current choice?
7                      T[i, v] = c_{i,j} + T[i + 1, v − v^T_{i,j}]// Update best response duration
8                      I_T[i, v] = j // Update choice of implementation
```

Figure 4 Pseudo-code for computing the tables $T[\ ,\ ]$ and $I_T[\ ,\ ]$ – see Section 3.2.2.2.

Observe that in this third equation we are restricting the choice of implementations (the values that the index j may take) to only those that satisfy the safety/ correctness condition of Expression 3, thereby ensuring that the choice of implementations in this i'th stage will not lead to an unsafe state.

For reasons that will become evident in Section 3.3, we concurrently also build a table I_W of the same dimensions as the table W, with entry $I_W[i, \boldsymbol{v}]$ storing the index j for which the RHS in the third equation in Recurrence 4 is minimized (i.e., choice of implementations for the i'th stage that minimizes the response time while guaranteeing a cumulative value $\boldsymbol{v}$ over the stages $i, \ldots, n-1$).

3.2.2.2 Optimizing for the typical case

The approach is very similar to the one shown in Section 3.2.2.1 above. Here we build a table T; for any $i \in \{0, i, \ldots, n\}$ and any $\boldsymbol{v} \in \{0, 1, 2, \ldots, \mathcal{V}\}$, $T[i, \boldsymbol{v}]$ denotes the smallest delay bound that we can guarantee under all typical circumstances, if we are to obtain a target value of at least $\boldsymbol{v}$ over the stages $i, (i+1), \ldots, (n-1)$. We have

$$\begin{aligned} T[n, 0] &= 0 \\ T[n, \boldsymbol{v}] &= \infty \text{ for all } \boldsymbol{v} > 0 \\ T[i, \boldsymbol{v}] &= \min_{\{j \mid v_{ij} + \Delta_{i+1} \geq \boldsymbol{v}\}} \left\{ c_{ij} + T[i+1, \boldsymbol{v} - v_{ij}^T] \right\} \end{aligned} \tag{5}$$

The only difference from Expression 4, the recurrence relation for worst-case analysis, arises in the third (recursive) equation, in specifying the value that remains to be obtained in the stages $(i+1), \ldots, (n-1)$. While in Expression 4 this is the worst-case value guaranteed by $\mathcal{I}_{i,j}$ (i.e., $v_{i,j}$), it is instead the lowest value guaranteed under *typical* circumstances (i.e., $v_{i,j}^T$) in Expression 5.

Analogous to the table I_W in Section 3.2.2.1 above, we also build a table I_T that stores, in $I_T[i, \boldsymbol{v}]$, the index j for which the RHS in the third equation in Recurrence 5 is minimized.

Implementation Details, and Running time

Tables W and I_W for worst-case analysis, and tables T and I_T for typical-case analysis, are constructed similarly in a bottom-up fashion. The procedure for computing the T and I_T tables is depicted in pseudo-code form in Figure 4 (the pseudo-code for computing the W and I_W tables is analogous and hence omitted). The bottom-most row of the table ($i == n$) is not explicitly stored, but rather implicitly assumed initialized to all zeros. Row i is filled in once

all the rows $(i+1), \ldots, n-1$ have been filled. To compute each entry in row i of the table, we may need to examine each of the m alternative implementations $\mathcal{I}_{i,0}, \mathcal{I}_{i,1}, \ldots, \mathcal{I}_{i,m-1}$ (this is done in the **for** loop of lines 4–8); hence computing each entry in the table takes $\Theta(m)$ time. Since there are $(n+1) \times (\mathcal{V}+1)$ entries in the table, the entire table can be fill in with an overall running time of $\Theta(mn\mathcal{V})$.

(Note that in order to compute the entries in the table it is necessary that the values $\Delta_0, \Delta_1, \ldots, \Delta_n$ be known. Hence the pseudo-code of Figure 3 must be executed prior to calling this procedure. It was previously argued, in Section 3.2.1, that the pseudo-code of Figure 3 has a running time $\Theta(nm)$; the overall computational complexity is therefore dominated by the cost of computing the tables W and/ or T, and remains $\Theta(mn\mathcal{V})$.)

3.3 Run-time algorithms

We now explain how the tables W or T, constructed as described in Section 3.2 above, are used during run-time. Section 3.3.1 discusses how the tables W and I_W may be used to optimize for performance in the worst case, and Section 3.3.2 discusses how the tables T and I_T may be used to optimize for performance in the typical case (while guaranteeing correctness in all cases, typical or not).

3.3.1 Optimizing for the worst case

Let us first suppose that we are optimizing for the worst case, and consider some actual execution of the system during run-time. A **static** schedule would pre-select an implementation for each stage prior to commencing execution of the first stage, and not change this decision during run-time, regardless of the actual values obtained at each stage. Such a schedule is readily determined using the tables W and I_W that were computed as discussed above, in Section 3.2:

1 Let $j = I_W[0, \mathcal{V}]$
2 choose implementation $\mathcal{I}_{0,j}$ for stage 0
3 $sumVal = v_{0,j}$
4 **for** $i = 1$ **to** $n-1$
5 Let $j = I_W[i, \mathcal{V} - sumVal]$
6 choose implementation $\mathcal{I}_{i,j}$ for stage j
7 $sumVal = sumVal + v_{i,j}$

Such a static schedule has the advantage of not requiring run-time monitoring: since the scheduling choices are not changed regardless of how much value is actually obtained at each stage, there is really no reason to determine this via run-time monitoring.

An **adaptive** schedule, in contrast, would, at each stage, only select which implementation of that stage should be executed in order to optimize for worst-case running time subject to the constraint that the remaining value needed for assuring safety be obtainable. For stage 0, the choice is identical to the one selected by the static schedule above. However, the choice of implementations for future stages would depend upon the actual values obtained, as determined by run-time monitoring, by the chosen implementations of already-executed stages:

1 Let $j = I_W[0, \mathcal{V}]$
2 execute implementation $\mathcal{I}_{0,j}$ for stage 0. Let $\widehat{v_0}$ denote the value so obtained
3 $sumVal = \widehat{v_0}$
4 **for** $i = 1$ **to** $n - 1$
5 Let $j = I_W[i, \mathcal{V} - sumVal]$
6 execute implementation $\mathcal{I}_{i,j}$ for stage j. Let $\widehat{v_i}$ denote the value so obtained.
7 $sumVal = sumVal + \widehat{v_i}$

Since $\widehat{v_i}$ may be larger than v_{ij} for each i (recall that the v_{ij} values are assumed to be very conservative *lower* bounds), an adaptive schedule will, in general, tend to be different from the static one; additionally, different executions of the instance may result in different schedules (since the $\widehat{v_i}$ values may be different on different executions).

3.3.2 Optimizing for the typical case

It is not generally possible to synthesize a completely static schedule that optimizes for the typical case, since run-time monitoring may reveal that typical-case assumptions are violated upon executing some stage and when that happens, it is essential that correctness be preserved at the cost of abandoning the pre-computed schedule. A form of "***semi-static***" schedule could be conceived of, which would

(a) Precompute a static schedule assuming typical behavior, in a manner analogous to the manner in which the static schedule optimizing for the worst case was synthesized.
(b) Corresponding to each i, $0 \leq i < n - 1$, pre-synthesize an alternative schedule for the stages $i + 1, \cdots, n - 1$ in the event that typical conditions are observed to have been violated while executing the selected implementation for the i'th stage. This alternative schedule would only seek to assure correctness without regard for performance; hence, it may simply execute at each stage the implementation characterized by largest worst-case value (the v_{ij} parameter).

However since such a semi-static schedule does not obviate the need for run-time monitoring (unlike static schedules optimizing for the worst case), there is no significant advantage to not going fully adaptive, as is done by the following run-time algorithm:

1 Let $j = I_T[0, \mathcal{V}]$
2 execute implementation $\mathcal{I}_{0,j}$ for stage 0. Let $\widehat{v_0}$ denote the value so obtained
3 $sumVal = \widehat{v_0}$
4 **for** $i = 1$ **to** $n - 1$
5 Let $j = I_T[i, \mathcal{V} - sumVal]$
6 execute implementation $\mathcal{I}_{i,j}$ for stage j. Let $\widehat{v_i}$ denote the value so obtained.
7 $sumVal = sumVal + \widehat{v_i}$

3.4 Characterization of Optimality

As stated in Sections 1 and 2, our objective has been to optimize for performance, measured as the duration of the computation, whilst assuring correctness: guaranteeing that the target value $\mathcal{V}$ will be obtained under all circumstances. We had pointed out that (at least) two distinct interpretations of the optimization objective seem reasonable – one optimizing for performance under all possible conditions and the other, optimizing for performance under all typical conditions– and had advocated in favor of adopting the latter interpretation (while accepting that the former may be more appropriate for certain systems). In Sections 3.2-3.3

above we defined a pair of pseudo-polynomial time algorithms that compute optimal solutions according to these two different objectives; in this section we will precisely state in what manner these solutions are optimal.

As stated in Section 3.3, the pair of tables W and I_W, and the pair of tables T and I_T, could each be used in two different ways at runtime. Tables W and I_W could be used to construct a **static** schedule or an **adaptive** scheduling strategy, both of which are optimized for the worst case; Tables T and I_T could be used to construct either a **semi-static** or an **adaptive** scheduling strategy, both optimized for the typical case. We now state the performance guarantees that are made by each of these choices of scheduling strategies.

1. **Static schedule, using Tables W and I_W**: Our run-time schedule guarantees a correct schedule[5] with response time no greater than $W(0, \mathcal{V})$; additionally, no non-clairvoyant scheduling strategy can guarantee a correct schedule with smaller response time.
2. **Semi-static schedule, using Tables T and I_T**: Our run-time scheduling strategy guarantees a correct schedule with response time no greater than $W(0, \mathcal{V})$ for all executions in which no implementation's behavior violates the typical-case assumptions (i.e., each implementation $\mathcal{I}_{ij}$ chosen for execution returns a value $\geq v_{ij}^T$); additionally, no non-clairvoyant scheduling strategy can guarantee a correct schedule with smaller response time for all typical executions.
3. **Adaptive scheduling, using Tables W and I_W**: Our run-time schedule guarantees a correct schedule with response time no greater than $W(0, \mathcal{V})$.
   Additionally, suppose that a value $\widehat{\mathcal{V}}$ has been obtained over the first $(i-1)$ stages during some execution of the system. Our scheduling strategy guarantees a correct schedule with remaining response time no larger than $W(i, \mathcal{V} - \widehat{\mathcal{V}})$, and no non-clairvoyant correct scheduling strategy can guarantee a smaller remaining response time.
4. **Adaptive scheduling, using Tables T and I_T**: Our run-time scheduling strategy guarantees a correct schedule with response time no greater than $W(0, \mathcal{V})$ for all executions satisfying the typical-case assumptions.
   Additionally, suppose that a value $\widehat{\mathcal{V}}$ has been obtained over the first $(i-1)$ stages during some execution of the system. Our scheduling strategy guarantees a correct schedule with remaining response time no larger than $T(i, \mathcal{V} - \widehat{\mathcal{V}})$ for all executions that satisfy the typical-case assumptions for the remaining stages, and no non-clairvoyant correct scheduling strategy can guarantee a smaller remaining response time under all typical-case executions of the remaining stages.

4 Heuristic Approaches: A Brief Exploration

We have seen (Section 3.1) that for the kinds of computations studied in this paper the problem of scheduling in a manner that optimizes for performance while assuring correctness is NP-hard; however we were able to develop (Sections 3.2–3.3) pseudo-polynomial time algorithms for solving this problem optimally. Our solutions additionally have the desirable feature that all pseudo-polynomial time processing is performed prior to run-time: during run-time once the actual computation has commenced, the schedule, defined as the choice of implementation for each stage, may be determined via rapid (constant-time) table lookup operations.

[5] Recall that a correct schedule guarantees to obtain a cumulative value $\mathcal{V}$ over all the stages of the computation.

What are our alternatives if one is does not wish to do pseudo-polynomial processing even before run-time? In that case a number of greedy heuristics suggest themselves; in this section, we briefly examine a few such heuristics:

G1: At each stage i, choose the safe implementation (recall that Section 3.2.1 explains how such safe implementations are identified) with the largest guaranteed value (i.e., largest value for v_{ij}). Stop[6] upon having obtained the target value $\mathcal{V}$.

G2: At each stage i, choose the safe implementation with the smallest execution time (i.e., smallest value for c_{ij}). As above, stop once the target value $\mathcal{V}$ has been obtained.

G3: At each stage i, choose the safe implementation with the largest typical value density per unit of execution time (i.e., largest value of the ratio v_{ij}^T/c_{ij}). Once again, stop once the target value $\mathcal{V}$ has been obtained.

For each of these heuristics, it is relatively straightforward to identify circumstances (i.e., generate problem instances) upon which the heuristic performed arbitrarily poorly. However, it often appears to be the case that for randomly-generated instances that are generated according to some particular stochastic methodology at least one out of these three greedy heuristics (or out of a few other equally obvious ones, not listed above) does not suffer too much of a performance degradation in comparison to our pseudo-polynomial optimal algorithm under typical-case conditions.[7] This leads us to conjecture that *for any particular application system for which it is reasonable to assume that the implementation choices are characterized by parameters drawn from some underlying probabilistic distribution, it may suffice to test the system's behavior in the typical case when scheduled using a number of such greedy heuristics and simply choose the heuristic with which it performs the best*: it is likely (although of course not guaranteed) that this heuristic will provide performance that is close to that provided by the optimal pseudo-polynomial time algorithmic approach described in Sections 3.2–3.3.

In the remainder of this section we will provide some evidence to back up our conjecture that some greedy heuristic seems to perform reasonably well on problem instances whose parameters are drawn from some underlying probabilistic distributions. We do so via simulation experiments upon randomly-generated workloads of a particular kind, as detailed below. We will see that our observations in these experiments reveal that for this kind of workload, the greedy heuristic **G3** offers performance that (with rare exceptions) tends to lie within 10% or so of the performance offered by the optimal algorithm; hence if we have good reason to believe that the actual workloads we will encounter are appropriately modeled by the probabilistic model we are using to generate our workload and we are willing to pay a $\approx 10\%$ performance penalty, then it is reasonable to use heuristic G3 rather than the pseudo-polynomial time optimal algorithm. (We emphasize that it remains *safe* to use the heuristic even if the probabilistic model turns out to be incorrect: assuming that the worst-case characterizations of each implementation $\mathcal{I}_{ij}$ – its v_{ij} parameter – is indeed a true lower bound on the actual value that will be obtained upon executing $\mathcal{I}_{ij}$, correctness remains assured even if performance is far from optimali due to a mismatch between the assumed probabilistic model and reality.)

---

[6] This assumes an interpretation of our computational model that if the required $\mathcal{V}$ units of computation are obtained upon completing i stages of the computation, then the remaining stages do not need to be executed. An alternative interpretation is to require that all stages be executed: under this interpretation, we subsequently execute that implementation of each remaining stage that has the smallest execution duration (c_{ij} parameter). Our broad conclusions remain unchanged for both interpretations.

[7] Of particular note, the performance of the heuristic tends to be *superior to* that of the optimal (pseudo-polynomial time) algorithm optimizing for the worst-case. While this is not particularly surprising – we are evaluating under typical-case conditions while that algorithm optimizes for the worst case – we felt that it merits a note.

§1: Workload Generation. We consider workloads in which there is a choice of exactly two implementations per stage (in the notation introduced in Section 2, $m = 2$) – one corresponding to "traditional" deterministic implementation and the other, to a learning-enabled component (LEC) that guarantees a smaller value but is likely to provide a greater value under typical conditions, than the traditional implementation. The parameters characterizing the workload generator are as follows:

(1) The number of stages n ($n \in \mathbb{N}$).

(2) Two positive integer parameters C_L and C_H ($C_L \leq C_H$).
The c_{ij} parameters of the implementations are drawn uniformly at random as integers from the range $[C_L, C_H]$.

(3) Two positive integer parameters V_L and V_H ($V_L \leq V_H$).
The v_{ij} parameters of the implementations are drawn uniformly at random as integers from the range $[V_L, V_H]$. For each stage, two numbers are so drawn: the smaller becomes the v_{ij} for the LEC and the larger, for the traditional (deterministic) component.

(4) Two positive integer parameters VT_L and VT_H ($\mathrm{VT}_L \leq \mathrm{VT}_H$).
- The v_{ij}^T parameters of the LEC implementations are obtained by multiplying their v_{ij} parameters by an integer drawn uniformly at random from the range $[\mathrm{VT}_L, \mathrm{VT}_H]$.
- The v_{ij}^T parameters of the traditional (deterministic) components are set equal to their v_{ij} values.

(5) A real number *Slack* ($0 < \mathit{Slack} \leq 1$).
This is used to assign a value to the target parameter $\mathcal{V}$; this assigned value is *Slack* times the largest value that can be guaranteed across all the stages.

§2: The Experiments Performed. An assignment of values to each of the eight parameters $\langle n, C_L, C_H, V_L, V_H, \mathrm{VT}_L, \mathrm{VT}_H, \mathit{Slack}\rangle$ constitutes a single configuration for our experiment. The precise configurations we examined are enumerated below; for each such examined configuration, we generated one hundred instances and determined the durations of the schedules that would be generated by the following five algorithm/ heuristics:

1. tOpt: The adaptive algorithm that optimizes for performance in the typical case.
2. wOpt: The adaptive algorithm that optimizes for performance in the worst case.
3. G1: The adaptive greedy heuristic in which we choose, for each state, that safe implementation that guarantees the largest value.
4. G2: The adaptive greedy heuristic in which we choose, for each state, that safe implementation that has the smallest execution duration.
5. G3: The adaptive greedy heuristic in which we choose, for each state, that safe implementation that returns, in the typical case, the largest value per unit of execution time.

in a run-time scenario in which each chosen implementation $\mathcal{I}_{ij}$ executes for a duration exactly equal to c_{ij} and returns a value exactly equal to v_{ij}^T. Note that tOpt, by design, generates the optimal schedule under these circumstances; hence for each configuration our experimental setup reports the average, over all hundred instances, of duration taken by each of the five algorithms normalized by the duration taken by tOpt. Here is one example of the kind of data reported by our experimental setup:

```
n= 5; CL=10; CH=50; VL=1; VH=5; VTL=2; VTH=10; Slack=0.5

W-OPT: 1.34
G1:    4.44
G2:    6.26
G3:    1.14
```

Table 1 Reported average of the durations of the schedules generated by Algorithm wOpt and the heuristics G1, G2, and G3, relative to the duration of the schedule generated by Algorithm tOpt, across 100 instances generated according to the configuration $\langle n = 5, C_L = 10, C_H = 50, V_L = 1, V_H = 5, \mathrm{VT}_L = 2, \mathrm{VT}_H = 10, \mathit{Slack}\rangle$, for the different values of *Slack* listed in the first column.

Slack	wOpt	G1	G2	G3
0.2	1.17	2.24	6.41	1.04
0.3	1.39	3.10	6.31	1.04
0.4	1.59	3.73	6.37	1.05
0.5	1.34	4.44	6.26	1.14
0.6	1.36	4.17	5.34	1.08
0.7	1.48	4.47	4.79	1.04
0.8	1.78	5.08	5.42	1.02
0.9	1.72	5.14	4.98	1.06

These data report that, across one hundred instances generated with the configuration

$$\langle n = 5, C_L = 10, C_H = 50, V_L = 1, V_H = 5, \mathrm{VT}_L = 2, \mathrm{VT}_H = 10, \mathit{Slack} = 0.5\rangle\,,$$

the duration taken by Algorithm wOpt was 1.34 times that taken by tOpt on average, the duration taken by heuristic G1 was 4.44 times that taken by tOpt on average, the duration taken by heuristic G2 was 6.26 times that taken by tOpt on average, and the duration taken by heuristic G3 was 1.14 times that taken by tOpt on average.

§3: Observations. We examined a wide range of configurations, and the effect of changing one or a few parameters (while keeping the other unchanged) in order to explore the performance of the two optimal algorithms and the three heuristics upon workloads that are compliant with the modeling assumptions stated in §1 above. A typical set of observations is reported in Table 1: this reports on the outcomes upon eight closely-related configurations in which only the *Slack* parameter is changed. As can be seen from the numbers in Table 1, heuristic G3 consistently offers performance close to the optimal algorithm, with the poorest relative performance for $\mathit{Slack} = 0.5$ where it is a mere 14% poorer than the optimal. The other heuristics, in contrast, may be sub-optimal by factors exceeding five and six; even the pseudo-polynomial time optimal algorithm that optimizes for the worst case is off by as much as 78% (for $\mathit{Slack} = 0.8$).

Another set of results is depicted in Table 2 – here, the parameter *Slack* is fixed (at 0.8), while the number of stages n is varied. It may be seen that once again it is heuristic G3 that performs well with a maximum performance degradation of 10% (for $n = 9$), while wOpt may be off by up to 74% and the heuristics G1 and G2 by more than a factor of six. It is also noteworthy that the performance of heuristic G3 does not appear to drop off steeply as the number of stages increases: this is significant since the setup of our experiments means that the target value $\mathcal{V}$ tends to increase linearly with the number of stages, meaning that the running time of the optimal algorithms, which are polynomial in the value of $\mathcal{V}$, will increase with increasing n. Hence, for large values of n (multistage computations with a large number of stages) the run-time savings of using the efficient greedy heuristic G3 may be particularly worth our while.

Another useful lesson learned from our experimental evaluation relates to the performance of wOpt relative to tOpt under typical-case conditions. The numbers in the columns labeled wOpt in Tables 1 and 2 indicate that one pays a significant performance penalty

Table 2 Reported average of the durations of the schedules generated by Algorithm wOpt and the heuristics G1, G2, and G3, relative to the duration of the schedule generated by Algorithm tOpt, across 100 instances generated according to the configuration $\langle n, C_L = 10, C_H = 50, V_L = 1, V_H = 5, \mathrm{VT}_L = 2, \mathrm{VT}_H = 10, \mathit{Slack} = 0.8\rangle$, for the different values of n listed in the first column.

n	wOpt	G1	G2	G3
5	1.53	4.56	4.89	1.01
6	1.51	4.84	5.21	1.04
7	1.74	5.41	5.77	1.03
8	1.41	6.15	6.46	1.07
9	1.38	6.01	6.40	1.10
10	1.50	5.99	6.52	1.02
15	1.50	5.13	5.67	1.04
20	1.54	5.44	6.11	1.05
30	1.52	5.55	6.31	1.05
40	1.42	5.37	6.14	1.10
50	1.49	5.47	6.26	1.08
100	1.51	5.64	6.49	1.08

(at least 17%, more typically in the $35\% - 60\%$ range and as high as 78%) under typical conditions by optimizing for the worst case; this provides further support for our advocating for explicitly optimizing for the typical case (while assuring safety in all cases, typical or not).

Limitations of our evaluation. Our experimental evaluation has been brief and somewhat cursory: we are not suggesting that these experiments are exhaustive enough, or have examined enough combinations of parameter values, to be able to draw authoritative or general conclusions. All that can be said is that these particular experiments do provide some support for our conjecture that perhaps efficient greedy heuristics are adequate for finding near-optimal solutions for workloads that are characterized by parameters drawn from certain well-behaved distributions. Hence our experimental evaluation is, at best, very preliminary: we plan to conduct a far more thorough evaluation in the future, considering a wider range of probabilistic models that are inspired by the observed characteristics of actual LECs and basing inferences upon more rigorous statistical methods (confidence intervals; refutable null hypotheses; etc.) than just the simple means (and standard deviations – not reported here) that we have collected and looked at thus far.

5 Generalizations

In this paper we have restricted our consideration of LEC-enabled computations (i.e., computations incorporating Learning-Enabled Components) to those that can be modeled as multi-stage computations with a choice of implementations per stage, and with each implementation $\mathcal{I}_{ij}$ characterized by a single worst-case execution time parameter c_{ij} and a pair of parameters v_{ij} and v_{ij}^T denoting the minimum value the implementation is guaranteed to yield under all and typical conditions respectively. We believe that analysis of this simple basic model is a necessary first step towards enabling the safe use of LECs in safety-critical systems. We now discuss several extensions to this basic model that would generalize it significantly in several directions and extend its applicability; we have been working on extending our analysis techniques to deal with these generalizations.

Typical and worst-case characterization of running time. The model used in this paper characterizes the running time of an implementation $\mathcal{I}_{ij}$ with a single worst-case execution time parameter c_{ij}. There is no reason why worst-case execution time could also not be determined by both worst-case and typical-case analysis – indeed, the idea of typical-case analysis was first proposed [9] in the context of estimating worst-case execution time. Although many currently-popular LECs tend to be relatively deterministic with regard to their timing behavior (see footnote 2), we could in principle have a model in which each implementation $\mathcal{I}_{ij}$ is characterized by both a c_{ij} and a c_{ij}^T denoting execution-time bounds under worst-case and typical-case circumstances. All the techniques developed in earlier sections of this paper generalized in a straight-forward manner to this situation.

State. It is possible for some *state* to be generated by each stage of a multi-stage computation and communicated to subsequent stages, with the behavior of these subsequent stages dependent upon the communicated state. For instance, the typical value that is obtained by an implementation at a particular stage may be dependent upon the particular computational operations performed by the implementations that were chosen for execution at previous stages. Consider for example a stage of an image progressing algorithm tasked with determining how many people there are in an image. The next stage may consist of classifiers, some of which are sensitive to this number. Knowing the value achieved at the previous stage is one method of capturing influence, but in general it is likely that further state information will be required.

The introduction of value-influencing state does not effect the framework developed in this paper. We retain the notions of worst-case value and duration, and hence retain the same definition of feasibility. However, the optimization problem becomes more difficult if there is a significant quantity of state with this influencing role; there may be more typical values to accommodate.

Modes. Some implementations of LECs may have multiple modes in which they are capable of operating: they offer a number of "*(value, computation-time)*" profiles, that are mutually incomparable. If the number of such modes is small then this is essentially equivalent to having more actual implementations (that happen to share the same worst-case behavior). However if the number of modes is high, or any one of a continuum of profiles is possible (as is the case with some anytime algorithms), then it is not immediately evident whether our proposed algorithms would scale appropriately with the number of modes that need to be considered. As future work we will attempt to classify the problem space into domains that are amenable to optimal solutions and those that will need to fall back on the use of heuristics.

6 Conclusions

Learning-Enabled Components (LECs) based upon deep learning and similar AI-based principles are poised to play a very significant role in safety-critical autonomous CPS's; it is therefore highly desirable that the safety-critical systems research community come up with techniques that enable the analysis of such systems to both assure safety (which is essential) and optimize performance (which, for cost and related reasons, is highly desirable). This paper reports on some of our ongoing efforts in this direction. Building off recent work on *typical-case analysis* pioneered in [9] and continued in [4, 2, 5], we have argued that safety-critical systems whose run-time behavior incorporates a great deal of uncertainty should be

designed to optimize for performance in the typical case (while guaranteeing safety in all cases, typical or not). We have further refined and expanded on a formal model that we had first proposed in [3], for representing the functional as well as the timing properties of some kinds of LECs that exhibit such uncertainties in a quantitative manner. We have formulated the problem of synthesizing computations that can be modeled as chains of functional blocks using LECs and that need to achieve a minimum cumulative value to assure safety, and for which performance is quantified by the total duration of the computation, as an optimization problem. We have developed an optimal algorithm for solving this problem – i.e., scheduling the computation in a manner that is safe under all circumstances and guarantees optimal performance (in our case, the duration taken to complete the computation) under all typical circumstances. We have also proposed three greedy heuristics that are sub-optimal but can be implemented to execute very efficiently with polynomial running time. We have compared our algorithm with these heuristics (and another algorithm – one that optimizes for performance in the worst, rather than typical, case) via simulation experiments upon synthetically generated workloads. As ongoing and future work we are evaluating, and will continue to evaluate, specific LECs (such as ones based on deep learning) to determine whether they are amenable to representation using our model and if not, how our model may be generalized to accommodate them (see, e.g., the discussion in Section 5 that has come out of our efforts in this direction).

References

1 Assuring autonomy international programme. https://www.york.ac.uk/assuring-autonomy/. Accessed: 2020-01-17.

2 Kunal Agrawal and Sanjoy Baruah. Adaptive real-time routing in polynomial time. In *Real-Time Systems Symposium (RTSS), 2019 IEEE*, December 2019.

3 Kunal Agrawal, Sanjoy Baruah, Alan Burns, and Abhishek Singh. Minimizing execution duration in the presence of learning-enabled components. In *Proceedings of the Second International Workshop on Autonomous Systems Design (ASD 2020)*, 2020.

4 Sanjoy Baruah. Rapid routing with guaranteed delay bounds. In *Real-Time Systems Symposium (RTSS), 2018 IEEE*, December 2018.

5 Sanjoy Baruah and Nathan Fisher. Choosing preemption points to minimize typical running times. In *Proceedings of the Twenty-Fourth International Conference on Real-Time and Network Systems*, RTNS '19, New York, NY, USA, 2019. ACM.

6 J. Lee, A. Prajogi, E. Rafalovsky, and P. Sarathy. Assuring behavior of autonomous UxV systems. In *S5: The Air Force Research Laboratory (AFRL) Safe and Secure Systems and Software Symposium*, July 2016.

7 Robert M. Nauss. The 0-1 knapsack problem with multiple choice constraints. *European Journal of Operational Research*, 2(2):125–131, 1978. doi:10.1016/0377-2217(78)90108-X.

8 Dr. Sandeep Neema. Assurance for Autonomous Systems is Hard. https://www.darpa.mil/attachments/AssuredAutonomyProposersDay_ProgramBrief.pdf. Accessed: 2019-03-07.

9 Sophie Quinton, Matthias Hanke, and Rolf Ernst. Formal analysis of sporadic overload in real-time systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, pages 515–520, San Jose, CA, USA, 2012. EDA Consortium. URL: http://dl.acm.org/citation.cfm?id=2492708.2492836.

10 John A. Stankovic and Krithi Ramamritham. What is predictability for real-time systems? *Real-Time Syst.*, 2(4):247–254, October 1990. doi:10.1007/BF01995673.

11 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 on Embedded Computing Systems*, 7(3):36:1–36:53, May 2008.

Attack Detection Through Monitoring of Timing Deviations in Embedded Real-Time Systems

Nicolas Bellec
University of Rennes, Inria, CNRS, IRISA, France
nicolas.bellec@irisa.fr

Simon Rokicki
University of Rennes, Inria, CNRS, IRISA, France
simon.rokicki@irisa.fr

Isabelle Puaut
University of Rennes, Inria, CNRS, IRISA, France
isabelle.puaut@irisa.fr

Abstract

Real-time embedded systems (RTES) are required to interact more and more with their environment, thereby increasing their attack surface. Recent security breaches on car brakes and other critical components have already proven the feasibility of attacks on RTES. Such attacks may change the control-flow of the programs, which may lead to violations of the system's timing constraints.

In this paper, we present a technique to detect attacks in RTES based on timing information. Our technique, designed for single-core processors, is based on a monitor implemented in hardware to preserve the predictability of instrumented programs. The monitor uses timing information (Worst-Case Execution Time - WCET) of code regions to detect attacks. The proposed technique guarantees that attacks that delay the run-time of any region beyond its WCET are detected. Since the number of regions in programs impacts the memory resources consumed by the hardware monitor, our method includes a region selection algorithm that limits the amount of memory consumed by the monitor. An implementation of the hardware monitor and its simulation demonstrates the practicality of our approach. In particular, an experimental study evaluates the attack detection latency.

2012 ACM Subject Classification Computer systems organization → Embedded hardware; Security and privacy → Embedded systems security

Keywords and phrases Real-time systems, security, attack detection, control flow hijacking, WCET estimation, hardware monitoring

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.8

Acknowledgements We would like to thanks Steven Derrien for the discussions that lead to this research and Stefanos Skalistis for his insight. We also warmly thank AbsInt for providing the aiT WCET estimator and modifying it for meeting our needs for region selection.

1 Introduction

Real-Time Embedded Systems (RTES) are becoming more and more present in our lives (IoT devices, embedded processors in cars, among others). Real-time constraints for these systems need to be validated through estimation of Worst-Case Execution Time (WCET) of their tasks [24] and schedulability analysis techniques.

Recent attacks on RTES [14] [7] [15] have shown attackers' increased attention to these systems. In particular, the predictability of RTES helps the attacker figure out the timing to strike, making it easier to mount attacks. Improving the security of RTES is a challenging task. Indeed, the deployed techniques (Address Space Randomization, Stack Smashing Protection, Position Independent Executables to cite only a few of them [18,19]) must protect

the system against attacks, while maintaining the system's predictability, and with limited overhead. However, the need for predictability of RTES is also a strength to protect the system against attacks. Information on the system behavior (typically WCET) is available offline, and can thus be used to detect anomalous behaviors resulting from attacks.

In this work, we propose to use the inherent predictability of RTES to protect them against control-flow hijacking attacks for single-core architectures. The new defense mechanism exploits fine-grained partial WCET estimations of the application to detect attacks. A custom hardware monitor is in charge of observing the execution time of code regions and raising the alarm if the region's execution time is larger than its analyzed WCET, determined offline. The hardware monitor has a local memory containing the list of regions to monitor, together with their associated WCET. It then observes the execution at every clock cycle, by reading the current cycle counter and the program counter (PC) to detect execution times of code regions that are higher than the region's WCET. The hardware monitor proceeds by snooping the PC and cycle counter, and this does not require any modification of the application under monitoring. The proposed technique can thus be used on legacy code.

The attacks detected by our technique are all those that divert a region of code from its original control flow for a duration larger than its WCET, whatever the source of the attack. In particular, *software attacks* caused by buffer overflows are supported[1]. The proposed technique also detects *fault attacks*, that inject faults into the system via physical access to the device, using optical or electromagnetic perturbations [22]. On the other hand, attacks that bypass checks in the code are not detected because they do not cause the observed run time of a code region to exceed the region's WCET. Similarly, attacks that are fast enough are not detected. The guarantee provided by the proposed method is the detection of all attacks that increases the timing of a region beyond its WCET, whatever their source.

The selection of the regions to be monitored has an impact on the latency of attack detection: the smaller the WCET of the monitored regions, the faster the attack detection. On the other hand, monitoring more regions also increases the memory required for the hardware monitor. Since minimizing resource consumption is essential in any system, the proposed technique comes with a region selection algorithm that selects the best regions to monitor under resource constraints for the hardware monitor. The algorithm selects regions such that all code is covered (to detect attacks whatever their location) and provides a guarantee on the latency of attack detection. In our experiments, the algorithm reaches the optimal attack detection latency by selecting 47% of all the existing regions, on average.

Compared to related work on using timing information for detecting attacks (e.g. [27]), our technique does not impact the system predictability, since the system under monitoring is not modified. In addition, a technique is proposed to automate the selection of monitored regions. Compared to techniques using watchdogs to protect the control flow and memory accesses of programs [13] the technique we propose does not need any program instrumentation. A deeper comparison with related work is proposed in Section 7.

In summary the contributions of this paper are threefold:

- First, we present an algorithm, executed at compile-time, to select a set of regions whose timing will be monitored at run-time. Together, the regions cover the entire code, such that attacks can be detected whatever their location. The algorithm provides guarantees on the attack detection latency (largest WCET of the monitored regions), and operates

[1] One may consider that buffer overflows cannot occur in systems using static WCET analysis, because static WCET estimation tools model the whole system memory and thus detect possible buffer overflows. However, static WCET estimation tools also accept user-provided annotations such as loop bounds. Such incorrect annotations can lead to vulnerabilities remaining undetected by the estimation tool.

under hardware constraints for the monitor, in particular limited memory for region storage. The algorithm is proven to provide the optimal attack detection latency in the absence of hardware constraints.
- Second, we contribute to a hardware monitoring co-processor, that uses WCET information on code regions to detect attacks. The practicality of the proposed hardware monitor is demonstrated using a simulation of the co-processor running along with a Leon 3 processor.
- Third, we provide an extensive evaluation of the proposed technique (compile-time selection of regions and hardware monitor) on a set of benchmarks. We evaluate the impact of memory constraints on the guaranteed attack detection latency and evaluate the actual detection latency and hardware cost of the monitor.

The rest of this paper is organized as follows. Section 2 gives an overview of the proposed time-based program monitoring technique. Section 3 details the algorithm we propose to select the regions to monitor, under resource constraints. Section 4 then gives implementation details. Experimental results are provided in Section 5. We discuss the scope and limitations of the method in Section 6. Related work is described in Section 7. Finally we conclude in Section 8.

2 Time-based detection of control flow hijacking attacks

The basic principle of our approach is to detect control-flow hijacking attacks by detecting if the run-time of code regions exceeds their statically estimated WCET. This approach handles attacks that divert the control-flow of the program to subvert a task (which could then be used to launch further attacks [3]). The approach is implemented using a dedicated hardware mechanism that measures the number of cycles spent in code regions and raises an alarm when the statically estimated WCET of the region is exceeded, meaning that an attack is under progress.

Selecting all regions in a given binary program not only consumes monitor memory but it is also not always beneficial. For example, selecting a very short region (in terms of WCET) does little for the protection if there exists a longer selected region, the maximum attack detection latency then being the duration of the longer region. Consequently, our approach comes with a compile-time selection of regions that are useful to reduce the attack detection latency. Region selection guarantees that all attacks modifying the control-flow for at least a known *Maximum Attack Window* (MAW) are detected, and operates under memory constraints for the hardware monitor. Thus the MAW is an upper bound of the attack detection latency. Region selection is performed on the program binary with no code modification. The set of regions selected at compile-time has the following properties:
- It completely covers the application binary, every reachable instruction is included in at least one region of the set. This way, attacks can be detected regardless of their location in the code. This ensures that there are no unprotected instructions.
- All regions are perfectly nested: two regions are either completely disjoint or one is fully included inside the other, such that monitored regions can be represented as a tree. This allows a simple yet efficient implementation of the monitor, using a stack representing regions entered and not exited yet.
- Regions only have one entry and one exit edge, as further detailed in Section 3.1.

For each selected region, the hardware monitor measures the execution time spent in the region, ignoring cycles spent in any inner selected regions, and compares it against the worst-case possible duration. The WCET of a region excluding inner selected regions is

called *Maximal Inner Duration* (MID) of the region. The MID of each region is computed at compile-time. The *Maximum Attack Window* (MAW) is the maximal MID of the set of monitored regions. An illustration of the MID of regions is depicted in Figure 1 for a very simple example of two regions R_1 and R_0, with R_1 nested in R_0. Plain lines at the bottom of the figure represent the MID of the two regions. Dotted lines at the top of the figure represent the possible execution times measured at run-time, an execution time higher than the MID reflecting the occurrence of an attack.

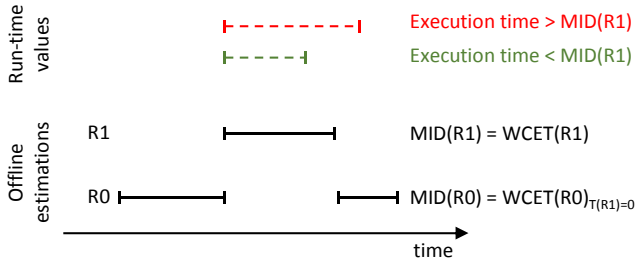


Figure 1 Maximal Inner Duration (MID) of nested regions.

Selecting the regions to monitor is a critical part of the proposed method, as it impacts the maximum attack window. The region selection algorithm we propose for solving this problem is presented in Section 3.

At run time, a dedicated hardware monitor measures the execution time of each region. As illustrated in Figure 2, the monitor has access to the program counter (PC) value of the processor at each cycle and uses it to determine whether the execution enters or exits a region. The monitor maintains a *stack* containing all regions currently active (entered and not yet exited) and the value of their cycle counters. The monitor only updates the cycle counter of the most nested region, currently under execution (stack top). The monitor uses two scratchpad memories: the *Stack Memory* (called SM in the following) is used to track active regions, and the *Region Memory* (called RM in the following) to store the tree of monitored regions.

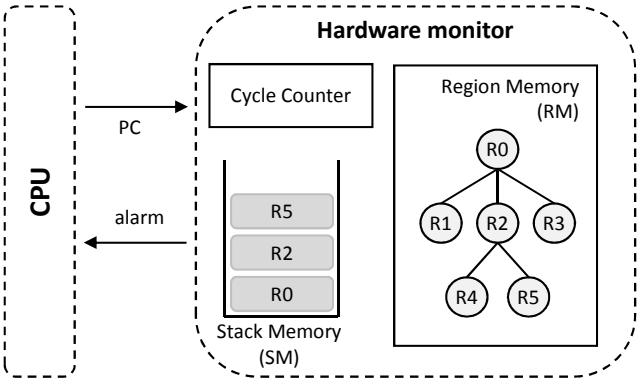


Figure 2 Overview of the monitoring system.

More precisely, at each cycle, the monitor successively tests if the following (non-exclusive) situations occur:

1. If the PC value corresponds to the exit point of the most nested region, this region is removed from the stack. The containing region then becomes the most nested region, and the cycle counter resumes from the value previously stored in the stack for that region.

2. If the PC value corresponds to the entry point of a new region, included in the most nested region, the counter value of the most nested region is saved on the stack, and the monitor starts profiling the new region, with a counter starting at zero.
3. In all cases, the monitor increments the cycle counter of the most nested region and compares it with the value of the MID of the region. If it is greater than the MID, an attack is detected and an alarm is raised.

The monitor is implemented in such a way that it is able to analyze a new PC value at each cycle, and thus does not induce any slowdown in the execution of the application.

Due to the mode of operation of the monitor, *false positives* cannot occur: an execution time higher than the MID of a region, provided that WCETs are safely estimated, can only result from a control flow hijacking attack or a fault. On the other hand, attacks may stay undetected if they are fast enough (faster than the guaranteed attack detection latency).

Hardware constraints

Implementing the hardware monitor requires bounding the size of the Stack Memory (SM) and Region Memory (RM). Moreover, we want the monitor to analyze a PC value at every clock cycle. Bounding the resources required for implementing the monitor imposes the following constraints on the tree of selected regions:

- **Limited height** (maximal number of regions possibly active at the same time) to have a bounded size for SM
- **Limited arity.** This constraint allows to bound the number of simultaneous accesses to RM when several sub-regions (children in the region tree) can be entered from the currently active region.

The region selection algorithm presented in Section 3 assumes that the respective sizes of RM and SM, as well as the maximum tree arity are known. Selecting the best values for these parameters is achieved when dimensioning the hardware monitor for a given application. This impact of hardware constraints on attack detection latency is studied in Section 5.1.4.

3 Off-line selection of monitored regions

This section is devoted to a description of the algorithm we propose to select the regions to monitor. First, the properties of monitored regions are defined in Section 3.1. The algorithm, that guarantees a limited attack detection latency under the hardware constraints mentioned in Section 2 as well as a full program coverage is presented in Section 3.2, and an analysis of its complexity is sketched in Section 3.3. Moreover, we prove in Section 3.4 that, in the absence of hardware constraints, the algorithm is optimal (finds the smallest MAW).

3.1 Properties of monitored regions

Monitored regions are Single-Entry Single-Exit (SESE) regions (regions with a single entry edge and a single exit edge). They are extracted at compile time from the Control Flow Graph (CFG) of program binaries. For the scope of this paper, we use *canonical SESE regions* (simply called SESE regions in the following) as defined in [10]. A canonical SESE region (a, b) is by definition a SESE region such that no other SESE regions starting at the edge a (resp. ending at b) is included in (a, b). Canonical SESE regions can be viewed as the smallest SESE with a as entry edge or b as exit edge. By construction, the smallest size of a SESE region is a basic block (sequence of machine instructions with no branch except the last

instruction). SESE regions are proven in [10] to be properly nested (either completely nested or disjoint, at the function level), which allows them to be organized as an inter-procedural region tree. This is illustrated in the left part of Figure 3, where we can see a CFG with the SESE regions surrounding the nodes they contain. Region R_0 is disjoint from R_1. Regions R_3, R_4 and R_5 are nested in R_1. The corresponding tree structure is given in the right part of Figure 3.

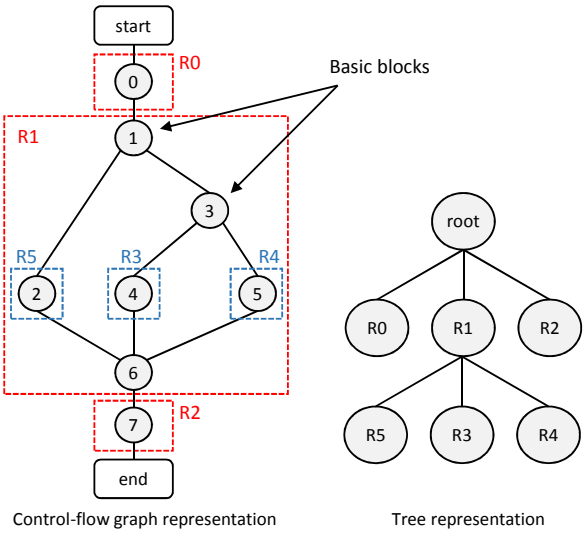


Figure 3 Single Entry Single Exit (SESE) regions viewed in the Control Flow Graph of the application and organized as a tree.

3.2 Region selection algorithm

According to the behavior of the monitor presented in Section 2, and assuming a set of monitored regions S, an attacker has at most the highest Maximal Inner Duration (MID) among monitored regions to perform their attack. As defined previously, the highest MID defines the guaranteed Maximum Attack Window (MAW).

The objective of the region selection problem is to select the monitored regions, that altogether must cover the entire program, such that their maximum MID is as small as possible. The selection operates under resource constraints (memory for stack and storage of region information, maximum number of regions entries/exists to check at every clock cycle). The region selection algorithm is iterative and greedy (never performs backtracking). At each iteration, the algorithm optimizes (reduces) the MID of the selected region R_m having the highest MID. The MID is reduced by selecting one sub-region S of R_m (children of R_m in the inter-procedural region tree), since by definition, other regions have no impact on the MID of R_m. The sub-region S is selected using a heuristic that estimates the new maximum attack window should S be selected. Finally, once S is selected, a new round of MID estimation takes place, with the new set of selected regions.

The algorithm is sketched in Listing 1. Variable *All* contains all regions (regardless of whether they will be selected or not). Variable *Selected* contains the current set of selected regions with their MID. R_m is the region under optimization and S the selected sub-region. Function *MaximalRegion* returns the region having the highest MID. Function *MIDEstimation(r,s)* computes (or re-computes) the MID of region r assuming a null execution

```
1   # Initialisation
2   for r in All:
3     MIDEstimation(r,[]);
4   Rm = MaximalRegion(All);
5   Selected.Append(Rm);
6
7   # Main loop
8   while remainingSpace > 0:
9     S = SelectSubRegion(Rm,Selected);
10    if S == None:
11      break
12    Selected.Append(S);
13
14    for r in All:
15      if r in UpperRegions(S):
16        MIDEstimation(r, Selected);
17
18    Rm = MaximalRegion(Selected);
```

Listing 1 Region selection algorithm

time for the regions in set s. Finally, function *SelectSubRegion(r,s)* implements the heuristic and selects a sub-region of r given the current set of selected regions s. In Listing 1, the MID of all regions is first computed and the region that covers the entire program is selected. The main loop of the algorithm then iteratively optimizes the region R_m with the highest MID.

There are two ways the algorithm can terminate. The first possibility is that there are no more sub-regions to select. This happens if all regions in a program have been selected, or if the region with the highest MID does not contain any unselected sub-region. The second possibility results from an impossibility to find a sub-region that meets the hardware constraints. Infringement of hardware constraints is tested in the main loop through variable *remainingSpace* (test of limited space for region storage) and in function *SelectSubRegion* (test of limited storage for the region stack and limited arity for the region tree).

Function *SelectSubRegion(R,Selected)* implements the heuristic that calculates a *score* for each as-yet-unselected sub-region S of R. The score estimates the new maximum attack window should R be selected, and then selects the region R having the best (here lowest) score. The score is computed as follows:

$$score = max(MID(R, Selected) - T(R, Selected, S), MID(S, Selected))$$

This score represents the best-case evolution of the MAW, i.e., what the new MAW would be if there was no other path than the worst-case path in the region. $T(R, Selected, S)$ represents the contribution of sub-region S to MID(R, *Selected*). The first parameter of the *max* function estimates the new MID of R should its sub-region S be selected. The second term is the MID of sub-region S. The score is the maximum of the two parameters because the MAW is defined as the maximum of the MID of all selected regions. As the score is computed for every sub-regions of R_m, the algorithm likely selects a sub-region that is not a direct child of R_m.

After a sub-region S has been selected by the algorithm, all the MIDs affected by the newly selected region S have to be re-estimated. The affected MID are the ones of the regions that are higher in the inter-procedural region tree and thus include the selected region S.

3.3 Complexity of region selection

We first evaluate the worst-case complexity of the region selection algorithm in terms of the number of operations on the regions. At each iteration, the algorithm selects a sub-region R_m. Such a selection requires verifying the stack and arity constraints for each sub-region whose number is, at worst, the number of regions in the program, n. The stack constraint check consists in finding the longest weighted path in the inter-procedural region tree, whose depth is at worst the number of regions. We can thus infer that the overall worst-case complexity of the stack check is $O(n^2)$. Verifying the arity requires checking, for each region already selected inside the maximal region, whether the newly selected sub-region increases its arity beyond the constraint. The arity of a region can be computed with a slightly modified depth-first search. Thus the worst-case complexity of arity checking is $O(n^3)$. As the number of iterations of the algorithm is, in the worst-case, the number of regions, we obtain a worst-case complexity of $O(n^4)$ in terms of the number of operations on regions.

As presented in section 5, most of the run-time of the region selection algorithm is spent in MID estimation. The complexity of MID estimation is equivalent to the complexity of WCET estimation, and is hidden in the WCET estimation tool. Thus, the most important complexity metric is the worst-case number of MID estimations. This number can be bounded as, in the worst-case, the algorithm evaluates the MID of all regions. Thus, the number of MID estimations cannot exceed n^2. In practice, the actual number is far lower, as the algorithm only has to estimate the MID of the regions higher in the inter-procedural region tree than the newly selected region, as the MID of other regions is not impacted. The observed run-time of region selection is studied in Section 5.1.2.

3.4 Maximum Attack Window optimality: proof sketch

The region selection algorithm is an iterative greedy algorithm, which selects a set of regions to monitor under hardware constraints (see Section 2) and aims at minimizing the MAW. The proposed algorithm satisfies the following optimality property: if hardware constraints are ignored, our algorithm selects the regions that minimize the MAW (as if *all* regions were selected). Proving this property relies on two lemmas.

▶ **Lemma 1.** *When a new region is added to the set of selected regions, the MAW cannot increase.*

▶ **Lemma 2.** *If adding a new region reduces the MAW of an application, then it is a sub-region of the selected region having the maximal MID.*

Only a proof sketch is given for Lemmas 1 and 2 for space considerations. Lemma 1 can be proven by noting that if a region R has a MID m, then any path within this region has a MID lower than or equal to m. Thus selecting a sub-region S inside R only lowers or maintains the time on those paths, and thus the MID with S selected, and consequently the MAW, does not increase. Lemma 2 is proven using the idea that, as regions are properly nested, if a selected region S impacts the MID of another region R, then S must be nested into R to decrease the time of some instructions in at least one path of the affected region.

Assuming Lemmas 1 and 2 hold, let us show that the algorithm selects the regions that minimize the MAW when the algorithm stops. According to the code of the algorithm, there are three possible causes for the algorithm to end:

1. Hardware constraints are not met (lines 8 and 9 in the algorithm). This cannot happen here because we ignore hardware constraints.

2. All regions have been selected. Since according to Lemma 1 the MAW never increases when selecting a new region, we have then reached the minimal MAW.
3. The region having the maximal MID has no as-yet-unselected sub-regions. Then no region could reduce the MAW according to Lemma 2.

Consequently, when the algorithm ends, no region could further reduce the MAW. ◀

4 Implementation

4.1 Target processor and compilation toolchain

Without loss of generality, we target in this paper a Leon 3 core, implementing the SPARC V8 instruction set, and in which we have deactivated the instruction and data caches. Programs are compiled using the *gaisler* compiler tool-chain [5]. The Leon 3 processor has *branch delay-slots*: the instruction located immediately after a (conditional or unconditional) branch instruction will always execute, regardless of the outcome of the branch. The processor also includes a branch predictor.

WCET and MID estimations use the commercial WCET estimation tool aiT for Leon 3, version 19.04i [2]. aiT implements value analysis using abstract interpretation, to determine the range of values in registers, automatic detection of loop bounds [6], as well as static analysis of the processor micro-architecture. This version implements a modification to the aiT original behavior to prevent the timing analysis of a part of the code (the time is then given by the user); the modification maintains the value analysis but discards the micro-architectural analysis.

4.2 Implementation of region selection

Region selection has been implemented in Python3. In a first step, the Control-Flow Graph is constructed, using the disassembled code of the provided binary. SESE regions are then extracted from the CFG, using the algorithm presented in [10]. Finally, we perform the region selection as previously presented in Listing 1. MID estimations are performed using aiT, by computing the WCET of the targeted regions and using aiT annotations to set the WCET of selected sub-regions to *zero*.

The algorithm computing the SESE regions provides the smallest SESE regions, with the minimum region size of one basic-block. Some of these regions are consecutive and thus can be merged into a larger SESE region, called *domain*. Including domains in the program region tree allows to perform the selection of larger regions with less resources. For example, instead of selecting 4 regions composing a loop body, it is possible to select the domain representing the fusion of these 4 regions, reducing memory requirements and mitigating arity constraints. It is still possible to select one of the 4 regions inside the domain to further reduce the MAW.

4.3 Implementation of the hardware monitor

As described in Section 2, the monitor needs to analyze a new PC value at each cycle (that may be the same as the one observed at the previous cycle). Based on this observed value, it decides whether to exit the active region and/or to enter a new one. In the following, we describe the design of the monitor capable of handling 4096 regions, with a maximal arity of 8 and a stack size of 64.

The program region tree is stored in the Region Memory (RM) using an array with 64 bits per region. Those 64 bits are divided in the following way:

- 24 bits for the address of the entry point (target of the region's entry edge),
- 24 bits for the address of the exit point (target of region's exit edge),
- 12 bits for encoding the index of the first child in the array (index start with 0), and
- 4 bits for encoding the number of children.

Moreover, we ensure that all children of a given region are stored consecutively. Hence the index of the first child of a region gives access to all children. Figure 4 is an example of representations. We can see that all children of region R_1 are stored consecutively, starting at index 3. In order to save memory, the MID of every region is not stored in RM. Instead, we use the MAW (highest MID) for all regions. This means that attacks are detected only when the counter of the current region reaches the MAW and not earlier (for regions whose MID is lower than the MAW). On the other hand, not saving the MID for every region saves memory and allows us to monitor more regions. RM is divided into eight banks, a region of index i being stored in bank $i \ mod \ 8$. Consequently, all children of a region can be accessed within a single clock cycle.

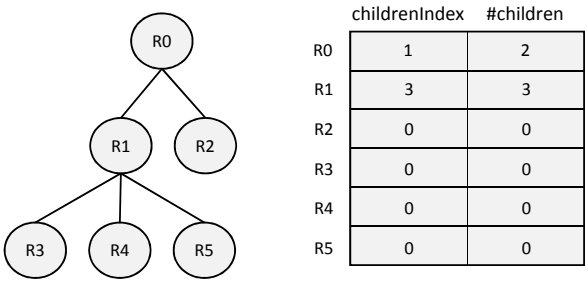


Figure 4 Example of the tree encoding used in the monitor. For the sake of simplicity, addresses of entry and exit points of each regions are not depicted in the figure.

The Stack Memory (SM) is encoded using 64-bit registers. In each register, 24 bits are used to store the counter value. The remaining bits contain a copy of the information on the region (24 bits for region exit, 12 bits for first children, 4 bits for the number of children) such that the monitor can test region exit without any access to RM.

The hardware monitor is described at the C level and synthesized into hardware description language using Mentor Catapult HLS (v10.3a). The system has been simulated at the C level and is currently being implemented on an Altera FPGA.

5 Experimental evaluation

In this section, we present the experimental study we conducted to evaluate our approach. This study is separated into two parts.

- Section 5.1 is dedicated to an evaluation of the region selection algorithm. We first study the effectiveness of region selection and the limits in the way we build regions. Then we measure the execution time of the selection algorithm, ignoring all hardware constraints. Finally, we evaluate the impact of hardware constraints on the quality of the results (MAW).
- Section 5.2 evaluates the hardware monitor. We first measure the latency of the attack detection. Then we compare our hardware-based approach against existing software-based methods. Finally, we evaluate the silicon area required to implement our approach and compare it with the area of the Leon core.

5.1 Evaluation of the region selection algorithm

5.1.1 Experimental setup

We use the Mälardalen [9] and PolyBench [17] benchmarks to evaluate the region selection algorithm. The Mälardalen benchmarks are composed of multiple programs written in C with many different program structures. In particular, among the benchmarks there are control-oriented programs such as *statemate* and *nsichneu* and more simple computational kernels such as *fft1* or *fir*. The PolyBench benchmarks are computation kernels written in C, composed mostly of loop nests. To evaluate the region selection algorithm on as realistic as possible codes, we ruled out all benchmarks with less than 30 regions in the program structure tree. Furthermore, we eliminated benchmarks that could not be analyzed with only basic loop bound annotations.

We evaluate our technique on the following 24 programs: 16 from the Mälardalen benchmarks: *adpcm*, *cnt*, *compress*, *crc*, *fft1*, *ludcmp*, *matmult*, *minver*, *ndes*, *nsichneu*, *statemate*, *ud*, *edn*, *st*, *lms* and *qsort-exam*, and 8 from the Polybench benchmarks: *gemver*, *3mm*, *ludcmp*, *covariance*, *nussinov*, *adi*, *fdtd-2d*, *heat-3d*. There are two implementations of the *ludcmp* algorithm, one in each benchmark suite. To differentiate them in the next section we add the prefix *poly_* to every program of the PolyBench suite.

5.1.2 Maximum Attack Window (MAW) without hardware constraints

We first estimate the best (i.e. smallest) MAW, obtainable by the selection algorithm when ignoring hardware constraints (size/arity). In Table 1, we compare, for each benchmark, the obtained MAW (obtained by the selection algorithm) with the WCET of the benchmark (equivalent to the MAW at the start of region selection). We also indicate the total number of regions per benchmark and the number of regions selected by the algorithm. The results show that 60% of the benchmarks have a MAW below 820 cycles. On the selected architecture, an instruction takes about 12 cycles (counting the SRAM access time). Thus, for 60% of the benchmarks, the MAW is about 69 instructions.

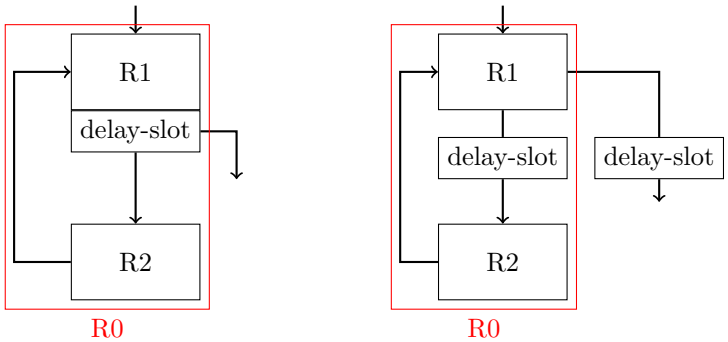
For 14 out of 24 benchmarks, the obtained MAW corresponds to a region containing a single basic block (labeled with a 'b' in the "Limiting reason" column). For these benchmarks, the MAW is between 274 and 5 039 cycles, with 50% of these benchmarks having a MAW below 614 cycles.

For the other 10 benchmarks (labeled with an 's'), higher values of MAW are obtained due regions containing a loop with a delay-slot. *aiT* handles delay-slots as special basic blocks to handle branch prediction. As the impact of a delay-slot on the WCET may differ depending on whether the branch is taken or not taken, *aiT* considers the delay-slot as two basic blocks, one for each outcome. This prevents us from placing the delay-slot instruction as the last instruction of a region. Thus, delay-slot instructions are often placed in an upper region in the region tree. As loops almost always have a header with a delay-slot, the delay-slot is placed in the region representing the loop. As the loop can iterate many times, the cost of the delay slot is multiplied by the loop bound, resulting in a region containing only a single delay-slot instruction but with a high MID. This region can then prevent the improvement of the MAW if it becomes the region with the highest MID.

For example, we sketch the desired behavior in Figure 5a: the delay-slot is part of the region R_1 and thus the delay-slot execution time is counted once in the MID of R_1. Figure 5b depicts the behavior of *aiT*, which duplicates the delay-slot for each branch outcome. It also adds guards on the delay-slot block, which forces us to include its execution time in R_0.

Table 1 Best attainable MAW, obtained when ignoring hardware constraints. For each benchmark, we provide the WCET of the whole application, the MAW obtained, the total number of regions and the number of selected regions. The last column gives the limiting factor for MAW reduction: either the size of the basic block (labeled 'b') or the way delay slots are handled (labeled 's').

Benchmark	WCET	MAW	overall regions	selected regions	Limiting reason
lms	9 404 429	1 609	76	35	s
qsort-exam	220 542	614	45	25	b
edn	1 774 300	3 155	78	32	b
st	3 218 793	8 001	60	18	s
poly_ludcmp	1 300 575 626	961	43	31	s
poly_heat-3d	916 894 881	1 953	35	9	b
poly_gemver	20 264 113	961	33	22	s
poly_nussinov	3 906 470 200	493	39	21	b
poly_3mm	254 765 399	450	44	24	b
poly_adi	552 919 400	1 456	33	18	b
poly_fdtd-2d	289 221 703	1 346	38	19	s
poly_covariance	379 138 134	801	34	21	s
ndes	1 225 593	661	101	38	b
matmult	3 339 921	340	53	24	b
compress	1 758 363	166 358	102	79	s
statemate	144 066	2 970	362	21	b
ludcmp	113 587	421	64	29	b
adpcm	2 116 152	16 001	175	17	s
fft	439 514	1 117	75	17	b
minver	35 614	5 039	62	8	b
crc	764 776	4 355	33	16	s
ud	103 136	398	56	27	b
nsichneu	231 282	21 891	755	753	s
cnt	79 166	274	36	17	b



(a) Desired behavior. **(b)** Actual behavior.

Figure 5 Delay-slot handling example.

As R_0 contains the loop, the MID of R_0 depends on the number of iterations. This situation occurs in the *st* benchmark, which contains a 1 000-iterations loop and a delay-slot whose WCET is estimated at 8 cycles. This results in an indivisible region of 8 001 cycles that blocks the improvement to the MAW.

5.1.3 Run-time of region selection without hardware constraints

The run-time of region selection, when ignoring hardware constraints, is given in Figure 6. Each benchmark is represented by a bar with the total time of MID estimations (*aiT* runtime) as one part (top) of the bar and the run-time of the rest of the region selection algorithm as the other part (bottom). We do not represent *nsichneu* as it would have flatten all the other benchmarks but we give the different times.

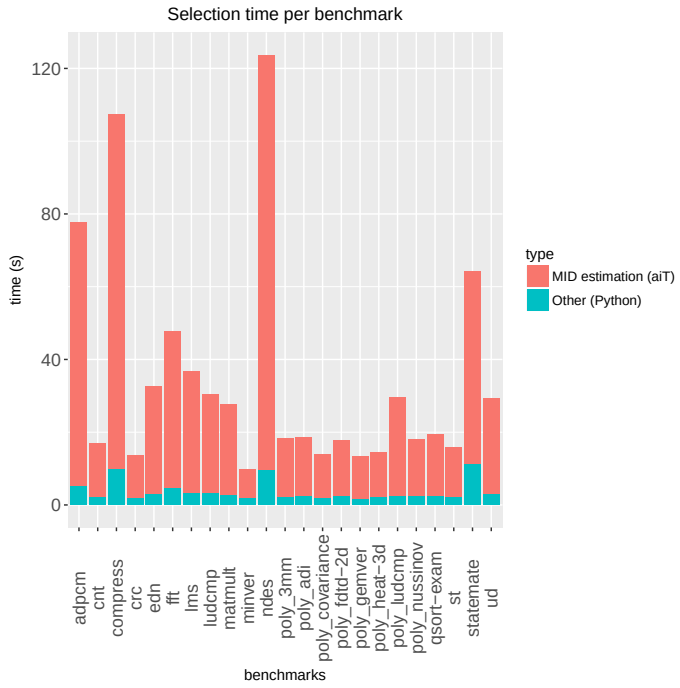


Figure 6 Run-time of the region selection algorithm. *nsichneu* is not presented to be able to avoid flattening the other benchmarks. For *nsichneu*, MID estimations (aiT) = 4 440 s, Other (Python) = 317 s.

These results empirically show that most of the computation time of the selection algorithm comes from the MID estimations by *aiT*. We also observe that the overall execution time remains acceptable for all represented benchmarks. In all benchmarks except *nsichneu*, region selection is performed in less than 200s. *nsichneu* takes more time due to its structure that forces the selection algorithm to select almost all the regions, which requires many MID estimations. Most programs do not have a structure as specific as the one of *nsichneu* and for them, the selection algorithm run-time remains very low.

5.1.4 Impact of hardware constraints

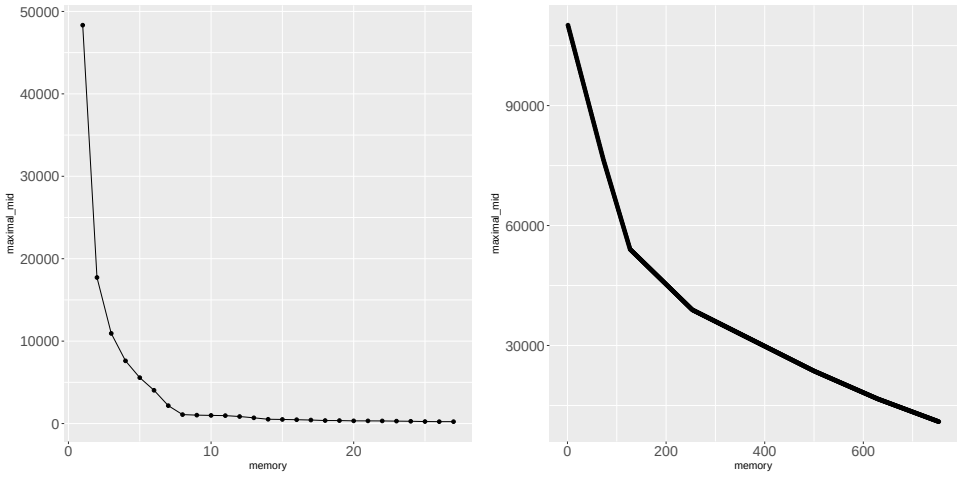
This section studies the impact of hardware constraints on the quality of region selection.

First, we observed that the required stack size in the benchmarks never exceeds 12 and the required size per stack element is 64 bits only. Thus, this factor has little importance on the resources consumed by the monitor. Consequently, we limit our analysis to the impact of limited memory for RM and limited arity for the region tree.

We first evaluate how limited size for RM impacts the obtained MAW. To do so, we extract the MAW for each iteration of the region selection algorithm without constraints. We can then have a representation of the MAW obtained if the selection algorithm had stopped due to limited RM size, without other constraints (stack / arity). The obtained curves depict the obtained MAW as a function of the number of selected regions, with each point being a newly selected region (which is equivalent to an increase of the RM size by 64 bits). Most of the curves have the same shape as *ud*, presented in Figure 7a.

For most benchmarks, the threshold plummets at the beginning of the selection and then decreases slowly or stabilizes, corresponding to two phases. The only benchmark for which the MAW decreases at approximately a constant rate is *nsichneu* as depicted in Figure 7b. This is due to the particular structure of this benchmark, imposing a one-by-one selection of a very small region at each iteration. Interestingly, on this benchmark, the algorithm does select from the most to the less interesting regions, as the slope of the curve decreases with the number of regions selected.

The best attainable MAW is reached when selecting a mean of 47% of the overall regions, with a standard deviation of 20% (noted 47±20%). When selecting the regions, 90% of the reduction is performed by selecting only 24%±19% of the regions selected at the end. The first selected regions are often very interesting to efficiently reduce the MAW. The large standard deviations are mostly due to the disparity in the complexities of the benchmarks.



(a) ud. **(b)** nsichneu.

Figure 7 MAW evolution with the number of selected regions.

The second studied hardware constraint is the arity constraint. To evaluate its influence, we perform the selection in three configurations, all without constraining the size of RM: unbounded arity, arity of 8 and arity of 4. The resulting MAW is given in Table 2.

The results indicate that for an arity of 8, all benchmarks except *nsichneu* have a MAW less than 2 times higher than with no arity constraint. 21 out of the 24 benchmarks have the same MAW as when the arity is unbounded. *nsichneu* remains a special case due to its structure, which makes the selection rely exclusively on the arity to reduce the MAW. We see in that case that the structure of the code has a strong impact on the capacity of our method to protect it.

For an arity of 4, 16 out of 24 benchmarks are not affected by the constraint. Among the affected 8 benchmarks, the increase in the MAW is so low that there is almost no difference in the protection for 3 of them. However, the other benchmarks except *compress*, see their MAW increase by a factor of 6 or more compared to a system with no arity constraint. In these cases, the attack window may be considered too large to efficiently protect the program and a superior arity is then required. *compress* is special as it has a very high MAW to begin with, so even a factor below two makes its MAW skyrocket even more. We conclude that arity has an important impact on the obtained MAW, but it seems that there is no need to increase the arity to more than 8 to have good results for almost all benchmarks.

Table 2 MAW in function of arity for all benchmarks (no limit on RM size).

Benchmark	No bound	4	8	Benchmark	No bound	4	8
adpcm	16 001	16 001	16 001	cnt	274	274	274
compress	166 358	313 124	196 406	crc	4 355	4 355	4 355
edn	3 155	145 340	3 155	fft	1 117	1 117	1 117
lms	1 609	1 609	1 609	ludcmp	421	621	421
matmult	340	340	340	minver	5 039	5 039	5 039
ndes	661	707	661	nsichneu	21 891	227 184	223 200
poly_3mm	450	450	450	poly_adi	1 456	1 456	1 456
poly_covariance	801	801	801	poly_fdtd-2d	1 346	1 346	1 346
poly_gemver	961	961	961	poly_heat-3d	1 953	1 953	1 953
poly_ludcmp	961	961	961	poly_nussinov	493	518	493
qsort-exam	614	4 047	1 262	st	8 001	8 001	8 001
statemate	2 970	3 014	2 970	ud	398	398	398

5.2 Evaluation of the hardware monitor

This section evaluates the hardware monitor, in terms of observed detection latency and implementation overheads.

5.2.1 Observed attack detection latency

The experiments reported in this section have two objectives: to demonstrate that the proposed approach effectively detects control-flow deviations and evaluate the achieved attack detection latency.

The experiments operate on traces obtained using *Modelsim SE-64 10.5a*, that performs a cycle-accurate simulation of the Leon 3 core while it executes code. We post-processed the traces produce by Modelsim to extract the trace of PC values at each cycle and remove the C values corresponding to the instructions canceled after branch misspredictions. The C version of our monitor is configured with the result of our region selection algorithm with stack and memory constraints set at 1024 and arity constraint set at 8 as for our hardware evaluation (see section 5.2.3). It scans the trace to detects when the execution enters or exits. We simulate a scenario where the attacker escapes from the standard execution path from an unknown path and never returns back. This is done by modifying the PC value in the execution trace at a cycle picked randomly. Then we verify that the proposed hardware monitor detects the attack, and we measure the latency between the attack and its detection. For each application, the attack is inserted at 100 000 random points. Due to

experimental limitations, some optimizations on the estimation of the MAW are disabled for this experiment. Consequently, the MAW used here are different from the MAW presented in Table 1. Results of this experiment are presented in Table 3, giving the mean and standard deviation of the attack latency, expressed as a percentage of the MAW.

Table 3 Latency of the detection of an attack using the proposed mechanism. The number of cycles between the attack and its detection are provided as a percentage of the MAW determined statically.

Benchmark	**Mean latency** (% of MAW)	**Standard deviation latency** (% of MAW)
crc	99 %	0.8 %
lms	94 %	5.7 %
minver	68 %	29.8 %
fft	94 %	4.9 %
qsort-exam	97 %	2.7 %

We observe that all attacks are detected by the monitor. The latency is often close to the MAW as the attacks often take place in regions with a MID far lower than the MAW. Consequently, even if the attack is triggered close to the end of the region, the alarm is raised when the counter reaches the MAW. The *minver* application has a different behavior: most of the execution takes place in the region having the limiting MID. Thus, when the attack is triggered at the end of the region, it is detected quickly. We can see in Table 3 that *minver* has a latency of 64% of its maximal MAW.

5.2.2 Evaluation of overheads as compared to software approaches

As we opted for a hardware implementation of the monitor, our approach has no impact on the execution time. In comparison, software-based approaches require to read a cycle counter from memory and to compare the value with the MAW. Depending on the implementation, this software implemented monitoring system could even require a system call, as in the work from Zimmer at al. [27].

In this experiment, we evaluate the overhead of our technique if it were implemented in software. We use the execution traces extracted during the previous experiment to count the number of times a region is entered or exited during the execution. Then, we define three scenarios with different penalties for switching of region (entry / exit), corresponding to different implementation strategies for the monitoring system (e.g. with/without system calls, optimized or not). In the *fast-soft* scenario, monitoring a region requires 10 cycles at each region switch. In the *medium-slow*, it requires 20 cycles and in the *slow-soft* it requires 200 cycles.

Table 4 summarizes the results of this experiment. The first column provides for each application simulated, the number of times the monitor switches from one region to another. The other columns correspond to the performance penalty due to the software implemented monitoring of regions. We provide both the number of cycles spent in monitoring and the overhead relatively to the original execution time of the application. We can see that a software based approach can add a significant overhead in the execution time. Even for the *fast-soft* scenario, the overhead reaches 51% for *qsort-exam*. For *medium-soft* and *slow-soft*, the monitoring overhead is often larger than the original execution time. Of course, this overhead could be reduced by increasing the size of the monitored regions, but this also increases the detection latency. The hardware monitors does not affect the execution time of the application.

Table 4 Comparison between the overheads of the hardware monitor (hardware) and the state of art assumed with a constant overhead at each region switch (entry or exit) in the worst-case scenario for a subset of the benchmarks.

		Overhead (cycle / % of execution time)			
Benchmark	**Region switch**	**0**/switch (hardware)	**10**/switch (fast-soft)	**20**/switch (medium-soft)	**200**/switch (slow-soft)
crc	1 718	0	17 180 19%	34 360 39%	343 600 387%
lms	6 134		61 340 5%	122 680 10%	1 226 800 99%
minver	12		120 1%	240 3%	2 400 29%
fft	1 090		10 900 28%	21 800 56%	218 000 556%
qsort-exam	306		3 060 51%	6 120 103%	61 200 1029%

5.2.3 Hardware overhead of monitor

The last experiments estimate the hardware cost of the monitor. We have synthesized both the Leon core and the monitor, targeting a 28 nm technology from STMicroelectronics and a working frequency of 50 MHz (default frequency of the Leon core). As the cost is dominated by memory, this cost is separated from the other costs when giving the results.

Table 5 Area cost of the Leon core and of the proposed hardware monitor. Results are given for different core configurations, with memory ranging from 64 kB to 512 kB, and a monitor capable of monitoring 1 024 regions (which corresponds to 8 kB).

Leon Memory Size	64 kB	128 kB	256 kB	512 kB
Monitor Size	8 kB (1 024 monitored regions)			
Leon Area (μm^2)	128 197	229 117	430 957	834 637
core only	27 277			
memory only	100 920	201 840	403 680	807 360
Monitor Area (μm^2)	28 598			
monitor only	15 983			
memory only	12 615			
Overhead	22%	12%	7%	3%

Results are reported in Table 5. The first two lines are a summary of the configurations studied. Then, area results for the Leon core and for the proposed monitor (in µm2) are presented. For both of them, we provide the area consumed ignoring the memories and the area of the memories. The last line gives the area overhead due to the use of the proposed approach. We can see that, depending on the core configuration, the overhead of the monitor goes from 22% to 3%. All these results corresponds to a monitor handling up to 1 024 regions. This value was picked knowing that the biggest application we studied requires 80 regions (ignoring *nsichneu* that requires 700 regions). We can estimate that doubling the number of region handled by the monitor would double the area consumed by the memories.

6 Discussion

As any security mechanism, the design we propose has a limited scope that we discuss in this section. Since our work heavily relies on WCET estimation techniques and tools, their limitations impact the quality of our work. In particular, the pessimism of WCET estimates, even on small regions such as basic blocks, induces higher bounds for our detection and thus more time for the attacker to bend the program behavior.

We selected an implementation of the monitor that uses the same bound for all regions, to reduce the hardware overhead of the design. Storing the bound of each region in the memory and checking the elapsed time against this bound would certainly improve the security of the protection, at the expense of a higher hardware overhead to store all these data.

Using debug interfaces already present on the processor is an interesting idea to use this protection with less hardware overhead. For example, we could implement the monitor in software and use another core to run it in order to reduce the intrusiveness). Extension to debug-interface is left to future work. Another way to improve the protection would be to use less pessimistic WCET estimations (e.g. using hybrid methods) but this may provide unsound WCET estimates which would detect attacks where there are not (i.e. false-positive).

A second limitation is the type of attacks that can be detected by our protection. Our protection targets attacks that would modify the control-flow and not return at the right place in time. Our design does not try to prevent attacks that are not based on control-flow hijacking (e.g. data-only attacks [12]).

Finally, our current design does not yet handle multi-tasking operating system and multi-processor. We believe that multi-processors can be easily handled by dispatching the task on the processors off-line and using one monitor per processor. Multi-tasking operating systems on the other hand requires to detect context switches such that it also switches the monitor context. It also requires to partitioning the memory of the monitor to maintain the context of suspended tasks while running another task. To ensure that the right task is being monitored, the protection must also protect the scheduling. Finally, the protection have to protect the operating system itself. This is left to future work.

7 Related work

Many methods exist to protect systems, including real-time systems, against attacks. A first class of techniques is to *prevent* the attacks from happening, for instance by hardening the binary using checksums, or by modifying the memory layout of programs, or by introducing some randomness when generating schedules. Address Space Layout Randomization (ALSR) randomizes the start address of the key memory regions of a process (code, data, stack, libraries) to guard against buffer overflows. The use of ASLR in small embedded systems (for example, on 8 or 16-bits architectures) is less effective than in 32 or 64-bits architectures, as the system does not always contain enough entropy for them to be efficient [18]. In addition, when timing guarantees are required, WCET estimation techniques have to be re-designed for supporting the presence of unknown addresses for memory regions. The research presented in [8] proposes WCET estimation techniques (in particular instruction cache analysis) for different diversification granularities. This work was able to compute a far tighter WCET estimate than the one obtained with *all miss* assumption (i.e., equivalent to a system without a cache). However, a rough analysis of the data provided in this paper shows that the cost of fine grain diversification can be up to a 50% worst-case overhead, which is often too much for industrial purposes [21]. Another level at which diversification can be used in real-time

systems is at the schedule level. In [11], the goal is to prevent targeted side-channel attacks on other tasks by *shuffling* the schedule while conserving the deadlines criteria, either off-line or on-line.

A second class of techniques, complementary to prevention, is to let the attacker perform some actions and detect them before they cause harm to the system. *Detection techniques* use monitoring techniques, implemented either in hardware or in software, to look for the results of the attack, and not the causes. The sensitivity of the monitor is used to trade-off the security the monitor provides against how much it interferes with the system. The technique proposed in this paper is a detection technique.

Detection techniques differ by the class of information they monitor.Yoon et al. [26] monitors system calls at run-time and compares them to a list of system calls under normal execution. Walls et al. [23] presents a software technique that checks the integrity of the control flow of a program by monitoring the target of indirect branches and comparing them to the value for a normal execution. Hardware for control flow integrity verification is presented in [1]. Zimmer et al. [27] monitors execution times of code snippets of different granularities and compares them against estimated WCETs, to detect attacks such as control-flow hijacking. Like [27] we use timing information for attack detection. However, our research differs from [27] in several aspects: (i) we use dedicated hardware instead of software for detection of timing violations, and then have no impact on the predictability of monitored programs; (ii) we propose an algorithm to automate the selection of monitored regions and thus provide guarantees on the attack detection latency. In addition, in comparison to [27], we do not need Best-Case Execution Times (BCET) in our approach, a metric that is not currently supported by state-of-the-art WCET estimation tools.

Watchdogs have been intensively studied to protect against faults in RTES [13] [25]. In [13], the authors present a survey of different techniques to protect the control flow and memory accesses. The control-flow protection uses signatures inserted at the beginning of each protected block. These signatures are read by the watchdog which can ensure that the block corresponds to the signature and that the received signatures follow a correct order. In [25], the authors use the WCET of a block as a signature to ensure the correct timing behavior of the program. However, it does not handle computed branches. All these techniques require an instrumentation of the program that slows down the program while our solution can be applied without modification of the program. The second point is that these techniques are not focused on security. A fault happening naturally has a very low probability of modifying the signature or modifying the instructions while maintaining a correct signature. On the other hand, a well-crafted attack could subvert the system by modifying the time allocated for a block and transmitted to the watchdog.

A crucial characteristic of a monitor – that impacts its security, responsiveness and invasiveness – is if the monitor is made on *dedicated hardware* or in *software* by the system. Hardware-based monitors have better responsiveness as they can monitor the system faster than software can. As they use dedicated hardware, this kind of monitor uses less system resources, thus reducing the invasiveness of the monitor and reducing its impact on the determinism. Finally, the security is reinforced because hardware is dedicated to the monitoring and can thus be isolated from the system as well as protect the system itself. However, these advantages come with the cost of developing dedicated hardware for each type of required monitor [20] [16] or using a dedicated co-processor [4]. The monitor proposed in this paper is fully implemented in hardware, and is shown not to impact the predictability of software. Protection of the monitor itself against attacks, although not addressed in the current status of our work, should be facilitated by the limited and well-delimited amount of information used for the monitor operation.

8 Conclusion and future work

We have presented in this paper a technique to detect attacks in real-time systems based on WCET of code regions. A monitor, implemented in hardware, tests if a code region exceeds its WCET a the result of an attack, and raises an alarm should this happen. Experimental results show that the Maximum Attack Window guaranteed by our approach is as short as a basic block (tens of instructions for most codes).

Two factors limit the MAW guaranteed by our approach. The first one is the presence of delay-slots in the targeted architecture. Delay-slots, as detailed in Section 5, artificially enlarge the MAW, because they are accounted for in the enclosing selected region. This issue will not appear on architectures without delay-slots, but unfortunately would need a deep change in aiT to have a more precise estimation of the MAW. The other factor that prevents an extremely small MAW is the presence of long basic blocks in some applications. Addressing this issue would require to split long basic blocks in several regions, which would add complexity to the region selection algorithm. A promising direction would be to subtly change the algorithm as follows: in case the MAW is the duration of one basic block the basic block is split in two equal-length (in terms of number of instructions) regions.

Another way to improve the efficiency of attack detection would be to use *Best Case Execution Times* (BCET) in addition to WCETs. We did not further follow this direction for technical reasons, because state-of-the-art WCET estimation tools are strictly oriented towards worst-case performance and do not provide BCET values.

For the scope of this paper, we addressed the monitoring of a single task on a mono-processor system. Extending this work for multi-processors and multi-tasks operating systems is left for future work.

As for the hardware implementation of the monitor, the monitor is currently described as C code and synthesized into VHDL using using the High Level Synthesis (HLS) tool Catapult. Functional correction has been demonstrated through simulations. Ongoing work concerns the integration of the monitor on an FPGA board embedding the Leon 3 core. We also plan as future work to protect the monitor itself, in particular its memory, against fault attacks, using redundancy techniques. The monitor will then be tested against real attacks, including fault attacks, and the observed attack detection latencies will be observed and compared to the guaranteed latency.

References

1 F. A. T. Abad, J. V. D. Woude, Y. Lu, S. Bak, M. Caccamo, L. Sha, R. Mancuso, and S. Mohan. On-chip control flow integrity check for real time embedded systems. In *2013 IEEE 1st International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pages 26–31, August 2013. doi:10.1109/CPSNA.2013.6614242.

2 AbsInt GmbH. ait worst-case execution time estimation tool. https://www.absint.com/ait/. Last accessed: 2020/01/22.

3 Chien-Ying Chen, Sibin Mohan, Rodolfo Pellizzoni, Rakesh B. Bobba, and Negar Kiyavash. A Novel Side-Channel in Real-Time Schedulers. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 90–102, April 2019. ISSN: 1545-3421. doi:10.1109/RTAS.2019.00016.

4 Ronny Chevalier, Maugan Villatel, David Plaquin, and Guillaume Hiet. Co-processor-based behavior monitoring: Application to the detection of attacks against the system management mode. In *Proceedings of the 33rd Annual Computer Security Applications Conference, Orlando, FL, USA, December 4-8, 2017*, pages 399–411, 2017. doi:10.1145/3134600.3134622.

5 Cobham Gaisler. Compiler toolchain for the leon processor. `https://www.gaisler.com/`. Last accessed: 2020/01/22.

6 Christoph Cullmann and Florian Martin. Data-Flow Based Detection of Loop Bounds. In Christine Rochange, editor, *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*, volume 6 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2007. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/OASIcs.WCET.2007.1193`.

7 N. Falliere, L. O. Murchu, and E. Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 5:6, 2011.

8 Joachim Fellmuth, Thomas Göthel, and Sabine Glesner. Instruction caches in static WCET analysis of artificially diversified software. In *30th Euromicro Conference on Real-Time Systems, ECRTS 2018, July 3-6, 2018, Barcelona, Spain*, pages 21:1–21:23, 2018. `doi:10.4230/LIPIcs.ECRTS.2018.21`.

9 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen wcet benchmarks - past, present and future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, July 2010. URL: `http://www.es.mdh.se/publications/1895-`.

10 Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. *SIGPLAN Not.*, 29(6):171–185, June 1994. `doi:10.1145/773473.178258`.

11 Kristin Krüger, Marcus Völp, and Gerhard Fohler. Vulnerability analysis and mitigation of directed timing inference based attacks on time-triggered systems. In *30th Euromicro Conference on Real-Time Systems, ECRTS 2018, July 3-6, 2018, Barcelona, Spain*, pages 22:1–22:17, 2018. `doi:10.4230/LIPIcs.ECRTS.2018.22`.

12 Tingting Lu and Junfeng Wang. Data-flow bending: On the effectiveness of data-flow integrity. *Computers & Security*, 84:365–375, July 2019. `doi:10.1016/j.cose.2019.04.002`.

13 A. Mahmood and E.J. McCluskey. Concurrent error detection using watchdog processors—a survey. *IEEE Transactions on Computers*, 37(2):160–174, February 1988. `doi:10.1109/12.2145`.

14 Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015.

15 Sen Nie, Ling Liu, and Yuefeng Du. Free-fall: hacking tesla from wireless to can bus. *Briefing, Black Hat USA*, pages 1–16, 2017.

16 C. Pilato, K. Wu, S. Garg, R. Karri, and F. Regazzoni. Tainthls: High-level synthesis for dynamic information flow tracking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2018. `doi:10.1109/TCAD.2018.2834421`.

17 Louis-Noël Pouchet and Tomofumi Yuki. PolyBench/C. URL: `http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/`.

18 Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM. `doi:10.1145/1030083.1030124`.

19 Peter Silberman and Richard Johnson. A comparison of buffer overflow prevention implementations and weaknesses. *IDEFENSE, August*, 2004.

20 C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. Hdfi: Hardware-assisted data-flow isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 1–17, May 2016. `doi:10.1109/SP.2016.9`.

21 Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 48–62, 2013. `doi:10.1109/SP.2013.13`.

22 N. Timmers, A. Spruyt, and M. Witteman. Controlling PC on ARM using fault injection. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 25–35, August 2016. doi:10.1109/FDTC.2016.18.

23 Robert J. Walls, Nicholas F. Brown, Thomas Le Baron, Craig A. Shue, Hamed Okhravi, and Bryan C. Ward. Control-flow integrity for real-time embedded systems. In *31st Euromicro Conference on Real-Time Systems, ECRTS 2019, July 9-12, 2019, Stuttgart, Germany.*, pages 2:1–2:24, 2019. doi:10.4230/LIPIcs.ECRTS.2019.2.

24 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3):36:1–36:53, 2008. doi:10.1145/1347375.1347389.

25 Julian Wolf, Bernhard Fechner, Sascha Uhrig, and Theo Ungerer. Fine-grained timing and control flow error checking for hard real-time task execution. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, pages 257–266, June 2012. ISSN: 2150-3117. doi:10.1109/SIES.2012.6356592.

26 Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Mihai Christodorescu, and Lui Sha. Learning execution contexts from system call distribution for anomaly detection in smart embedded system. In *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation, IoTDI 2017, Pittsburgh, PA, USA, April 18-21, 2017*, pages 191–196, 2017. doi:10.1145/3054977.3054999.

27 Christopher Zimmer, Balasubramanya Bhat, Frank Mueller, and Sibin Mohan. Time-based intrusion detection in cyber-physical systems. In *ACM/IEEE 1st International Conference on Cyber-Physical Systems, ICCPS '10, Stockholm, Sweden, April 12-15, 2010*, pages 109–118, 2010. doi:10.1145/1795194.1795210.

Demystifying the Real-Time Linux Scheduling Latency

Daniel Bristot de Oliveira
Red Hat, Italy
bristot@redhat.com

Daniel Casini
Scuola Superiore Sant'Anna, Italy
daniel.casini@santannapisa.it

Rômulo Silva de Oliveira
Universidade Federal de Santa Catarina, Brazil
romulo.deoliveira@ufsc.br

Tommaso Cucinotta
Scuola Superiore Sant'Anna, Italy
tommaso.cucinotta@santannapisa.it

Abstract

Linux has become a viable operating system for many real-time workloads. However, the black-box approach adopted by `cyclictest`, the tool used to evaluate the main real-time metric of the kernel, the scheduling latency, along with the absence of a theoretically-sound description of the in-kernel behavior, sheds some doubts about Linux meriting the real-time adjective. Aiming at clarifying the PREEMPT_RT Linux scheduling latency, this paper leverages the *Thread Synchronization Model* of Linux to derive a set of properties and rules defining the Linux kernel behavior from a scheduling perspective. These rules are then leveraged to derive a sound bound to the scheduling latency, considering all the sources of delays occurring in all possible sequences of synchronization events in the kernel. This paper also presents a tracing method, efficient in time and memory overheads, to observe the kernel events needed to define the variables used in the analysis. This results in an easy-to-use tool for deriving reliable scheduling latency bounds that can be used in practice. Finally, an experimental analysis compares the `cyclictest` and the proposed tool, showing that the proposed method can find sound bounds faster with acceptable overheads.

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

Keywords and phrases Real-time operating systems, Linux kernel, PREEMPT_RT, Scheduling latency

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.9

Supplementary Material ECRTS 2020 Artifact Evaluation approved artifact available at `https://doi.org/10.4230/DARTS.6.1.3`.
Supplement material and the code of the proposed tool is available at: `https://bristot.me/demystifying-the-real-time-linux-latency/`

Funding This work has been partially supported by CAPES, The Brazilian Agency for Higher Education, project PrInt CAPES-UFSC "Automation 4.0."

Acknowledgements The authors would like to thank Thomas Gleixner, Peter Zijlstra, Steven Rostedt, Arnaldo Carvalho De Melo and Clark Williams for the fruitful discussions about the model, analysis, and tool.

1 Introduction

Real-time Linux has been a recurring topic in both research [5, 6, 30] and industry [10, 11, 12, 21, 39] for more than a decade. Nowadays, Linux has an extensive set of real-time related features, from theoretically-supported schedulers such as SCHED_DEADLINE [27] to the priority inversion control in locking algorithms and a fully-preemptive mode. Regarding the fully-preemptive mode, Linux developers have extensively reworked the Linux kernel to reduce the code sections that could delay the scheduling of the highest-priority thread, leading to the well-known PREEMPT_RT variant. `cyclictest` is the primary tool adopted in the evaluation of the fully-preemptive mode of PREEMPT_RT Linux [8], and it is used to compute the time difference between the expected activation time and the actual start of execution of a high-priority thread running on a CPU. By configuring the measurement thread with the highest priority and running a background taskset to generate disturbance, `cyclictest` is used in practice to measure the *scheduling latency* of each CPU of the system. Maximum observed latency values generally range from a few microseconds on single-CPU systems to 250 microseconds on non-uniform memory access systems [35], which are acceptable values for a vast range of applications with sub-millisecond timing precision requirements. This way, PREEMPT_RT Linux closely fulfills theoretical fully-preemptive system assumptions that consider atomic scheduling operations with negligible overheads.

Despite its practical approach and the contributions to the current state-of-art of real-time Linux, `cyclictest` has some known limitations. The main one arises from the opaque nature of the latency value provided by `cyclictest` [4]. Indeed, it only informs about the latency value, without providing insights on its root causes. The tracing features of the kernel are often applied by developers to help in the investigation. However, the usage of tracing is not enough to resolve the problem: the tracing overhead can easily mask the real sources of latency, and the excessive amount of data often drives the developer to conjunctures that are not the actual cause of the problem. For these reasons, the debug of a latency spike on Linux generally takes a reasonable amount of hours of very specialized resources.

A common approach in the real-time systems theory is the categorization of a system as a set of independent variables and equations that describe its integrated timing behavior. However, the complexity of the execution contexts and fine-grained synchronization of the PREEMPT_RT make application of classical real-time analysis for Linux difficult. Linux kernel complexity is undoubtedly a barrier for both expert operating system developers and real-time systems researchers. The absence of a theoretically-sound definition of the Linux behavior is widely known, and it inhibits the application of the rich arsenal of already existing techniques from the real-time theory. Also, it inhibits the development of theoretically-sound analysis that fits all the peculiarities of the Linux task model [23].

Aware of the situation, researchers and developers have been working together in the creation of models that explain the Linux behavior using a formal notation, abstracting the code complexity [2]. The *Thread Synchronization Model for the fully-preemptive PREEMPT RT Linux Kernel* [14] proposes an automata-based model to explain the synchronization dynamics for the *de facto* standard for real-time Linux. Among other things, the model can be used as an abstraction layer to translate the kernel dynamics as analyzed by real-time Linux kernel developers to the abstractions used in the real-time scheduling theory.

Paper approach and contributions: This paper leverages the *Thread Synchronization Model* [14] of Linux to derive a set of properties and rules defining the Linux kernel behavior from a scheduling perspective. These properties are then leveraged in an analysis that derives

a theoretically-sound bound to the scheduling latency that comprehensively considers the sources of delays, including all possible synchronization flows in the kernel code. The analysis builds upon a set of practically-relevant modeling variables inspired by the foundational principles behind the development of the PREEMPT_RT Linux Kernel. This paper also presents an efficient tracing method to observe the kernel events, which are used to define observed values for the variables used in the analysis, while keeping the runtime overhead and storage space to figures that make its use feasible in practice. The tool also analyzes the trace, serving to distinguish the various sources of the latency. Moreover, by exploring the interference caused by adopting different interrupt characterizations, the tool also derives latency bounds based on real execution traces. Finally, the experimental section compares the results obtained by the `cyclictest` and the proposed tool, showing that the proposed method can find sound bounds faster with acceptable overheads.

2 Background

This section provides background information on the main concepts used in this paper, and discusses related research works.

2.1 Linux Execution Contexts, Scheduling Dynamics, and Tracing

With the PREEMPT_RT patchset, Linux has four different preemption models for kernel space activities. This paper targets the fully-preemptive mode, in which there are three different execution contexts: non-maskable interrupts (NMI), maskable interrupts (IRQs), and threads [20]. Both NMIs and IRQs are asynchronous interrupts, i.e., mechanisms used to deliver events coming either from external hardware or by code running on other CPUs via inter-processor interrupts. The interrupt controller manages interrupts, both queueing and dispatching one NMI per-CPU and multiple IRQs. For each CPU, the NMI is the highest-priority interrupt, so it postpones and preempts IRQs. As a design choice, Linux (in the fully-preemptive mode) handles IRQs with IRQs disabled. Hence an IRQ cannot preempt another IRQ. Threads have no control over the NMI, but they can delay the execution of IRQs by temporarily disabling (masking) them.

Given the potential interference on the execution of threads, one of the design goals of PREEMPT_RT was to reduce the code executing in interrupt context to the bare minimum, by moving most of it to thread context. Despite the existence of different memory contexts in which a regular program can run, like kernel threads, or the process context in the user-space, from the scheduling viewpoint they are all threads. Linux has not one but five schedulers. When invoked, the set of schedulers are queried in a fixed order. The following schedulers are checked: (i) stop-machine, a pseudo-scheduler used to execute kernel facilities, (ii) SCHED_DEADLINE [27], (iii) the fixed-priority real-time scheduler, (iv) the completely fair scheduler (CFS), (v) the IDLE scheduler. Every time the schedulers execute, the highest-priority thread is selected for a context switch. When no ready threads are available, the IDLE scheduler returns the *idle thread*, a particular thread always ready to run. For simplicity, we refer hereafter with the term *scheduler* when mentioning the kernel code handling all the scheduling operations related to all five schedulers. The scheduler is called either voluntarily by a thread leaving the processor, or involuntarily, to cause a preemption. Any currently executing thread can postpone the execution of the scheduler while running in the kernel context by either disabling preemption or the IRQs. It is a goal of the fully-preemptive kernel developers to reduce the amount of time in which sections of code can postpone the scheduler execution.

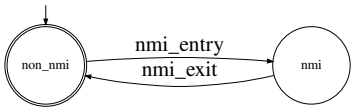


Figure 1 Example of automaton: the NMI generator (Operation O1).

Linux has an advanced set of tracing methods [28]. An essential characteristic of the Linux tracing feature is its efficiency. Currently, the majority of Linux distributions have the tracing features enabled and ready to use. When disabled, the tracing methods have nearly zero overhead, thanks to the extensive usage of runtime code modifications. Currently, there are two main interfaces by which these features can be accessed from user-space: `perf` and `ftrace`. The most common action is to record the occurrence of events into a trace-buffer for post-processing or human interpretation of the events. Furthermore, it is possible to take actions based on events, such as to record a *stacktrace*. Moreover, tools can also hook to the trace methods, processing the events in many different ways, and also be leveraged for other purposes. For example, the Live Patching feature of Linux uses the `function tracer` to hook and deviate the execution of a problematic function to a revised version of the function that fixes a problem [32]. A similar approach was used for runtime verification of the Linux kernel, proving to be an efficient approach [18].

2.2 Automata Models and the PREEMPT_RT Synchronization Model

An automaton is a well-known formal method, utilized in the modeling of *Discrete Event Systems* (DES). The evolution of a DES is described with all possible sequences of events $e_1, e_2, e_3, ...e_n$, with $e_i \in E$, defining the language $\mathcal{L}$ that describes the system.

Automata are characterized by a directed graph or state transition diagram representation. For example, consider the event set $E = \{nmi\_entry, nmi\_exit\}$ and the state transition diagram in Figure 1, where nodes represent system states, labeled arcs represent transitions between states, the arrow points to the initial state, and the nodes with double circles are *marked states*, i.e., safe states of the system.

Formally, a deterministic automaton, denoted by G, is a tuple $G = \{X, E, f, x_0, X_m\}$; where: X is the set of states; E is the set of events; $f : X \times E \to X$ is the transition function, defining the state transition between states from X due to events from E; x_0 is the initial state and $X_m \subseteq X$ is the set of marked states.

An important operation is the *parallel composition* of two or more automata that are combined to compose a single, augmented-state, automaton [7], enabling the model of complex systems using the modular approach. In the modular approach, the system is modeled as a set of two classes of automata: *generators* and *specifications*. Each sub-system has a *generator* of events modeled independently. The synchronization rules of each sub-system are stated as a set of *specification* automata. Each *specification* synchronizes the actions of two or more *generators*. The parallel composition of all the generators and specifications creates the synchronized model [33].

The *Thread Synchronization Model for the PREEMPT_RT Linux Kernel* [14] proposes an automata-based model for describing the behavior of threads in the Linux PREEMPT_RT kernel. The model defines the events and how they influence the timeline of threads' execution, comprising the preemption control, interrupt handlers, interrupt control, scheduling and locking, describing the delays occurred in this operation in the same granularity used by kernel developers. The model is constructed using the modular approach.

2.3 Related Work

Abeni et al. [1] defined a metric similar to `cyclictest`, evaluating various OS latency components of several standard and real-time Linux kernels existing at the time (2002).

Matni and Dagenais [29] proposed the use of automata for analyzing traces generated by the kernel of an operating system. Automata are used to describe patterns of problematic behavior. An off-line analyzer checks for their occurrences. Cerqueira and Brandenburg [9] described experiments with `cyclictest` to evaluate the scheduling latency experienced by real-time tasks under LITMUS$^{\text{RT}}$, vanilla Linux and Linux with the PREEMPT_RT patch. The authors also discussed the advantages and limitations of using `cyclictest` for estimating the capability of a system to provide temporal guarantees. A similar experimental study is presented in [22]. Reghanzani et al. [36] empirically measured the latencies of a real-time Linux system under stress conditions in a mixed-criticality environment.

Herzog et al. [24] presented a tool that systematically measures interrupt latency, at run-time, in the Linux vanilla kernel. No attempt is made to model Linux kernel scheduling. Regnier et al. [37] presented an evaluation of the timeliness of interrupt handling in Linux.

The `ftrace preemptirqsoff` tracer [38] enables the tracing of functions with either preemption or IRQs disabled, trying to capture the longest window. The approach in [38] does not differentiate between interference due to interrupts and the contribution due to different code segments disabling preemption or interrupts. Moreover, by adding tracing of functions it adds overhead to the measurement, thus potentially heavily affecting the result, often mispointing the real source of the latency.

Previous work in [20] and [14] used the timeline and automata-based models, respectively, to describe the Linux kernel behavior regarding threads and interrupt handlers context switches. This work uses the *Thread Synchronization Model* [14] as the description of a single-CPU PREEMPT_RT Linux system configured in the fully-preemptive mode. The advantages of using the model is many-fold: (1) it was developed in collaboration with kernel developers, and widely discussed by us with both practitioners [15, 16] and academia [13, 19]; (2) the model is deterministic, i.e, in a given state a given event can cause only one transition; (3) the model was extensively verified; (4) it abstracts the code complexity by using a set of small automata, each one precisely describing a single behavior of the system. Building upon these approaches, in this work we derive a set of properties and rules defining the Linux kernel scheduling behavior, from the mentioned *Thread Synchronization Model* [14] based on automata. These properties are then used to derive a theoretically-sound bound to the scheduling latency. The proposed bound is based on a formalization of the Linux kernel behavior, where the value of the variables is experimentally measured. To the best of our knowledge, this is the first time that such a detailed analysis of the Linux scheduling behavior is done.

3 System Model

The task set is composed of a single NMI τ^{NMI}, a set $\Gamma^{\text{IRQ}} = \{\tau_1^{\text{IRQ}}, \tau_2^{\text{IRQ}}, \ldots\}$ of maskable interruptions (IRQ for simplicity), and a set of threads $\Gamma^{\text{THD}} = \{\tau_1^{\text{THD}}, \tau_2^{\text{THD}}, \ldots\}$. The NMI, IRQs, and threads are subject to the scheduling hierarchy discussed in Section 2.1, i.e., the NMI has always a higher priority than IRQs, and IRQs always have higher priority than threads. Given a thread τ_i^{THD}, at a given point in time, the set of threads with a higher-priority than τ_i^{THD} is denoted by $\Gamma_{\text{HP}_i}^{\text{THD}}$. Similarly, the set of tasks with priority lower than τ_i^{THD} is denoted by $\Gamma_{\text{LP}_i}^{\text{THD}}$. Although the schedulers might have threads with the same priority in their queues, only one among them will be selected to have its context loaded, and consequently, starting to run. Hence, when scheduling, the schedulers elect a single thread as the highest-priority one.

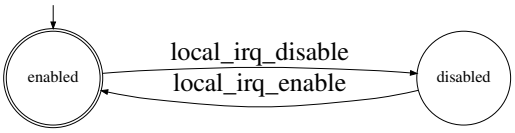


Figure 2 IRQ disabled by software (O2).

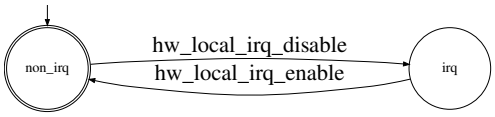


Figure 3 IRQs disabled by hardware (O3).

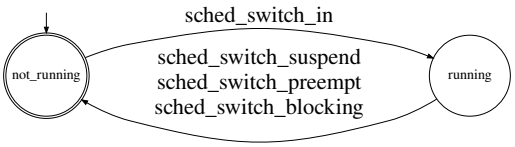


Figure 4 Context switch generator (04).

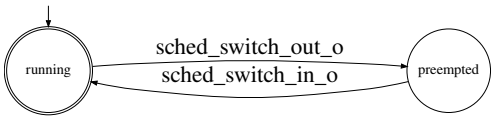


Figure 5 Context switch generator (05).

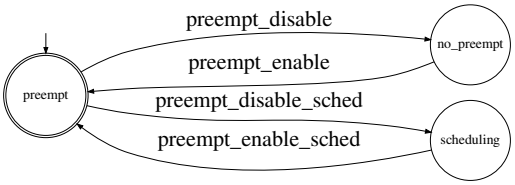


Figure 6 Preempt disable (06).

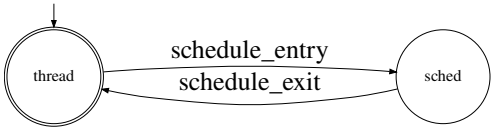


Figure 7 Scheduling context (07).

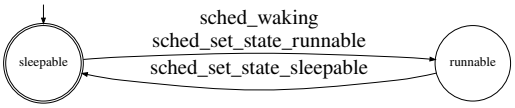


Figure 8 Thread runnable/sleepable (08).

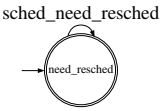


Figure 9 Need re-schedule operation (09).

The system model is formalized using the modular approach, where the *generators* model the independent action of tasks and synchronization primitives, and the *specification* models the synchronized behavior of the system. The next sections explains the *generators* as the basic operations of the system, and the specifications as a set of *rules* that explains the system behavior.

3.1 Basic Operations

This section describes *generators* relevant for the scheduling latency analysis, starting with the interrupt behavior:

- **O1**: The NMI context starts with the entry of the NMI handler (`nmi_entry`), and exits in the return of the handler (`nmi_exit`). This operation is modeled as in Figure 1 (in Section 2).
- **O2**: Linux allows threads to temporarily mask interrupts (`local_irq_disable`), in such a way to avoid access to shared data in an inconsistent state. Threads need to unmask interrupts (`local_irq_enable`) at the end of the critical section, as modeled in Figure 2.
- **O3**: To enforce synchronization, the processor masks interrupts before calling an interrupt handler on it. IRQs stays masked during the entire execution of an interrupt handler (`hw_local_irq_disable`). Interrupts are unmasked after the return of the handler (`hw_local_irq_enable`), as shown in Figure 3. In the model, these events are used to identify the begin and the return of an IRQ execution.

The reference model considers two threads: the thread under analysis and an arbitrary other thread (including the idle thread). The corresponding operations are discussed next.

- **O4**: The thread is not running until its context is loaded in the processor (`sched_switch_in`). The context of a thread can be unloaded by a suspension (`sched_switch_suspend`), blocking (`sched_switch_blocking`), or preemption (`sched_switch_preempt`), as in Figure 4.
- **O5**: The model considers that there is always *another thread* ready to run. The reason is that, on Linux, the *idle state* is implemented as a thread, so at least the *idle thread* is ready to run. The other thread can have its context unloaded (`sched_switch_out_o`) and loaded (`sched_switch_in_o`) in the processor, as modeled in Figure 5.
- **O6**: The preemption is enabled by default. Although the same *function* is used to disable preemption, the model distinguishes the different reasons to disable preemption, as modeled in Figure 6. The preemption can be disabled either to postpone the scheduler execution (`preempt_disable`), or to protect the scheduler execution of a recursive call (`preempt_disable_sched`). Hereafter, the latter mode is referred to as *preemption disabled to call the scheduler* or *preemption disabled to schedule*.
- **O7**: The scheduler starts to run selecting the highest-priority thread (`schedule_entry`, in Figure 7), and returns after scheduling (`schedule_exit`).
- **O8**: Before being able to run, a thread needs to be awakened (`sched_waking`). A thread can set its state to *sleepable* (`sched_set_state_sleepable`) when in need of resources. This operation can be undone if the thread sets its state to runnable again (`sched_set_state_runnable`). The automata that illustrates the interaction among these events is shown in Figure 8.
- **O9**: The set need re-schedule (`sched_need_resched`) notifies that the currently running thread is not the highest-priority anymore, and so the current CPU needs to re-schedule, in such way to select the new highest-priority thread (Figure 9).

3.2 Rules

The *Thread Synchronization Model* [14] includes a set of *specifications* defining the synchronization rules among *generators* (i.e., the basic operations discussed in Section 3.1). Next, we summarize a subset of rules extracted from the automaton, which are relevant to analyze the scheduling latency. Each rule points to a related specification, graphically illustrated with a corresponding figure.

IRQ and NMI rules. First, we start discussing rules related to IRQs and NMI.

- **R1**: There is no specification that blocks the execution of a NMI (**O1**) in the automaton.
- **R2**: There is a set of events that are not allowed in the NMI context (Figure 10), including:
  - **R2a**: set the need resched (**O9**).
  - **R2b**: call the scheduler (**O7**).
  - **R2c**: switch the thread context (**O4** and **O5**)
  - **R2d**: enabling the preemption to schedule (**O6**).
- **R3**: There is a set of events that are not allowed in the IRQ context (Figure 11), including:
  - **R3a**: call the scheduler (**O7**).
  - **R3b**: switch the thread context (**O4** and **O5**).
  - **R3c**: enabling the preemption to schedule (**O6**).
- **R4**: IRQs are disabled either by threads (**O2**) or IRQs (**O3**), as in the model in Figure 12. Thus, it is possible to conclude that:
  - **R4a**: by disabling IRQs, a thread postpones the begin of the IRQ handlers.
  - **R4b**: when IRQs are not disabled by a thread, IRQs can run.

Thread context. Next, synchronization rules related to the thread context are discussed. We start presenting the necessary conditions to call the scheduler (**O7**).

Necessary conditions to call and run the scheduler.

- **R5**: The scheduler is called (and returns) with interrupts enabled (Figure 13).
- **R6**: The scheduler is called (and returns) with preemption disabled to call the scheduler (i.e., via the `preempt_disable_sched` event, Figure 14).
- **R7**: The preemption is never enabled by the scheduling context (Figure 15).

Regarding the context switch (**O4** and **O5**), the following conditions are required.

Necessary conditions for a context switch.

- **R8**: The context switch occurs with interrupts disabled by threads (**O2**) and preemption disabled to schedule (**O6**, Figure 16).
- **R9**: The context switch occurs in the scheduling context (**O7**, Figure 17)

The necessary conditions to set the need resched (**O9**) and to wakeup a thread (**O8**) are the same. They are listed below, and show in Figure 18.

Necessary conditions to set the need resched and to wakeup a thread.

- **R10** Preemption should be disabled, by any mean (**O6**).
- **R11** IRQs should be masked, either to avoid IRQ (**O2**) or to postpone IRQs (**O3**).

Until here, we considered necessary conditions. From now on, we will present sufficient conditions.

Sufficient conditions to call the scheduler and to cause a context switch.

- **R12** Disabling preemption to schedule (**O6**) always causes a call to the scheduler (**O7**, Figure 19).
- **R13** Calling the scheduler (**O7**) always results in a context switch (**O4**,**O5**). Recall that if the system is idle, the idle thread is executed after the context switch. (Figure 20).
- **R14** Setting need resched (**O9**) always results in a context switch (**O4**,**O5**, Figure 21).

4 Demystifying the Real-time Linux Scheduling Latency

4.1 Problem Statement

We start defining the scheduling latency (hereafter only latency) and then we leverage the rules presented in Section 3 and the related automaton model to derive an upper bound reflecting all the peculiarities of Linux. The *latency* experienced by a *thread instance* (also called *job*) may be informally defined as the maximum time elapsed between the instant in which it becomes ready while having the highest-priority among all ready threads, and the time instant in which it is allowed to execute its own code after the context switch has already been performed. By extension, the latency of a thread is defined as reported in Definition 1.

▶ **Definition 1** (Thread Scheduling Latency). *The scheduling latency experienced by an arbitrary thread* $\tau_i^{THD} \in \Gamma^{THD}$ *is the longest time elapsed between the time* A *in which* any *job of* τ_i^{THD} *becomes ready and with the highest priority, and the time* F *in which the scheduler returns and allows* τ_i^{THD} *to execute its code, in any possible schedule in which* τ_i^{THD} *is not preempted by any other thread in the interval* $[A, F]$.

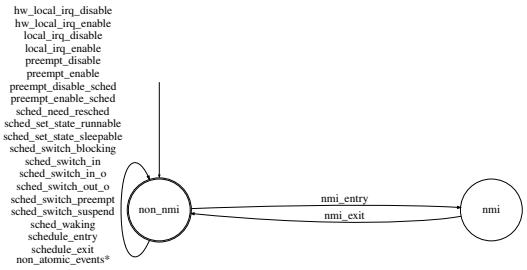


Figure 10 Operations blocked in the NMI context (R2).

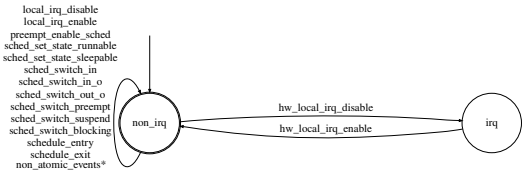


Figure 11 Operations blocked in the IRQ context (R3).

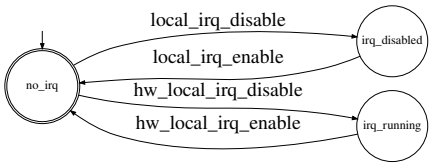


Figure 12 IRQ disabled by thread or IRQs (R4).

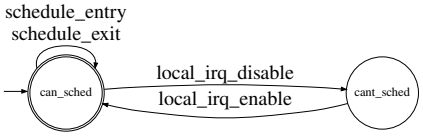


Figure 13 The scheduler is called with interrupts enabled (R5).

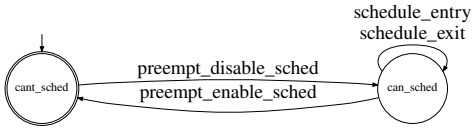


Figure 14 The scheduler is called with preemption disabled to call the scheduler(R6).

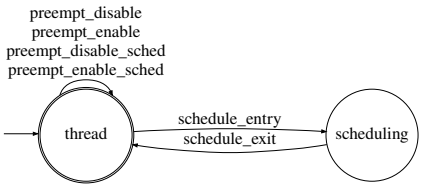


Figure 15 The scheduler context does not enable the preemption (R7).

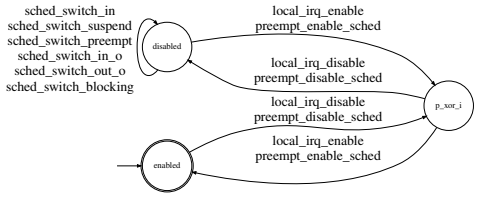


Figure 16 The context switch occurs with interrupts and preempt disabled (R8).

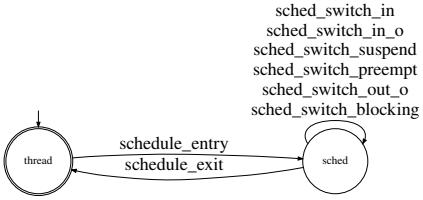


Figure 17 The context switch occurs in the scheduling context (R9).

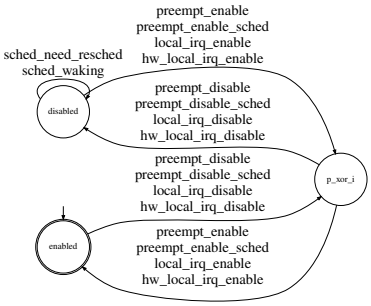


Figure 18 Wakeup and need resched requires IRQs and preemption disabled (R10 and R11).

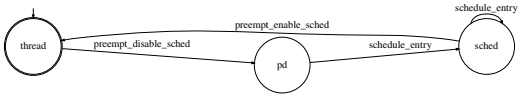


Figure 19 Disabling preemption to schedule always causes a call to the scheduler (R12).

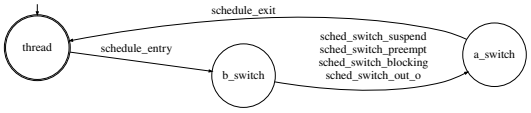


Figure 20 Scheduling always causes context switch (R13).

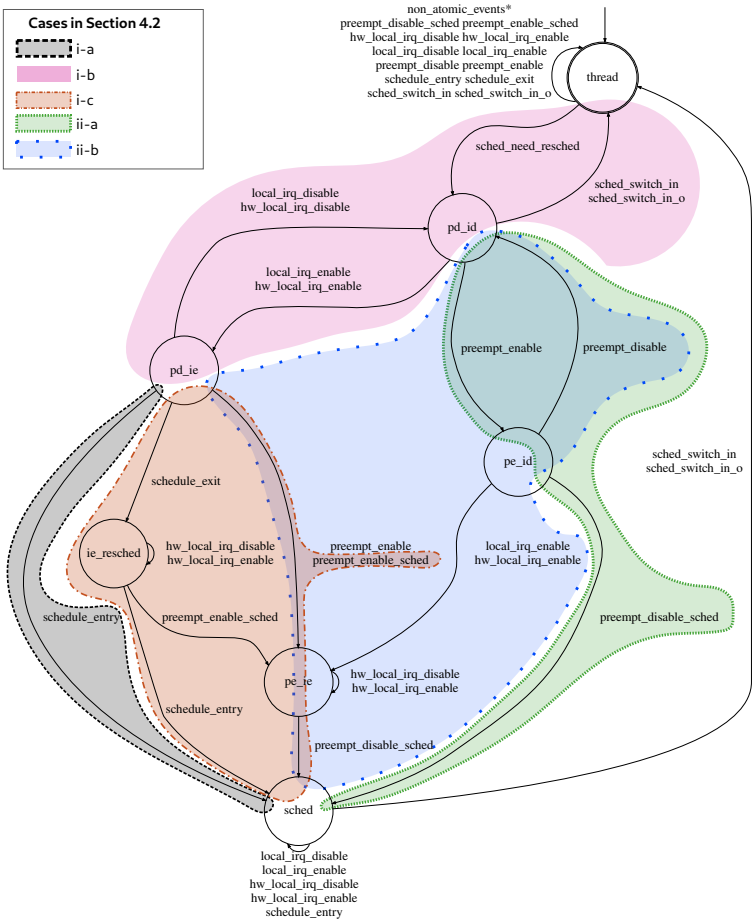


Figure 21 Setting need resched always causes a context switch (R14).

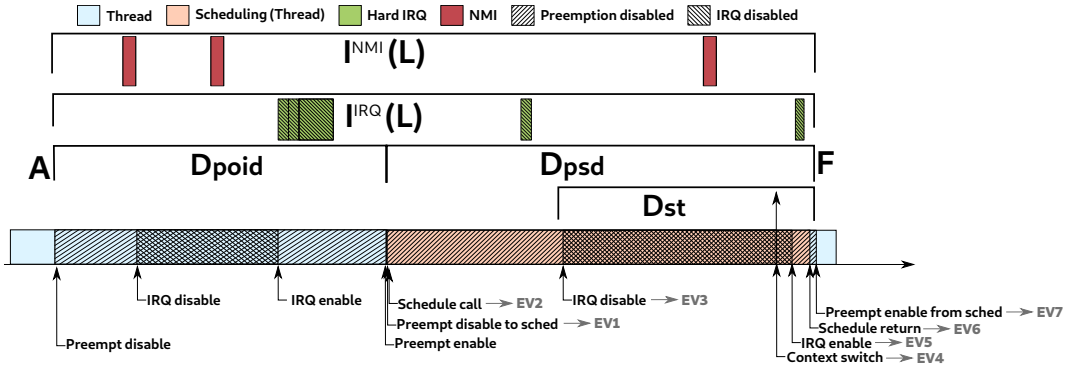


Figure 22 Reference timeline.

For brevity, we refer next to the event that causes any job of τ_i^{THD} becoming ready *and* with the maximum priority as $\texttt{RHP}_i$ event[1]. With Definition 1 in place, this paper aims at computing a theoretically-sound upper bound to the latency experienced by an arbitrary $\tau_i^{\text{THD}} \in \Gamma^{\text{THD}}$ under analysis. To this end, we extract next some formal properties and lemmas from the operations and rules presented in Section 3. We begin determining which types of entities may prolong the latency of τ_i^{THD}.

▶ **Property 1.** *The scheduling latency of an arbitrary thread* $\tau_i^{THD} \in \Gamma^{THD}$ cannot *be prolonged due to high-priority interference from other threads* $\tau_j^{THD} \in \Gamma_{HP_i}^{THD}$.

Proof. By contradiction, assume the property does not hold. Then, due to the priority ordering, it means that either: **(i)** τ_i^{THD} was not the highest-priority thread at the beginning of the interval $[A, F]$ (as defined in Definition 1), or **(ii)** τ_i^{THD} has been preempted in $[A, F]$. Both cases contradict Definition 1, hence the property follows. ◀

Differently, Property 2 shows that the latency of a thread may be prolonged due to *priority-inversion blocking* caused by other threads $\tau_j^{\text{THD}} \in \Gamma_{\text{LP}_i}^{\text{THD}}$ with a lower priority.

▶ **Property 2.** *The latency of an arbitrary thread* $\tau_i^{THD} \in \Gamma^{THD}$ can *be prolonged due to low-priority blocking from other threads* $\tau_j^{THD} \in \Gamma_{LP_i}^{THD}$.

Proof. The property follows by noting that, for example, a low-priority thread may disable the preemption to postpone the scheduler, potentially prolonging the latency of τ_i^{THD}. ◀

With Property 1 and Property 2 in place, we bound the Linux latency as follows, referring to an arbitrary thread τ_i^{THD} under analysis. First, as a consequence of Property 1, only the NMI and IRQs may prolong the latency due to high-priority interference, and such an interference is equal for all threads $\tau_i^{\text{THD}} \in \Gamma^{\text{THD}}$ since NMI and IRQs have higher priorities than threads. We model the interference due to the NMI and IRQs in a time window of length t with the functions $I^{\text{NMI}}(t)$ and $I^{\text{IRQ}}(t)$, respectively. We then show next in Section 5 how to derive such functions. Besides interference, the latency is caused by constant kernel overheads (e.g., due to the execution of the kernel code for performing the context switch) and priority-inversion blocking (see Property 2), which we bound with a term L^{IF}. In principle, the delays originating L^{IF} may be different for each thread $\tau_i^{\text{THD}} \in \Gamma^{\text{THD}}$. However, for

[1] Note that $\texttt{RHP}_i$ is an event external to *the model*, for instance, it can be a hardware event that dispatches an IRQ, or the event that causes a thread to activate another thread.

simplicity, we conservatively bound L^{IF} in a thread-independent manner as discussed next in Section 4.2 and 5. The latency of τ_i^{THD} is then a function of the above delays, and is bounded by leveraging standard techniques for response-time analysis in real-time systems [3, 25, 26], i.e., by the least positive value fulfilling the following equation:

$$L = L^{\text{IF}} + I^{\text{NMI}}(L) + I^{\text{IRQ}}(L). \tag{1}$$

Next, we show how to bound L^{IF}.

4.2 Bounding L^{IF}

Analysis Approach. As discussed in Section 3, after the $\texttt{RHP}_i$ event occurs (i.e., when τ_i^{THD} becomes the ready thread with the highest priority), the kernel identifies the need to schedule a new thread when the `set_need_resched` event takes place. Then, an ordered sequence of events occurs. Such events are motivated by the operations and rules discussed in Section 3, graphically illustrated in the lower part of Figure 22, and discussed below.

EV1 The necessary conditions to call the scheduler need to be fulfilled: IRQs are enabled, and preemption is disabled to call the scheduler. It follows from rule R5 and R6;
EV2 The scheduler is called. It follows from R12;
EV3 In the scheduler code, IRQs are disabled to perform a context switch. It follows from rule R8;
EV4 The context switch occurs. It follows from rule R13 and R14;
EV5 Interrupts are enabled by the scheduler. It follows from R5;
EV6 The scheduler returns;
EV7 The preemption is enabled, returning the thread its own execution flow.

Note that, depending on what the processor is executing when the $\texttt{RHP}_i$ event occurs, not all the events may be involved in (and hence prolong) the scheduling latency. Figure 21 illustrates all the allowed sequences of events from the occurrence of the `set_need_resched` event (caused by $\texttt{RHP}_i$) until the context switch (EV4), allowing the occurrence of the other events (EV5-EV7). According to the automaton model, there are five possible and mutually-exclusive cases, highlighted with different colors in Figure 21. Our strategy for bounding L^{IF} consists in deriving an individual bound for each of the five cases, taking the maximum as a safe bound. To derive the five cases, we first distinguish between: **(i)** if $\texttt{RHP}_i$ occurs when the current thread $\tau_j^{\text{THD}} \in \Gamma_{\text{LP}_i}^{\text{THD}}$ is in the scheduler execution flow, both voluntarily, or involuntarily as a consequence of a previous `set_need_resched` occurrence, after disabling the preemption to call the scheduler and, **(ii)** otherwise.

We can distinguish three mutually-exclusive sub-cases of (i):

i-a if $\texttt{RHP}_i$ occurs between events EV1 and EV2, i.e., after that preemption has been disabled to call the scheduler and before the actual scheduler call (black in Figure 21);
i-b if $\texttt{RHP}_i$ occurs in the scheduler between EV2 and EV3, i.e., after that the scheduler has already been called and before interrupts have been disabled to cause the context switch (pink in Figure 21);
i-c if $\texttt{RHP}_i$ occurs in the scheduler between EV3 and EV7, i.e., after interrupts have already been masked in the scheduler code and when the scheduler returns (brown in Figure 21);

In case (ii), $\texttt{RHP}_i$ occurred when the current thread $\tau_j^{\text{THD}} \in \Gamma_{\text{LP}_i}^{\text{THD}}$ is not in the scheduler execution flow. Based on the automaton of Figure 21, two sub-cases are further differentiated:

ii-a when $\texttt{RHP}_i$ is caused by an IRQ, and the currently executing thread may delay $\texttt{RHP}_i$ only by disabling interruptions (green in Figure 21).
ii-b otherwise (blue in Figure 21).

Table 1 Parameters used to bound L^{IF}.

Param.	Length of the longest interval
D_{PSD}	in which preemptions are disabled to schedule.
D_{PAIE}	in which the system is in state `pe_ie` of Figure 21.
D_{POID}	in which the preemption is disabled to postpone the scheduler or IRQs are disabled.
D_{ST}	between two consecutive occurrences of EV3 and EV7.

Variables Selection. One of the most important design choices for the analysis consists in determining the most suitable variables to be used for deriving the analytical bound. Since the very early stages of its development, the PREEMPT_RT Linux had as a target to minimize the code portions executed in interrupt context and the code sections in which the preemption is disabled. One of the advantages of this design choice consists indeed in the reduction of scheduling delays. Nevertheless, disabling the preemption or IRQs is sometimes merely mandatory in the kernel code. As pointed out in Property 2, threads may also disable the preemption or IRQs, e.g., to enforce synchronization, thus impacting on the scheduling latency. Building upon the design principles of the fully-preemptive PREEMPT_RT kernel, Table 1 presents and discusses the set of variables selected to bound the latency, which are more extensively discussed next in Sections 5, and graphically illustrated in Figure 22. Such variables considers the longest intervals of time in which the preemption and/or IRQs are disabled, taking into consideration the different disabling modes discussed in Section 3.

Deriving the bound. Before discussing the details of the five cases, we present a bound on the interference-free duration of the scheduler code in Lemma 2.

▶ **Lemma 2.** *The interference-free duration of the scheduler code is bounded by* D_{PSD}.

Proof. It follows by noting that by rule R6 the scheduler is called and returns with the preemption disabled to call the scheduler and, by rules R2d, R3c, and R7, the preemption is not enabled again until the scheduler returns. ◀

Next, we provide a bound to L^{IF} in each of the five possible chains of events.

Case (i). In case (i), the preemption is already disabled to call the scheduler, hence either `set_need_resched` has already been triggered by another thread $\tau_j^{\text{THD}} \neq \tau_i^{\text{THD}}$ or the current thread voluntarily called the scheduler. Then, due to rules R13 and R14, a context switch will occur. Consequently, the processor continues executing the scheduler code. Due to rule R5, the scheduler is called with interrupts enabled and preemption disabled, hence $\texttt{RHP}_i$ (and consequently `set_need_resched`) must occur because of an event triggered by an interrupt. By rule R2, NMI cannot cause `set_need_resched`; consequently, it must be caused by an IRQ or the scheduler code itself. Due to EV3, IRQs are masked in the scheduler code before performing the context switch. We recall that case (i) divides into three possible sub-cases, depending on whether $\texttt{RHP}_i$ occurs between EV1 and EV2 (case i-a), EV2 and EV3 (case i-b), or EV3 and EV7 (case i-c). Lemma 3 bounds L^{IF} for cases (i-a) and (i-b).

▶ **Lemma 3.** *In cases (i-a) and (i-b), it holds*

$$L^{IF}_{(i-a)} \leq D_{PSD}, \; L^{IF}_{(i-b)} \leq D_{PSD}. \tag{2}$$

Proof. In both cases it holds that preemption is disabled to call the scheduler and IRQs have not been disabled yet (to perform the context switch) when $\texttt{RHP}_i$ occurs. Due to rules

R2 and R5, $\texttt{RHP}_i$ may only be triggered by an IRQ or the scheduler code itself. Hence, when $\texttt{RHP}_i$ occurs `set_need_resched` is triggered and the scheduler performs the context switch for τ_i^{THD}. Furthermore, in case (i-b) the processor already started executing the scheduler code when $\texttt{RHP}_i$ occurs. It follows that L^{IF} is bounded by the interference-free duration of the scheduler code. By Lemma 2, such a duration is bounded by D_{PSD}. In case (i-a), the scheduler has not been called yet, but preemptions have already been disabled to schedule. By rule R12, it will immediately cause a call to the scheduler, and the preemption is not enabled again between EV1 and EV2 (rules R2d, R3c, and R7). Therefore, also for case (i-a) L^{IF} is bounded by D_{PSD}, thus proving the lemma. ◀

Differently, case (i-c), in which $\texttt{RHP}_i$ occurs between EV3 and EV7, i.e., after interrupts are disabled to perform the context switch, is discussed in Lemma 4.

▶ **Lemma 4.** *In case (i-c), it holds*

$$L^{IF}_{(i-c)} \leq D_{ST} + D_{PAIE} + D_{PSD}. \tag{3}$$

Proof. In case (i), the scheduler is already executing to perform the context switch of a thread $\tau_j^{\text{THD}} \neq \tau_i^{\text{THD}}$. Due to rules R2 and R5, $\texttt{RHP}_i$ may only be triggered by an IRQ or the scheduler code itself. If the scheduler code itself caused $\texttt{RHP}_i$ before the context switch (i.e., between EV3 and EV4), the same scenario discussed for case (i-b) occurs, and the bound of Equation 2 holds. Then, case (i-c) occurs for $\texttt{RHP}_i$ arriving between EV4 and EV7 for the scheduler code, or EV3 and EV7 for IRQs. IRQs may be either disabled to perform the context switch (if $\texttt{RHP}_i$ occurs between EV3 and EV5), or already re-enabled because the context switch already took place (if $\texttt{RHP}_i$ occurs between EV5 and EV7). In both cases, thread τ_i^{THD} needs to wait for the scheduler code to complete the context switch for τ_j^{THD}. If $\texttt{RHP}_i$ occurred while IRQs were disabled (i.e., between EV3 and EV5), the IRQ causing $\texttt{RHP}_i$ is executed, triggering `set_need_resched`, when IRQs are enabled again just before the scheduler returns (see rule R5).

Hence, due to rule R14, the scheduler needs to execute again to perform a second context switch to let τ_i^{THD} execute. As shown in the automaton of Figure 21, there may exist a possible system state in case (i-c) (the brown one in Figure 21) in which, after $\texttt{RHP}_i$ occurred and before the scheduler code is called again, both the preemption and IRQs are enabled before calling the scheduler (state `pe_ie` in Figure 21). This system state is visited when the kernel is executing the non-atomic function to enable preemption, because the previous scheduler call (i.e., the one that caused the context switch for τ_j^{THD}) enabled IRQs before returning (EV5). Consequently, we can bound L^{IF} in case (i-c) by bounding the interference-free durations of the three intervals: I_{ST}, which lasts from EV3 to EV7, I_{PAIE}, which accounts for the kernel being in the state `pe_ie` of Figure 21 while executing EV7, and I_{S}, where preemption is disabled to call the scheduler and the scheduler is called again to schedule τ_i^{THD} (from EV1 to EV7). By definition and due to Lemma 2 and rules R2d, R3c, R7, and R12, I_{ST}, I_{PAIE}, and I_{S} cannot be longer than D_{ST}, D_{PAIE}, and D_{PSD}, respectively. The lemma follows by noting that the overall duration of L^{IF} is bounded by the sum of the individual bounds on I_{ST}, I_{PAIE}, and I_{S}. ◀

Case (ii). In case (ii), $\texttt{RHP}_i$ occurs when the current thread $\tau_j^{\text{THD}} \in \Gamma_{\text{LP}_i}^{\text{THD}}$ is not in the scheduler execution flow. As a consequence of the $\texttt{RHP}_i$ events, `set_need_resched` is triggered. By rule R14, triggering `set_need_resched` always result in a context switch and, since $\texttt{RHP}_i$ occurred outside the scheduler code, the scheduler needs to be called to perform the context switch (rule R9). Hence, we can bound L^{IF} in case (ii) by individually bounding two time

intervals I_{S} and I_{SO} in which the processor is *executing* or *not executing* the scheduler execution flow (from EV1 to EV7), respectively. As already discussed, the duration of I_{S} is bounded by D_{PSD} (Lemma 2). To bound I_{SO}, we need to consider individually cases (ii-a) and (ii-b). Lemma 5 and Lemma 6 bound L^{IF} for cases (ii-a) and (ii-b), respectively.

▶ **Lemma 5.** *In case (ii-a), it holds*

$$L^{IF}_{(ii-a)} \leq D_{POID} + D_{PSD}. \tag{4}$$

Proof. In case (ii-a) $\texttt{RHP}_i$ occurs due to an IRQ. Recall from Operation O3 that when an IRQ is executing, it masks interruptions. Hence, the IRQ causing $\texttt{RHP}_i$ can be delayed by the current thread or a lower-priority IRQ that disabled IRQs. When $\texttt{RHP}_i$ occurs, the IRQ triggering the event disables the preemption (IRQs are already masked) to fulfill R10 and R11, and triggers `set_need_resched`. If preemption was enabled before executing the IRQ handler and if `set_need_resched` was triggered, when the IRQ returns, it first disables preemptions (to call the scheduler, i.e., `preempt_disable_sched`). It then unmasks interrupts (this is a safety measure to avoid stack overflows due to multiple scheduler calls in the IRQ stack). This is done to fulfill the necessary conditions to call the scheduler discussed in rules R5 and R6. Due to rules R3a and R12, the scheduler is called once the IRQ returns. Hence, it follows that in the whole interval I_{SO}, either the preemption or interrupts are disabled. Then it follows that I_{SO} is bounded by D_{POID}, i.e., by the length of the longest interval in which either the preemption or IRQs are disabled. The lemma follows recalling that the duration of I_{S} is bounded by D_{PSD}. ◀

▶ **Lemma 6.** *In case (ii-b), it holds*

$$L^{IF}_{(ii-b)} \leq D_{POID} + D_{PAIE} + D_{PSD}, \tag{5}$$

Proof. In case (ii-b) the currently executing thread delayed the scheduler call by disabling the preemption or IRQs. The two cases in which the $\texttt{RHP}_i$ event is triggered either by a thread or an IRQ are discussed below.

(1) $\texttt{RHP}_i$ is triggered by an IRQ. Consider first that $\texttt{RHP}_i$ is triggered by an IRQ. Then, the IRQ may be postponed by a thread or a low-priority IRQ that disabled interrupts. When the IRQ is executed, it triggers `set_need_resched`. When returning, the IRQ returns to the previous preemption state[2], i.e, if it was disabled before the execution of the IRQ handler, preemption is disabled, otherwise it is enabled. If the preemption was enabled before executing the IRQ, the same scenario discussed for case (ii-a) occurs, and the bound of Equation 4 holds. Otherwise, if the preemption was disabled to postpone the scheduler execution, the scheduler is delayed due to priority-inversion blocking. Then it follows that when delaying the scheduler execution, either the preemption or IRQs are disabled. When preemption is re-enabled by threads and interrupts are enabled, the preemption needs to be disabled again (this time not to postpone the scheduler execution, but to call the scheduler) to fulfill the necessary conditions listed in rules R5 and R6, hence necessarily traversing the `pe_ie` state (shown in Figure 21), where both preemptions and interrupts are enabled. Hence, it follows that I_{SO} is bounded by $D_{\mathrm{POID}} + D_{\mathrm{PAIE}}$ if $\texttt{RHP}_i$ is triggered by an IRQ.

(2) $\texttt{RHP}_i$ is triggered by a thread. In this case, the thread triggers `set_need_resched`. Since the `set_need_resched` event requires IRQs and preemption disabled, the scheduler

[2] Note that, internally to the IRQ handler, the preemption state may be changed, e.g., to trigger `set_need_resched`.

execution is postponed until IRQs and preemption are enabled (`pe_ie` state). Once both are enabled, the preemption is disabled to call the scheduler. Then it follows that I_{SO} is bounded by $D_{\text{POID}} + D_{\text{PAIE}}$ if $\texttt{RHP}_i$ is triggered by a thread. Then it follows that I_{SO} is bounded by $D_{\text{POID}} + D_{\text{PAIE}}$ in case (ii-b). The lemma follows recalling that I_S is bounded by D_{PSD}. ◀

By leveraging the individual bounds on L^{IF} in the five cases discussed above, Lemma 7 provides an overall bound that is valid for all the possible events sequences.

▶ **Lemma 7.**

$$L^{IF} \leq max(D_{ST}, D_{POID}) + D_{PAIE} + D_{PSD}, \tag{6}$$

Proof. The lemma follows by noting that cases (i-a), (i-b), (i-c), (ii-a), (ii-b) are mutually-exclusive and cover all the possible sequences of events from the occurrence of $\texttt{RHP}_i$ and `set_need_resched`, to the time instant in which τ_i^{THD} is allowed to execute (as required by Definition 1), and the right-hand side of Equation 6 simultaneously upper bounds the right-hand sides of Equations 2, 3, 4, and 5. ◀

Theorem 8 summarizes the results derived in this section.

▶ **Theorem 8.** *The scheduling latency experienced by an arbitrary thread τ_i^{THD} is bounded by the least positive value that fulfills the following recursive equation:*

$$L = max(D_{ST}, D_{POID}) + D_{PAIE} + D_{PSD} + I^{NMI}(L) + I^{IRQ}(L) \tag{7}$$

Proof. The theorem follows directly from Lemmas 7 and Equation 1. ◀

5 rt_sched_latency: Efficient Scheduling Latency Estimation Tool Kit

The validation tool used in the development of the *Thread Synchronization Model* [14] exports all the kernel events to the user-space using `perf`, for later analysis. Although useful for the model validation purpose, the low granularity nature of the synchronization primitives generates a prohibitive amount of information for a performance measurement tool. For instance, one second of trace could generate more than 800 MB of data per CPU. Doing the whole trace analysis in-kernel has shown to be very efficient [18]. The problem for such an approach lies in the amount of information that can be stored in kernel memory. While only the worst observed value for some variables, such as D_{POID}, are used in the analysis, the IRQ and NMI analysis required the recording of all interrupts occurrence during the measurements. So the experimental tool kit developed in this work, called `rt_sched_latency`, has a hybrid approach: it uses an in-kernel event parsing and an extension to the `perf script` tool for a post-processing phase. Figure 23 describes the interaction of the tools in the tool kit. The tool kit comprises the `latency parser` and the `perf script` extension, named `rtsl`.

The `latency parser` uses the kernel `tracepoints` from the *Thread Synchronization Model* to observe their occurrence from inside the kernel. The `latency parser` registers a *callback function* to the kernel `tracepoints`. When a `tracepoint` from the model is hit, rather than writing the trace to the trace buffer (a buffer maintained by the `perf` tool to store trace data) the respective *function* is called. The *callback functions* are used to pre-process the events, transforming them into relevant information. For example, *nmi_entry* event records the arrival time (all the values are *observed values*, but the *observed* qualifiers are omitted for simplicity) without printing the occurrence of the event. When the *nmi_exit*

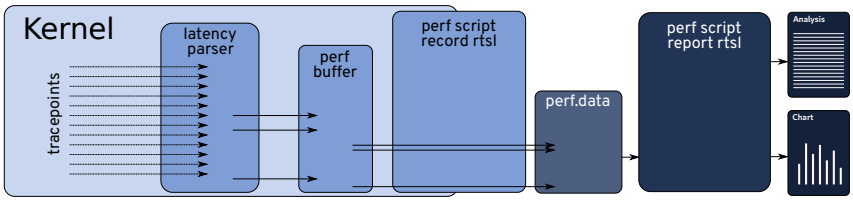


Figure 23 `rt_sched_latency`: tool kit components.

occurs, it computes the execution time of the NMI, and prints the arrival time and the execution time of the NMI. A similar behavior is implemented for other metrics, for instance for the IRQ occurence. The difference is that the interference must be removed from other metrics. For example, if an NMI and an IRQ occur while measuring a candidate D_{POID}, the IRQ and the NMI execution time are discounted from the measured value.

The `latency parser` communicates with `perf` using a new set of `tracepoints`, and these are printed to the trace buffer. The following events are generated by the `latency parser`:

- **irq_execution**: prints the IRQ identifier, starting time, and execution time;
- **nmi_execution**: prints the starting time, and execution time;
- **max_poid**: prints the new maximum observed D_{POID} duration;
- **max_psd**: prints the new maximum observed D_{PSD} duration;
- **max_dst**: prints the new maximum observed D_{ST} duration;
- **max_paie**: prints the new maximum observed D_{PAIE} duration;

By only tracing the return of interrupts and the new maximum values for the thread metrics, the amount of data generated is reduced to the order of 200KB of data per second per CPU. Hence, reducing the overhead of saving data to the trace buffer, while enabling the measurements to run for hours by saving the results to the disk. The data collection is done by the `perf rtsl script`. It initiates the `latency parser` and start recording its events, saving the results to the `perf.data` file. The command also accepts a workload as an argument. For example, the following command line will start the data collection while running `cyclictest` concurrently:

```
perf script record rtsl cyclictest -smp -p95 -m -q
```

Indeed, this is how the data collection is made for Section 6. The trace analysis is done with the following command line: `perf script report rtsl`. The perf script will read the `perf.data` and perform the analysis. A `cyclictest.txt` file with `cyclictest` output is also read by the script, adding its results to the analysis as well. The script to run the analysis is implemented in `python`, which facilitates the handling of data, needed mainly for the IRQ and NMI analysis.

IRQ and NMI analysis. While the variables used in the analysis are clearly defined (Table 1), the characterization of IRQs and NMI interference is delegated to functions (i.e., $I^{\text{NMI}}(L)$ and $I^{\text{IRQ}}(L)$), for which different characterizations are proposed next. The reason being is that there is no consensus on what could be the single best characterization of interrupt interference. For example, in a discussion among the Linux kernel developers, it is a common opinion that the classical sporadic model would be too pessimistic [17]. Therefore, this work assumes that there is no single way to characterize IRQs and NMIs, opting to explore different IRQs and NMI characterizations in the analysis. Also, the choice to analyze the

```
Interference Free Latency:
    paie is lower than 1 us -> neglectable
    latency = max(poid,    dst) + paie +   psd
      42212 = max(22510, 19312) +    0 + 19702
Cyclictest:
    Latency =     27000 with Cyclictest
No Interrupts:
    Latency =     42212 with No Interrupts
Sporadic:
    INT:     oWCET           oMIAT
    NMI:          0               0
     33:     16914          257130
     35:     12913            1843 <- oWCET > oMIAT
    236:     20728            1558 <- oWCET > oMIAT
    246:      3299         1910321
    Did not converge.
```

```
continuing....
Sliding window:
    Window: 42212
           NMI:        0
            33:    16914
            35:    14588
           236:    20728
           246:     3299
    Window: 97741
           236:    21029 <- new!
    Window: 98042
    Converged!
    Latency =     98042 with Sliding Window
```

Figure 24 `perf rtsl` output: excerpt from the textual output (time in nanoseconds).

```
# perf record -a -g -e rtsl:poid --filter "value > 60000"
# perf script
     php 25708 [001] 754905.013632: rtsl:poid: 68391
         ffffffff921cbb6d trace_preempt_on+0x13d ([kernel.kallsyms])
         ffffffff921039ca preempt_count_sub+0x9a ([kernel.kallsyms])
         ffffffff929a507a _raw_spin_unlock_irqrestore+0x2a ([kernel.kallsyms])
         ffffffff92109a55 wake_up_new_task+0x1c5 ([kernel.kallsyms])
         ffffffff920d4c5e _do_fork+0x14e ([kernel.kallsyms])
         ffffffff92004552 do_syscall_64+0x72 ([kernel.kallsyms])
         ffffffff92a00091 entry_SYSCALL_64_after_hwframe+0x49 ([kernel.kallsyms])
             7f2d61d7a685 __libc_fork+0xc5 (/usr/lib64/libc-2.26.so)
             55d87cba3b15 [unknown] (/usr/bin/php)
```

Figure 25 Using `perf` and the *latency parser* to find the cause of a large D_{POID} value.

data in user-space using `python` scripts were made to facilitate the extension of the analysis by other users or researchers. The tool presents the latency analysis assuming the following interrupts characterization:

- **No Interrupts**: the interference-free latency (L^{IF});
- **Worst single interrupt**: a single IRQ (the worst over all) and a single NMI occurrence;
- **Single (worst) of each interrupt**: a single (the worst) occurrence of each interrupt;
- **Sporadic**: sporadic model, using the observed minimum inter-arrival time and WCET;
- **Sliding window**: using the worst-observed arrival pattern of each interrupt and the observed execution time of individual instances;
- **Sliding window with oWCET**: using the worst-observed arrival pattern of each interrupt and the observed worst-case execution time among all the instances (oWCET).

These different characterization lead to different implementations of $I^{\text{NMI}}(L)$ and $I^{\text{IRQ}}(L)$.

perf rtsl output. The `perf rtsl` tool has two outputs: the textual and the graphical one. The textual output prints a detailed description of the latency analysis, including the values for the variables defined in Section 4. By doing so, it becomes clear what are the contributions of each variable to the resulting scheduling latency. An excerpt from the output is shown in Figure 24. The tool also creates charts displaying the latency results for each interrupt characterization, as shown in the experiments in Section 6.

When the dominant factor of latency is an IRQ or NMI, the textual output already serves to isolate the context in which the problem happens. However, when the dominant factor arises from a thread, the textual output points only to the variable that dominates the latency. Then, to assist in the search for the code section, the `tracepoints` that prints each occurrence of the variables from `latency parser` can be used. These events are not used during the measurements because they occur too frequently, but they can be used in

the debug stage. For example, Figure 25 shows the example of the `poid tracepoint` traced using `perf`, capturing the stack trace of the occurrence of a D_{POID} value higher than 60 microseconds[3]. In this example, it is possible to see that the spike occurs in the `php` thread while waking up a process during a `fork` operation. This trace is precious evidence, mainly because it is already isolated from other variables, such as the IRQs, that could point to the wrong direction.

6 Experimental Analysis

This section presents latency measurements, comparing the results found by `cyclictest` and `perf rtsl` while running concurrently in the same system. The main objective of this experimental study is to corroborate the practical applicability of the analysis tool. To this end, we show that the proposed approach provides latency bounds respecting the under millisecond requirement in scheduling precision (which is typical of applications using PREEMPT_RT) for most of the proposed interrupt characterizations. The proposed `perf rtsl` tool individually characterizes the various sources of latency and composes them leveraging a theory-based approach allowing to find highly latency-intensive schedules in a much shorter time than `cyclictest`. The experiment was made in a workstation with one Intel *i7-6700K CPU @ 4.00GHz* processor, with eight cores, and in a server with two Non-Uniform Memory Access (NUMA) Intel *Xeon L5640 CPU @ 2.27GHz* processors with six cores each. Both systems run the Fedora 31 Linux distribution, using the kernel-rt *5.2.21-rt14*. The systems were tuned according to the best practices of real-time Linux systems [34].

The first experiment runs on the workstation three different workloads for 30 minutes. In the first case, the system is mostly *idle*. Then workloads were generated using two `phoronix-test-suite` (`pts`) tests: the `openssl` stress test, which is a *CPU intensive* workload, and the `fio`, `stress-ng` and `build-linux-kernel` tests together, causing a mixed range of *I/O intensive* workload [31]. Different columns are reported in each graph, corresponding to the different characterization of interrupts discussed in Section 5. The result of this experiment is shown in Figure 26: 1.a, 1.b and 1.c, respectively. In the second experiment, the *I/O intensive* workload was executed again, with different test durations, as described in 2.a, 2.b, and 2.c. The results from `cyclictest` did not change substantially as the time and workload changes. On the other hand, the proposed approach results change, increasing the hypothetical bounds as the kernel load and experiment duration increase. Consistently with `cyclictest` results, the *No Interrupts* column also do not vary substantially. The difference comes from the interrupt workload: the more overloaded the system is, and the longer the tests run, the more interrupts are generated and observed, influencing the results. In all the cases, the *sporadic task model* appears to be overly pessimistic for IRQs: regularly, the *oWCET* of IRQs were longer than the minimal observed inter-arrival time of them. The *Sliding Window with oWCET* also stand out the other results. The results are truncated in the charts 2.b and 2.c: their values are 467 and 801 microseconds, respectively.

Although the reference automata model was developed considering single-core systems, the same synchronization rules are replicated in the multiple-core (*mc*) configuration, considering the local scheduling latency of each CPU. The difference between single and multiple-core cases resides in the inter-core synchronization using, for example, `spinlocks`. However, such synchronization requires preemption and IRQs to be disabled, hence, taking place inside the

[3] The *latency parser* `tracepoints` are also available via `ftrace`.

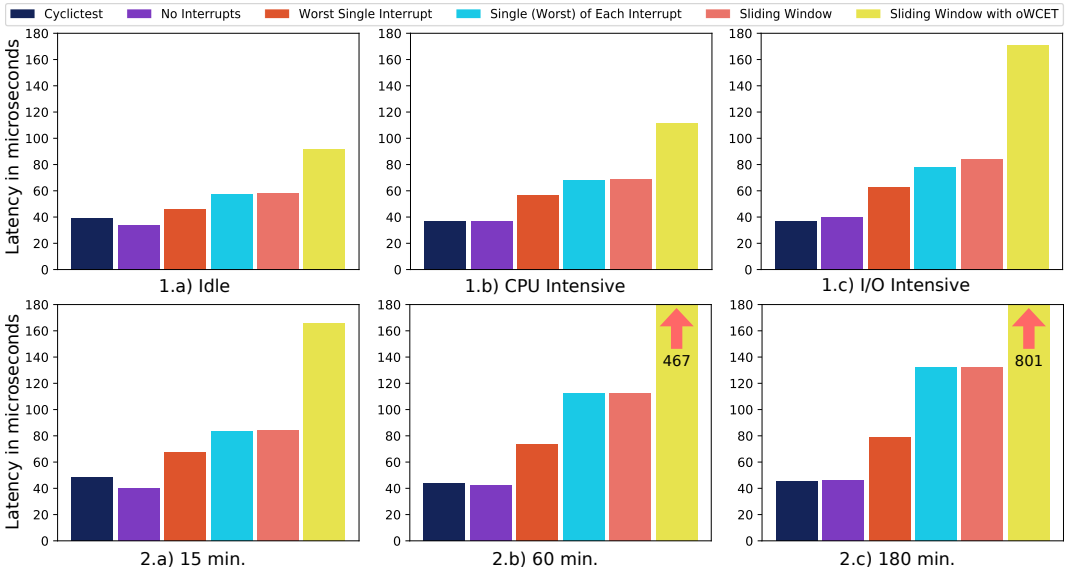


Figure 26 Workstation experiments: single-core system.

already defined variables. Moreover, when `cyclictest` runs in the `-smp` mode, it creates a thread per-core, aiming to measure the local scheduling latency. In a *mc* setup, the workload experiment was replicated in the workstation. Furthermore, the *I/O intensive* experiment was replicated in the server. The results of these experiments are shown in Figure 27. In these cases, the effects of the high kernel activation on I/O operations becomes evident in the workstation experiment (3.c) and in the server experiment(4.a). Again the *Sliding Window with oWCET* also stand out the other results, crossing the milliseconds barrier. The source of the higher values in the thread variables (Table 1) is due to cross-core synchronization using `spinlocks`. Indeed, the trace in Figure 25 was observed in the server running the *I/O* workload. The `php` process in that case was part of the `phoronix-test-suit` used to generate the workload.

Finally, by running `cyclictest` with and without using the `perf rtsl` tool, it was possible to observe that the trace impact in the minimum, average and maximum values are in the range from one to four microseconds, which is an acceptable range, given the frequency in which events occurs, and the advantages of the approach.

7 Conclusions and Future Work

The usage of the *Thread Synchronization Model* [14] was a useful logical step between the real-time theory and Linux, facilitating the information exchange among the related, but intricate, domains. The analysis, built upon a set of practically-relevant variables, ends up concluding what is informally known: the preemption and IRQ disabled sections, along with interrupts, are the evil for the scheduling latency. The tangible benefits of the proposed technique come from the decomposition of the variables, and the efficient method for observing the values. Now users and developers have precise information regarding the sources of the latency on their systems, facilitating the tuning, and the definition of where to improve the Linux code, respectively. The improvement of the tool and its integration with the Linux kernel and `perf` code base is the practical continuation of this work.

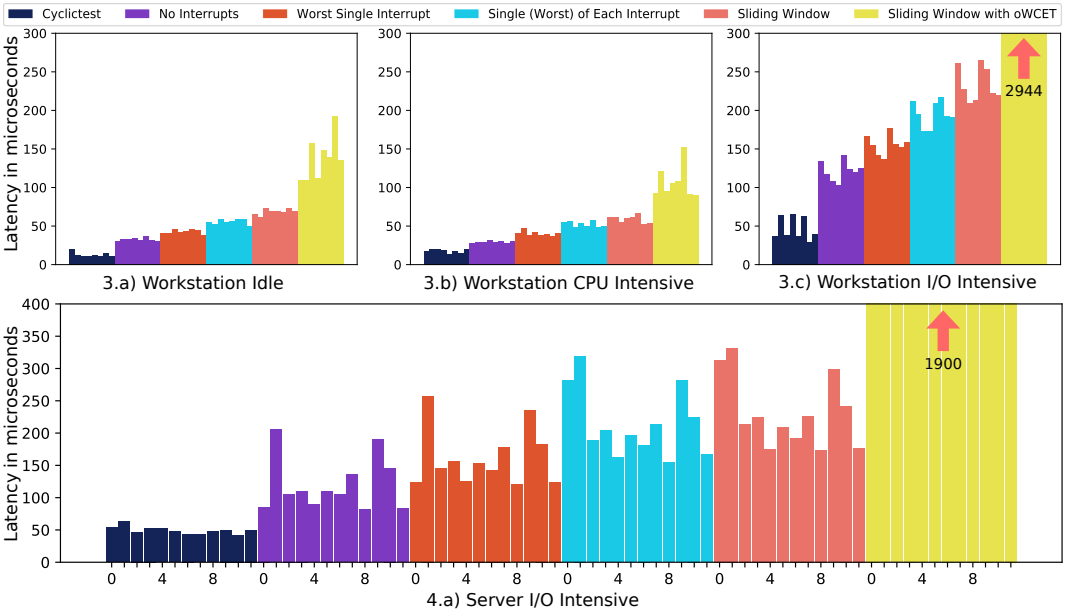


Figure 27 Workstation and Server experiments: multicore systems.

References

1 L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole. A measurement-based analysis of the real-time performance of linux. In *Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, September 2002.

2 Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 405–418, New York, NY, USA, 2018. ACM. doi:10.1145/3173162.3177156.

3 Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software engineering journal*, 8(5):284–292, 1993.

4 Bjorn Brandenbug and James Anderson. Joint Opportunities for Real-Time Linux and Real-Time System Research. In *Proceedings of the 11th Real-Time Linux Workshop (RTLWS 2009)*, pages 19–30, September 2009.

5 B. B. Brandenburg and M. Gül. Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 99–110, November 2016. doi:10.1109/RTSS.2016.019.

6 John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. Litmus^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, RTSS '06, pages 111–126, Washington, DC, USA, 2006. IEEE Computer Society. doi:10.1109/RTSS.2006.27.

7 Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Springer Publishing Company, Incorporated, 2nd edition, 2010.

8 F. Cerqueira and B. Brandenburg. A Comparison of Scheduling Latency in Linux, PREEMPT-RT, and LITMUS-RT. In *Proceedings of the 9th Annual Workshop on Operating Systems Platforms for Embedded Real-Time applications*, pages 19–29, 2013.

9 Felipe Cerqueira and Björn Brandenburg. A comparison of scheduling latency in linux, preempt-rt, and litmus rt. In *9th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 19–29. SYSGO AG, 2013.

10 H. Chishiro. RT-Seed: Real-Time Middleware for Semi-Fixed-Priority Scheduling. In *2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 124–133, May 2016. doi:10.1109/ISORC.2016.26.

11 J. Corbet. Linux at NASDAQ OMX, October 2010.
URL: https://lwn.net/Articles/411064/.

12 T. Cucinotta, A. Mancina, G. F. Anastasi, G. Lipari, L. Mangeruca, R. Checcozzo, and F. Rusina. A real-time service-oriented architecture for industrial automation. *IEEE Transactions on Industrial Informatics*, 5(3):267–277, August 2009. doi:10.1109/TII.2009.2027013.

13 D. B. de Oliveira, R. S. de Oliveira, T. Cucinotta, and L. Abeni. Automata-based modeling of interrupts in the Linux PREEMPT RT kernel. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, September 2017. doi:10.1109/ETFA.2017.8247611.

14 Daniel B. de Oliveira, Rômulo S. de Oliveira, and Tommaso Cucinotta. A thread synchronization model for the preempt_rt linux kernel. *Journal of Systems Architecture*, page 101729, 2020. doi:10.1016/j.sysarc.2020.101729.

15 Daniel Bristot de Oliveira. Mind the gap between real-time Linux and real-time theory, Part I, 2018.
URL: https://wiki.linuxfoundation.org/realtime/events/rt-summit2018/schedule#abstracts.

16 Daniel Bristot de Oliveira. Mind the gap between real-time Linux and real-time theory, Part II, 2018.
URL: https://www.linuxplumbersconf.org/event/2/contributions/75/.

17 Daniel Bristot de Oliveira. Mathmatizing the Latency - Presentation at the Real-time Linux micro-conference, at the Linux Plumbers Conference, September 2019.
URL: https://linuxplumbersconf.org/event/4/contributions/413/.

18 Daniel Bristot de Oliveira, Tommaso Cucinotta, and Rômulo Silva de Oliveira. Efficient formal verification for the linux kernel. In *International Conference on Software Engineering and Formal Methods*, pages 315–332. Springer, 2019.

19 Daniel Bristot de Oliveira, Tommaso Cucinotta, and Rômulo Silva de Oliveira. Untangling the Intricacies of Thread Synchronization in the PREEMPT_RT Linux Kernel. In *Proceedings of the IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*, Valencia, Spain, May 2019.

20 Daniel Bristot de Oliveira and Rômulo Silva de Oliveira. Timing analysis of the PREEMPT_RT Linux kernel. *Softw., Pract. Exper.*, 46(6):789–819, 2016. doi:10.1002/spe.2333.

21 A. Dubey, G. Karsai, and S. Abdelwahed. Compensating for timing jitter in computing systems with general-purpose operating systems. In *2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 55–62, March 2009. doi:10.1109/ISORC.2009.28.

22 Hasan Fayyad-Kazan, Luc Perneel, and Martin Timmerman. Linux preempt-rt vs commercial rtoss: How big is the performance gap? *GSTF Journal on Computing*, 3(1), 2013.

23 Thomas Gleixner. Realtime Linux: academia v. reality. *Linux Weekly News*, July 2010. URL: https://lwn.net/Articles/397422/.

24 B. Herzog, L. Gerhorst, B. Heinloth, S. Reif, T. Hönig, and W. Schröder-Preikschat. Intspect: Interrupt latencies in the linux kernel. In *2018 VIII Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 83–90, November 2018.

25 M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, January 1986.

26 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*, pages 166–171, December 1989.

27 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.

28 Linux Kernel Documentation. Linux tracing technologies. https://www.kernel.org/doc/html/latest/trace/index.html, February 2020.

29 G. Matni and M. Dagenais. Automata-based approach for kernel trace analysis. In *2009 Canadian Conference on Electrical and Computer Engineering*, pages 970–973, May 2009. doi:10.1109/CCECE.2009.5090273.

30 L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari. AQuoSA – Adaptive Quality of Service Architecture. *Softw. Pract. Exper.*, 39(1):1–31, January 2009. doi:10.1002/spe.v39:1.

31 Phoronix Test Suite. Open-source, automated benchmarking. www.phoronix-test-suite.com, February 2020.

32 Josh Poimboeuf. Introducing kpatch: Dynamic kernel patching. https://www.redhat.com/en/blog/introducing-kpatch-dynamic-kernel-patching, February 2014.

33 P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1):206–230, January 1987. doi:10.1137/0325013.

34 Red Hat. Inc,. Advanced tuning procedures to optimize latency in RHEL for Real Time. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/8/html/tuning_guide/index, February 2020.

35 Red Hat. Inc,. Red Hat Enterprise Linux Hardware Certification. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_hardware_certification/1.0/html/test_suite_user_guide/sect-layered-product-certs#cert-for-rhel-for-real-time, February 2020.

36 F. Reghenzani, G. Massari, and W. Fornaciari. Mixed time-criticality process interferences characterization on a multicore linux system. In *2017 Euromicro Conference on Digital System Design (DSD)*, August 2017.

37 Paul Regnier, George Lima, and Luciano Barreto. Evaluation of interrupt handling timeliness in real-time linux operating systems. *ACM SIGOPS Operating Systems Review*, 42(6):52–63, 2008.

38 Steven Rostedt. Finding origins of latencies using ftrace, 2009.

39 Carlos San Vicente Gutiérrez, Lander Usategui San Juan, Irati Zamalloa Ugarte, and Víctor Mayoral Vilches. Real-time linux communications: an evaluation of the linux communication stack for real-time robotic applications, August 2018.
URL: https://arxiv.org/pdf/1808.10821.pdf.

AMD GPUs as an Alternative to NVIDIA for Supporting Real-Time Workloads

Nathan Otterness
The University of North Carolina at Chapel Hill, NC, USA
otternes@cs.unc.edu

James H. Anderson
The University of North Carolina at Chapel Hill, NC, USA
anderson@cs.unc.edu

Abstract

Graphics processing units (GPUs) manufactured by NVIDIA continue to dominate many fields of research, including real-time GPU-management. NVIDIA's status as a key enabling technology for deep learning and image processing makes this unsurprising, especially when combined with the company's push into embedded, safety-critical domains like autonomous driving. NVIDIA's primary competitor, AMD, has received comparatively little attention, due in part to few embedded offerings and a lack of support from popular deep-learning toolkits. Recently, however, AMD's ROCm (Radeon Open Compute) software platform was made available to address at least the second of these two issues, but is ROCm worth the attention of safety-critical software developers? In order to answer this question, this paper explores the features and pitfalls of AMD GPUs, focusing on contrasting details with NVIDIA's GPU hardware and software. We argue that an open software stack such as ROCm may be able to provide much-needed flexibility and reproducibility in the context of real-time GPU research, where new algorithmic or analysis techniques should typically remain agnostic to the underlying GPU architecture. In support of this claim, we summarize how closed-source platforms have obstructed prior research using NVIDIA GPUs, and then demonstrate that AMD may be a viable alternative by modifying components of the ROCm software stack to implement spatial partitioning. Finally, we present a case study using the PyTorch deep-learning framework that demonstrates the impact such modifications can have on complex real-world software.

2012 ACM Subject Classification Computer systems organization → Heterogeneous (hybrid) systems; Computer systems organization → Real-time systems; Software and its engineering → Scheduling; Software and its engineering → Concurrency control; Computing methodologies → Graphics processors; Computing methodologies → Concurrent computing methodologies

Keywords and phrases real-time systems, graphics processing units, parallel computing

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.10

Supplementary Material Source code is available online, including modified ROCm code (`https://github.com/yalue/rocm_mega_repo/tree/hip_modification_only`), scripts and source-code patches for PyTorch (`https://github.com/yalue/ecrts_2020_pytorch_experiment`), and microbenchmarking framework (`https://github.com/yalue/hip_plugin_framework`).

Funding Work was supported by NSF grants CNS 1563845, CNS 1717589, and CPS 1837337, ARO grant W911NF-17-1-0294, ONR grant N00014-20-1-2698, and funding from General Motors.

1 Introduction

While the battle among chip manufacturers for dominance in the GPU (graphics processing unit) market has raged for years, the most prominent contenders for at least the past decade have been two companies: NVIDIA and AMD. Of these, NVIDIA is currently the clear leader, boasting nearly 77% of the market share for discrete desktop GPUs [37] at the time of writing.

Unfortunately, the new reality of autonomous vehicles means that artificial-intelligence and image-processing accelerators, including GPUs, cannot be relegated to the domains of datacenters and video games. When human lives directly depend on GPUs, software developers should seek the safest option irrespective of brand loyalty or sales numbers. In short, the real-time GPU research community should not ignore either side in the NVIDIA-vs.-AMD battle.

Much of NVIDIA's dominance in the area of general-purpose GPU computing can be attributed to the popularity of CUDA, NVIDIA's proprietary GPU programming framework [32]. On top of CUDA itself, NVIDIA has heavily supported and encouraged the use of its GPUs in the area of artificial intelligence, as exemplified by its popular CUDA library supporting deep neural networks, cuDNN [33]. cuDNN was first released in 2007, and has since been adopted into many prominent deep-learning and image-processing frameworks such as Caffe [25], PyTorch [4], and TensorFlow [1].

In contrast, these popular deep-learning frameworks lacked AMD support for years, causing AMD to fall behind in research adoption. However, AMD has recently been improving its own response to CUDA with the ROCm software stack [12], which includes HIP, a GPU programming interface sporting near-identical features to CUDA. The introduction of HIP has allowed AMD to quickly develop versions of the aforementioned deep-learning and AI frameworks compatible with its own GPUs. For example, as of May 2018, the official PyTorch repository has shipped with scripts that allow it to be compiled for AMD GPUs, using ROCm.[1] Similarly, AMD maintains a fork of TensorFlow compatible with its own GPUs,[2] despite a lack of upstream support. Despite AMD's efforts, however, the dearth of research directed at AMD GPUs could lead one to question whether a serious competitor to NVIDIA even exists. In this paper we wish to rectify this lack of research by focusing on AMD GPUs, and how their features may be used to improve timing safety in a real-time setting.

Given that NVIDIA GPUs are well-established and better-studied, what do AMD GPUs have to offer real-time programmers? The answer requires considering one of the most important research questions pertaining to real-time GPU usage: *how can multiple applications safely and efficiently share a GPU?* Size, weight, and power concerns are already leading to increased centralization in embedded processors for autonomous systems, meaning that a wider range of devices and functions are managed by a smaller number of processors. Following this trend, future embedded applications will likely need to share a small number of low-power GPUs. Doing so inevitably increases hardware contention, which in turn, if not managed carefully, can lead to unsafe timing or extreme pessimism in analysis.

We claim that some features currently unique to AMD GPUs, especially *an open-source software stack* and *support for hardware partitioning*, have the potential to accelerate and aid the long-term viability of real-time GPU research. Of course AMD GPUs also have their own set of drawbacks, some of which may justifiably warrant continued research on NVIDIA GPUs regardless of the availability of an alternative. So, in this paper we also examine some drawbacks of AMD GPUs, namely, fewer supported platforms, less documentation, and instability stemming from less-mature software. To summarize: a greater variety of research platforms can only serve to help advance the field, even if some work is not possible on every alternative. After all, we do not seek to declare victory for one GPU manufacturer or the other, but to further the cause of developing safer GPU-accelerated software.

[1] `https://github.com/pytorch/pytorch/commit/cd86d4c5548c15e0bc9773565fa4fad73569f948`
[2] `https://github.com/ROCmSoftwarePlatform/tensorflow-upstream`

Contributions. In this paper, our primary goal is to shed light on an under-explored platform in real-time research: AMD GPUs. In doing so, we identify potential benefits and drawbacks of AMD GPUs, along with their implications for real-time GPU management. We especially contrast features of AMD GPUs with those offered by NVIDIA in the context of how different possible approaches could reshape real-time GPU research. Finally, we describe our implementation of several modifications to AMD's software stack to expose a basic hardware-partitioning API, which we apply to improve timing predictability when multiple PyTorch neural networks share a single GPU.

Organization. The rest of the paper is organized as follows. Sec. 2 describes background and relevant work on real-time GPU management. Next, Sec. 3 further contrasts features of NVIDIA and AMD GPUs, and introduces several benefits and drawbacks of AMD's platform. Sec. 4 describes our case studies, including our modifications to the ROCm framework and PyTorch. Finally, Sec. 5 contains our concluding remarks and plans for future work.

2 Background and Related Work

We begin by providing essential information about GPU hardware, software, and prior research.

2.1 Overview of GPU Programming

Despite their different vendors, general-purpose programming on both NVIDIA and AMD GPUs requires following similar steps, so this section attempts to give a general overview while remaining vendor-agnostic, leaving a detailed comparison for Sec. 3.

Programs must carry out the following steps to run code on a GPU:

1. Copy input data from CPU memory to GPU memory.
2. Invoke a GPU kernel to process the data. (In the context of GPU programming, the term *kernel* typically refers to a section of code that runs on a GPU.) The program must also specify the number of parallel GPU threads to use.
3. Copy output data from GPU memory to CPU memory.

All three of these steps (including other details such as allocating and freeing GPU memory buffers) are carried out at the request of a CPU program, via a GPU-programming API. Typically, the program issues commands to one or more API-managed queues, called *streams*, after which the underlying runtime will execute the commands in FIFO order. On an NVIDIA-based system, the API of choice is typically CUDA, with HIP being a close analogue for AMD GPUs. We draw special attention to the API, because in a GPU-augmented safety-critical system, *the GPU API is safety-critical software.*

GPU hardware organization. Fig. 1 shows the high-level organization of the AMD RX 570 GPU, used as a test platform in this paper. Even though Fig. 1 shows an AMD GPU, NVIDIA GPUs exhibit a similar high-level layout, albeit with some different terminology.

A *discrete* GPU, such as the AMD RX 570, consists of self-contained DRAM and processors, and typically communicates with a host system using the PCI bus. This is in contrast to an *integrated* GPU, which shares components, possibly including memory, with the CPU. Both NVIDIA and AMD GPUs contain a shared L2 cache in addition to separate L1 caches assigned to independent compute units (CUs).

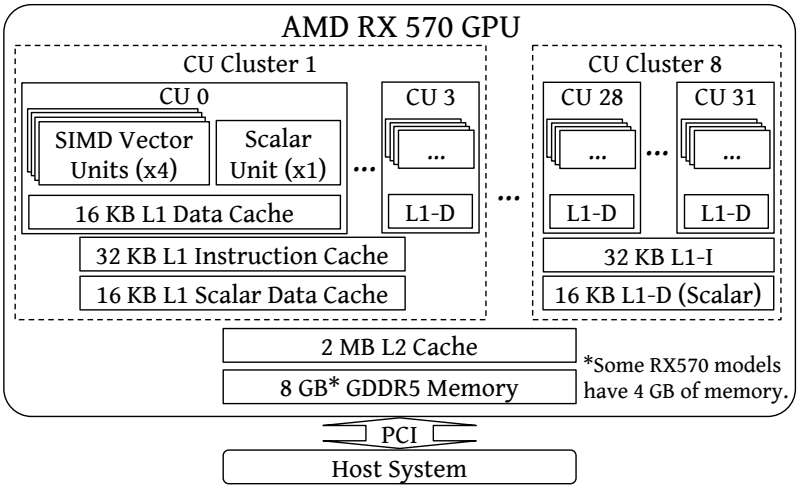


Figure 1 Organization of the AMD RX 570 GPU.

A GPU's parallel-computing capability comes from its collection of CUs (known as *streaming multiprocessors*, or SMs, in NVIDIA GPUs). Each CU is responsible for a subset of parallel threads associated with a kernel. In reality, a GPU's SIMT (single-instruction-multiple-thread) architecture means that GPU threads are organized into groups, where each thread in a group executes the same instruction at a given time on a different piece of data. This SIMT architecture simplifies GPU programming because it allows programmers to write GPU-accelerated programs using roughly the same logic as a traditional multithreaded application. This comes at the expense of potential inefficiencies in cases where GPU kernels contain code with a heavily divergent control flow, but avoiding inefficient, branchy code is typically left as a responsibility of the programmer.

In addition to the memory and compute units described above, GPUs contain many other necessary pieces of hardware that are not represented in Fig. 1, including dedicated memory copy engines responsible for transferring data to and from CPU memory, and significant portions of hardware exclusively used for rendering graphics.

2.2 Related Work

This section gives an overview of prior research in real-time GPU management. We only give a brief overview of relevant work and topics here. We save further details concerning how several of these works implement real-time GPU management for Sec. 3, after we have described relevant GPU features in more detail.

Early real-time GPU-management systems focused on scheduling jobs on entire GPUs. Some of the earliest publications apply traditional non-preemptive real-time scheduling techniques to GPU computations [27, 28]. Subsequent works include a body of papers by Verner et al., that assign tasks to GPUs in ways that improve worst-case response times [38, 39, 40]. An alternative approach, by Elliott et al., treats GPU access as a resource-locking problem [17].

Other prior real-time GPU work seeks to reduce the impact of non-preemptivity on GPU computations, and does so by subdividing larger kernels and data transfers into chunks that individually require less time to complete, reducing worst-case blocking. This includes some work from the prior paragraph [27], in addition to several other papers [5, 30, 43].

Recent research has shifted from scheduling work on multi-GPU systems to *GPU sharing*, necessitated by the increasing prominence of embedded GPUs. Many of the works discussed in the previous paragraphs view GPUs as a black box, which leads to an obvious difficulty: *timing depends on unknown intra-GPU scheduling behavior*. This led those early works to avoid GPU sharing altogether, overriding the decision-making abilities of the "black box" hardware and software by only allowing it to schedule one job per GPU at a time.

There have been a few ways in which real-time GPU research has attempted to enable concurrent GPU sharing. In an attempt to provide a behavioral model for GPU-sharing task systems, our group has used black-box experiments to infer additional details about the scheduling behavior of NVIDIA GPUs [3, 34]. In 2018, Jain et al. presented a work that enables partitioning L2 caches, DRAM, and GPU compute units in order to reduce interference during GPU sharing [24]. This was preceded by other approaches that partition compute units only [36].

Some real-time GPU-sharing research focuses on improving predictable access to memory on platforms with integrated GPUs where, as mentioned in Sec. 2.1, memory is shared between the CPU and GPU. These works propose techniques including scheduling-based approaches to isolate memory-sensitive GPU work [7, 18], and throttling-based techniques to limit worst-case memory contention [2, 22].

Finally, some researchers have opted to collaborate with NVIDIA in order to obtain access to internal closed-source code and documentation. One such effort, by Capodieci et al., implements earliest-deadline-first (EDF) scheduling on an embedded NVIDIA GPU [6].

Prior real-time research using non-NVIDIA GPUs. Even though some of the concepts from prior research evaluated for NVIDIA GPUs would be applicable on other systems, very little real-time GPU research has specifically targeted non-NVIDIA systems. The only recent real-time GPU paper explicitly implemented using a non-NVIDIA system is by Lee et al., and implements transactional GPU kernel support for a Samsung Exynos system-on-chip [31].

In fairness, and as mentioned in Sec. 1, AMD continues to lag behind NVIDIA in broader research adoption. Additionally, AMD's open-source ROCm software stack was first released in 2016 [12], and the `amdgpu` open-source Linux driver was released only slightly earlier. This has left relatively little time for a body of research to be established. In any case, to the best of our knowledge, this paper is the first that considers real-time GPU computing specifically using AMD GPUs.

3 Comparison between NVIDIA and AMD

We now provide further details regarding AMD GPUs. We assume that most readers will be more familiar with NVIDIA GPUs, so we often explain the behavior of AMD GPUs in contrast to those from NVIDIA. This section serves several purposes: **1)** familiarize readers with AMD GPU architectures and software, **2)** contrast AMD GPUs with NVIDIA GPUs, and **3)** explain the potential impact of various GPU features on real-time systems, especially in light of prior NVIDIA-centric research.

3.1 Software for GPGPU Programming

As mentioned earlier, safety-critical GPU software includes not only user-level applications, but also all of the underlying library and driver code. In the following paragraphs, we focus on the software used for general-purpose GPU (GPGPU) programming, how real-time work focusing on NVIDIA's CUDA framework is implemented, and how approaches can change with more openness.

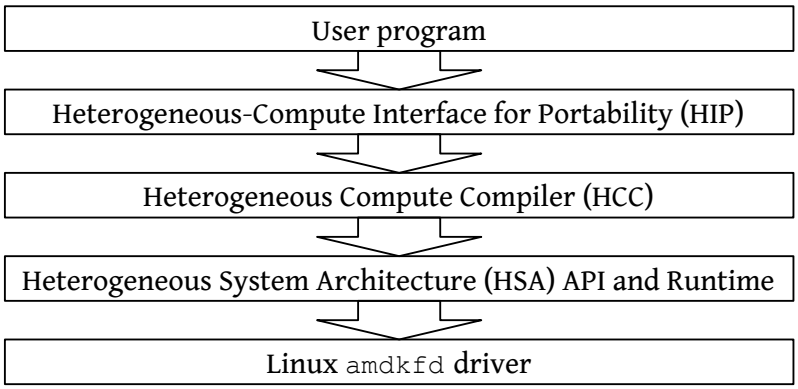


Figure 2 The ROCm software stack used for AMD GPGPU programming.

The NVIDIA GPU software stack. Even though other GPU-programming frameworks may offer performance benefits [8], the CUDA toolkit remains the best-studied and most popular method for general-purpose programming on NVIDIA GPUs. In brief, *CUDA* collectively refers to the CUDA compiler, API, and runtime library, which in turn relies on the GPU driver. Due to the opaque nature of all parts of a CUDA program below the API level, a typical user has no ability to directly modify any components below the user-level program. Additionally, even documented portions of the CUDA API often lack important details or contain inaccurate information about timing [41].

The AMD GPU software stack. AMD's ROCm (Radeon Open Compute) platform is largely analogous to the CUDA platform on NVIDIA systems. While the CUDA framework can be considered fairly monolithic (at least to its end users), ROCm is openly composed of several different modules and can support multiple programming languages [12]. However, the most important difference for the sake of real-time work is undoutedly the following:

▶ **Benefit 1.** *The ROCm software stack for computations on AMD GPUs is open source.*

Benefit 1 applies to every aspect of the ROCm stack used to program AMD GPUs, shown in Fig. 2. At the bottom of the stack is the `amdkfd` driver, which was added to the mainline Linux kernel in 2014 [29]. Above the driver lies ROCm's implementation of the Heterogeneous System Architecture (HSA) API, which was designed as a lightweight user-level API wrapping low-level functionality like memory-management, synchronization routines, kernel queues, etc.[20]. The next "layer" of the ROCm stack is the Heterogeneous Compute Compiler (HCC). HCC is a LLVM-based compiler capable of compiling code to target AMD GPUs, and uses the HSA API behind the scenes.

Finally, HIP (Heterogeneous-Compute Interface for Portability) is a programming language specifically designed to simplify porting CUDA applications to AMD GPUs. Technically, the HIP language is capable of targeting both AMD and NVIDIA GPUs: when targeting AMD the features of HIP are ultimately implemented on top of the HCC compiler, and when targeting NVIDIA it uses the CUDA compiler and a set of lightweight wrapper functions around the CUDA API. HIP's platform independence is facilitated by the fact that HIP, as a programming language, is nearly identical to CUDA. Converting CUDA to HIP code is only slightly more complicated than a find-and-replace of the term "`cuda`" with "`hip`" in source code. In fact, AMD provides a tool, `hipify`, that attempts to automatically transform CUDA code to HIP.

```
// Define a kernel setting vector c = a + b
__global__ void VectorAdd(int *a, int *b, int *c) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  c[i] = a[i] + b[i];
}
int main() {
  // ... Allocate vectors a, b, and c
  // Launch the kernel
  hipLaunchKernelGGL(VectorAdd, block_count, thread_count,
    0, 0, a, b, c);
  // Wait for the kernel to complete
  hipDeviceSynchronize();
  // Copy results from GPU, cleanup, etc.
}
```

Figure 3 Definition and launch of a vector-add kernel using the HIP programming language.

```
// Define a kernel setting vector c = a + b
__global__ void VectorAdd(int *a, int *b, int *c) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  c[i] = a[i] + b[i];
}
int main() {
  // ... Allocate vectors a, b, and c
  // Launch the kernel
  VectorAdd<<<block_count, thread_count>>>(a, b, c);
  // Wait for the kernel to complete
  cudaDeviceSynchronize();
  // Copy results from GPU, cleanup, etc.
}
```

Figure 4 Definition and launch of a vector-add kernel using the CUDA programming language.

▶ **Benefit 2.** *Existing CUDA code can often be automatically converted to the cross-platform HIP programming language.*

As a concrete illustration of Benefit 2, Figs. 3 and 4 show the same kernel and CPU code fragment written in HIP and CUDA, respectively. In this simple case, the `VectorAdd` kernel code required no changes at all, with the only changes being the syntax for invoking the kernel and the `...DeviceSynchronize` function, which blocks until the kernel completes. For the sake of space, Figs. 3 and 4 omit `#include` directives and code for memory management or error checking, but all such changes are similar to the those already represented in the figures.

Of course, notable exceptions exist to the automatic conversion, such as GPU-architecture-specific inline assembly, which tends to be used for manual optimizations or accessing model-specific GPU registers.

The importance of openness. The appeal of a software stack like ROCm hinges on the answer to one question: why should developers care that ROCm is open source? Fortunately, the potential benefit of an open-source stack on real-time GPU research is easy to demonstrate. To see why, one only needs to revisit how prior real-time GPU management work was implemented.

TimeGraph [28] and *GPES* [43] require using the third-party open-source "Nouveau" driver for NVIDIA GPUs, which does not support CUDA. *RGEM* [27] uses a third-party

CUDA implementation by PathScale, which has by now been discontinued for several years.[3] Papers that rely on reverse engineering [3, 24] require re-validation after every hardware or software change, and papers that intercept driver communication like *GPUSync* [17] are subject to a stable interface between the CUDA runtime libraries and the underlying device driver, but NVIDIA makes no such stability guarantees for these non-public implementation details. The key point here is that, when using NVIDIA GPUs, *working around the lack of source-code access is a prerequisite for meaningful research.*

Other prior work demonstrates what is possible with a greater amount of access to internal GPU details. For example, working in collaboration with NVIDIA enabled Capodieci et al. to implement the popular preemptive EDF real-time scheduler for GPU workloads, by directly modifying NVIDIA's source code [6]. In a second example, a significant reverse-engineering effort yielded enough internal details to enable Jain et al. to implement not only SM partitioning but also L2 cache and DRAM partitioning for NVIDIA GPUs [24]. Naturally, these two papers also worked around the lack of open-source access as well as those mentioned in the previous paragraph, but they also illustrate an additional conclusion: *improved access to GPU internals enables fundamentally more powerful management paradigms.*

The ROCm stack means that AMD GPUs are subject to almost none of the software-based limitations discussed above. Developers can modify the underlying device drivers and user-level HSA libraries, removing the need to use less-supported libraries or to enter non-disclosure agreements simply to add more functionality to existing code. Additionally, transformations of application-level source code could largely be rendered unnecessary given the ability to directly modify HIP or the HCC compiler. All of this means that a GPU-management system targeting AMD GPUs using the ROCm stack should not only be easier to implement, but also easier to use and easier for third parties (i.e., other researchers) to validate.

3.2 Hardware Organization in AMD GPUs

We now shift our focus to the hardware organization of AMD GPUs, in this case exemplified by our test platform: the RX 570 represented in Fig. 1.

In 2011, AMD announced its *Graphics Core Next* (GCN) GPU architecture. For AMD, the GCN architecture was characterized by an increased focus on general-purpose GPU computing. This focus was represented in hardware by AMD's transition away from VLIW (very long instruction word) processors to SIMD (single instruction, multiple data) processors [10]. AMD's GCN architecture remains in use to this day, with progressive updates to accommodate new hardware. GCN's successor, codenamed "RDNA" [15], first appeared in consumer GPUs in 2019, but is not yet officially supported by the ROCm framework.[4] The RX 570, used in this paper, was released in 2017 and uses the fourth generation of the GCN architecture. Many of the interesting features visible in Fig. 1, including the L1 caches and computational elements, are actually specified by the GCN architecture and have remained relatively consistent since the first generation of GCN.

Processing in GCN GPUs. In AMD GPUs, parallel processors are divided into *Compute Units* (CUs), which are highly analogous to the *Streaming Multiprocessors* (SMs) in NVIDIA GPUs. The bulk of each CU's processing power lies within its four SIMD "vector units." Each

[3] According to the PathScale website: `https://www.pathscale.com/`.

[4] At the time of writing, "Navi" or "RDNA" GPUs are not present in the official list at: `https://github.com/RadeonOpenCompute/ROCm#Hardware-and-Software-Support`.

vector unit has sixteen *lanes*, meaning that each is capable of carrying out a computation on sixteen different pieces of data in parallel. Since each CU contains four sixteen-lane vector units, each CU can carry out up to 64 threads' worth of computation at any given instant. This is highly similar to NVIDIA SMs, which generally have 64 or 128 "CUDA cores," each of which carries out a computation for a single thread. Each CU in a GCN GPU also contains a single *scalar unit*, generally used for instructions such as control-flow operations.

Despite having 64 SIMD lanes per CU, each CU in the RX 570 can support up to 2,560 in-flight threads at a time.[5] This is similar to the SMs in NVIDIA GPUs, which each support up to 2,048 in-flight threads. Both NVIDIA and AMD GPUs use the large number of in-flight threads for *latency hiding*, swapping to a different group of 64 threads whenever the currently executing group of 64 stalls for some reason, e.g., while waiting for memory.

GPU threads and thread blocks. We briefly return to Figs. 3 and 4 to call attention to the fact that both HIP and CUDA kernel code make use of GPU threads and *thread blocks*. This is seen in the `int i = ...` lines in the `VectorAdd` kernels in both figures, where the thread and thread-block indices are used to select a vector element to process. In both CUDA and HIP, thread blocks act as *schedulable entities* – a single thread block is never split across CUs (or SMs on NVIDIA GPUs), and all threads from a thread block must finish executing before a block is considered complete. After a block completes execution, its threads become available, allowing a new block to start executing on the same CU.

On both NVIDIA and AMD GPUs, the size of a thread block is specified by the programmer at runtime, when a kernel is launched. This can be seen in the `hipLaunchKernel...` line of Fig. 3 and the `VectorAdd«<...»>` line of Fig. 4. Blocks are limited to at most 1,024 threads on GPUs from both manufacturers, but programs requiring more than 1,024 threads can request multiple thread blocks.

CU management. Understanding the division of processing capacity into CUs is important: due to the fundamental division of modern GPUs' computing resources among CUs or SMs, partitioning tasks to non-overlapping sets of CUs intuitively should reduce possible interference when sharing a GPU among multiple real-time workloads. In fact, such an idea has received attention from both manufacturers. NVIDIA introduced a mechanism called *Execution Resource Provisioning* (ERP) with its Volta GPU architecture, but ERP only allows users to limit the number of SMs a task can use, not to assign tasks to specific SMs. Perhaps because of this limitation, to our knowledge, no prior real-time GPU management system has attempted to use ERP.

▶ **Benefit 3.** *AMD GPUs natively allow partitioning kernels among compute units.*

Compared to the limited options on NVIDIA systems, compute-unit management on recent AMD GPUs is vastly more powerful, leading to Benefit 3. Starting with the fourth generation of the GCN architecture (often referred to by the code name *Polaris*), AMD GPUs allow each queue of kernels to be associated with a *compute-unit mask* – simply a set of compute units upon which kernels in a queue are allowed to execute [11]. This offers an immediate advantage over the prior SM-partitioning systems on NVIDIA GPUs because the compute-unit mask is completely transparent to kernel code, meaning that general-purpose computations on AMD GPUs can be partitioned among compute units without requiring any source-code modification.

[5] This is the maximum number of in-flight threads according to the GCN documentation [10], but, as discussed in Sec. 4, we only ever observed a maximum of 2,048 in-flight threads in practice.

Unfortunately, AMD does not provide a function for setting the compute-unit mask in its HIP API, but we were able to add such a function without too much effort. In Sec. 4, we describe our implementation, and demonstrate the efficacy of the compute-unit-mask mechanism for improving predictability of GPU kernel response times.

Cache hierarchy in GCN GPUs. Readers familiar with NVIDIA GPUs have likely already noticed the difference between the cache hierarchy shown in Fig. 1 and the layout in NVIDIA's GPUs. Like NVIDIA GPUs, AMD GPUs contain an L2 cache that is shared among all CUs, but the picture is slightly more complicated when it comes to L1 caches. In NVIDIA GPUs, each SM contains its own L1 data and instruction cache, but in GCN GPUs, each CU only contains its own L1 data cache. The L1 instruction cache is instead shared by a cluster of four CUs, as well as a separate L1 scalar data cache, which is a read-only cache used by the scalar unit.

This cache hierarchy means that although CU partitioning implies L1 data cache isolation, certain CU assignments may still be subject to L1 instruction cache interference. In practice, we saw little observable impact of this in our experiments, which we explain in more detail in the following section.

4 Case Studies

We next describe our experiments using the ROCm software stack on AMD GPUs, including our modifications to ROCm and experimental results. Naturally, any practical usage of a large software stack like ROCm is bound to encounter unforeseen complications, a few of which we also discuss in this section. All of our source code is available online, including modified ROCm code,[6] scripts and source-code patches for PyTorch,[7] and a microbenchmarking framework.[8]

4.1 Our Test Platform

We conducted our experiments on a system running Ubuntu 18.04 using Linux kernel version `5.4.0-rc8+`. Our host system contains 16 GB of memory and an Intel Xeon E5-2630 CPU, with eight physical cores supporting two hardware threads each.

We conducted our experiments using an AMD RX 570 GPU. The RX 570 is a mid-range GPU from AMD, costing approximately 180 US Dollars at the time of writing. We used an RX 570 containing 8 GB of GDDR5 memory, but models with 4 GB of memory are also available. The choice of ROCm-compatible platforms is limited, and most ROCm-capable systens will be similar to ours for the following reason:

▶ **Drawback 1.** *AMD's ROCm platform only supports Linux and discrete GPUs.*

Drawback 1 is easily learned by reading the list of supported systems on ROCm's website [12]. As its name implies, ROCm (again, *Radeon Open Compute*) seems motivated by an effort to recapture a portion of the compute-oriented market share currently dominated by NVIDIA. This has the implication that platforms less-suitable for high-performance computing are a lower priority at best.

[6] Modified ROCm code: `https://github.com/yalue/rocm_mega_repo/tree/hip_modification_only`.
[7] PyTorch scripts and patch: `https://github.com/yalue/ecrts_2020_pytorch_experiment`.
[8] Microbenchmarking framework: `https://github.com/yalue/hip_plugin_framework`.

We grant that the lack of Windows or MacOS support is unlikely to be a problem for researchers hoping to experiment with open-source software. Unfortunately, constraining support to discrete GPUs is a bigger limitation. Specifically, size and power constraints sometimes render discrete GPUs unsuitable for embedded applications, limiting the ability to directly test AMD-based GPU research in the real world. In contrast, Drawback 1 is not a problem for researchers choosing to target NVIDIA GPUs. Unlike AMD, NVIDIA offers several embedded-oriented GPUs, and its *Jetson* and *DRIVE* embedded platforms have been popular in real-time research [3, 6, 18, 22, 35, 42].

Currently, ROCm's kernel-level components such as the `amdgpu` or `amdkfd` Linux device drivers generally *do* support AMD's APUs (Application Processing Units), which include an integrated GPU, but AMD's APUs seem geared towards laptops or budget-oriented PCs rather than embedded applications.

4.2 Working with ROCm Source Code

While Sec. 4.1 focuses on our base hardware platform, it contains less information about our software choice apart from the underlying operating system. This is because our choice of the base ROCm software version and configuration bears a more in-depth explanation, which we cover in the following paragraphs. Recall that our proof-of-concept objective is to enable transparent compute-unit management in ROCm programs, but we can accomplish this in several different ways, depending on which of the "layers" depicted in Fig. 2 we wish to modify.

Our software platform is ROCm version 3.0. The source code for ROCm components (e.g., HCC, HIP, etc.) are contained in separate repositories, while a central ROCm repository merely contains a list of revisions required for each ROCm version. We used this list to obtain an unmodified copy of the ROCm 3.0 source code.[9]

ROCm version 3.0 is up-to-date at the time of this paper's submission, but the code has undergone several revisions and releases since we began working. ROCm was currently on version 2.6 when we began our study of AMD GPUs in mid-2019, so it has already undergone four major revisions by the time of this paper's writing. This trend indicates a second potential drawback of conducting real-time research on AMD GPUs:

▶ **Drawback 2.** *Large portions of the ROCm code base undergo frequent, major changes.*

One can hardly fault AMD for the fact that ROCm continues to be under active development. The fact that continually updating the software is desirable for most users does not, however, negate the inconvenience for researchers. To date, our work has encountered two concrete consequences of the rapid pace of change in ROCm code.

First, our modifications target the HIP programming language, which in turn targets the HCC compiler. However, due to Drawback 2 the HCC compiler was announced as deprecated in March 2019, with support supposedly ended in January 2020. For the time being, HIP code still depends at least in part on the HCC compiler, but AMD's apparent plan is to replace all remaining uses of HCC with the AMDGPU support already present in the LLVM compiler framework [23].

Unfortunately, when the removal of HIP's dependency on the HCC compiler is complete, our CU-management modifications to HIP are likely to increase in complexity. This is due to the fact that HCC already contains an API for managing CU masks, so adding a

[9] The exact list we used can be found at `https://github.com/RadeonOpenCompute/ROCm/blob/roc-3.0.0/default.xml`.

CU-mask-management function to HIP does not currently require modifications to any lower levels of the ROCm stack. This would not be the case with the AMDGPU LLVM backend, because it does not currently contain functions for managing CU masks. So, supporting CU-management functionality in HIP will, in the future, also require adding CU-management APIs to LLVM. While this is an added difficulty, access to source code means that it should at least remain possible.

A less major but equally time-consuming aspect of Drawback 2 is the effort required to work across the many components involved in the ROCm code base. In fact, a lack of clear documentation eventually caused us to abandon our efforts to compile the entire ROCm code base from source. Instead, we now install ROCm from AMD's pre-built debian repositories, and replace individual components with new versions we compile from source.[10] At one point, AMD maintained a repository of scripts capable of compiling a standalone version of ROCm,[11] but at the time of writing these scripts were last updated for ROCm 2.0, released in February 2019. While these scripts may provide a useful point of reference, old install scripts are a poor substitute for up-to-date human-readable instructions, especially when, as we have learned first-hand, questions arise about compatibility between various ROCm components' compile-time options, assumptions regarding installation directory structures, etc.

In fact, difficulties with ROCm's documentation are not limited to code compilation:

▶ **Drawback 3.** *There is no centralized, official source of ROCm documentation.*

Even though some may argue that Drawback 3 is mitigated by source-code availability, the entirety of ROCm source code is sufficiently large to make it an impractical source of "documentation." The userspace ROCm code required to run HIP contains over 200,000 lines of C and C++, and even that requires generous omissions from the overall line count. For example, there are many additional libraries and programming languages (including OpenCL) that are included in ROCm but not required to run basic HIP programs. The code relevant to our work is distributed among the components shown in Fig. 2 as follows:

- **HIP**: Roughly 46,000 lines, not including tests, samples, the `hipify` utility, or code for NVIDIA GPUs.
- **HCC**: Roughly 110,000 lines, only counting AMD-specific additions to the base LLVM code. Including all of the LLVM and `clang` compiler code brings the total closer to 1,375,000 lines.
- **HSA API and Runtime**:[12] Roughly 63,000 lines.
- **Drivers**: Roughly 223,000 lines. Of these, 23,000 are in the compute-specific `amdkfd` driver, and remainder are in the `amdgpu` driver, which also handles graphics.

So, even though code can serve as a definitive source of technical information, it is a poor source for high-level instructions or best practices, especially if it requires reading hundreds of thousands of lines.

To our knowledge, there is not a definitive "first-stop" location for ROCm documentation. Instead, there are documents from several different sources that may be useful to researchers working with the ROCm stack:

[10] We document our procedure in the README file with the code linked in Footnote 6.
[11] `https://github.com/RadeonOpenCompute/Experimental_ROC`
[12] We included both the ROCR-Runtime (`https://github.com/RadeonOpenCompute/ROCR-Runtime`) and ROCT-Thunk-Interface (`https://github.com/RadeonOpenCompute/ROCT-Thunk-Interface`) code in this number.

1) The HSA documentation [19] defines the HSA interface and communication protocol supported by the ROCm userspace libraries.
2) The ROCm website [12] contains an overview of the software system, but seems mostly geared towards "consumers" looking to install an unmodified version.
3) The GCN architecture manuals [14] and white papers [10, 11, 13] provide information about AMD GPU hardware.
4) The LLVM AMDGPU documentation [23] describes the LLVM backend for compiling code targeting AMD GPUs. This includes information about the executable-file format and HSA interface, and is likely to be useful for researchers looking to modify or understand low-level details.

In summary, a reasonable amount of documentation exists, but ROCm offers no close analogue to a comprehensive document like NVIDIA's CUDA programming guide [16]. As mentioned, the lack of compilation instructions for the entire ROCm toolchain is one such example of Drawback 3, and instructions on the main ROCm website assume that users will install ROCm from the pre-compiled repositories [12]. Another facet of Drawback 3 is the fact that documentation may be updated in a piecewise manner, leading to confusing information such as webpages that still recommend using HCC[13] without noting that HCC is deprecated. Of course, the existence of our modifications to ROCm is itself a testament to the fact that none of the documentation-related difficulties are insurmountable.

4.3 Our Modifications to ROCm

We chose to add several different CU-management interfaces to ROCm, by modifying different layers of the stack shown in Fig. 2. Our modifications fell into two categories: we modified the `amdkfd` driver and HCC driver to add two different mechanisms for specifying an application-wide default CU mask, and we also added a new function to the HIP API that enables an application to set the CU mask associated with a HIP stream. While the evaluation we present in Sec. 4.4 only requires our HIP modification, our additions to HCC or the `amdkfd` driver may remain useful for setting process-wide CU masks without modifying application code. In this subsection, we describe all three modifications.

Modifications to the `amdkfd` driver. The `amdkfd` driver is responsible for setting up memory-mapped queues of GPU commands, which are directly shared between userspace and GPU hardware. The HSA API, mentioned in Sec. 3, issues commands to the GPU by writing them into the in-memory queues. User-level applications can control additional settings, such as CU masks, by issuing various ioctl commands to the driver. Even though existing ioctl commands are capable of changing existing queues' CU masks, being limited to per-queue settings makes partitioning an entire GPU-using application difficult, because the application may create new queues at any time in its execution.

To provide a more convenient interface, we added a new feature to the `amdkfd` driver: a `SET_DEFAULT_CU_MASK` ioctl. As the name of this command implies, it causes the driver to apply a default CU mask to any queues the calling process creates in the future. Implementing this change required straightforward changes to the driver code.[14] Using the modification requires a simple modification to the GPU-using applications themselves – they must be modified to issue our new ioctl command at the start of execution, before creating any ROCm queues.

[13] This can still be seen on `https://rocm.github.io/languages.html` as of February 2020.
[14] We included a patch and further instructions in the code linked in Footnote 6.

Modifications to HCC. The driver modification outlined above provides one possible method for specifying a default process-wide CU mask, but it suffers from a couple of usability drawbacks. First, as mentioned, one must issue the new ioctl command from within the context of the GPU-using process, which, while a simple change, nonetheless requires modifying application source code. Second, setting a CU mask in the driver may lead to a discrepancy between what higher-level applications "think" a queue's CU mask is, and what the underlying CU mask actually has been set to. Our modifications to HCC allow users to specify a default CU mask with neither of these two drawbacks.

Since HCC is implemented in userspace, we were able to implement a default-CU-mask modification using a well-known software feature: environment variables. Internally, HCC maintains a copy of each queue's expected CU mask, and uses this additional bookkeeping to potentially share a smaller pool of underlying driver-managed queues among multiple higher-level "queue" abstractions (the details of which, fortunately, are not necessary to understand our modification). In order to implement our environment-variable interface, we found the C++ constructor responsible for initializing one of the aforementioned HCC-managed queues, and modified it to initialize both the CU mask of the actual underlying HSA queue as well as the internal HCC-managed copy of the CU mask based on the contents of the (arbitrarily named) `HSA_DEFAULT_CU_MASK` environment variable. Using this modification is extremely simple: one only needs to set the environment variable before creating a process.

We are aware of no applications that do so, but it is conceivable that an application could circumvent our environment variable by directly interacting with the HSA API layer "below" HCC, as shown in Fig. 2. If necessary, we could even prevent such behavior by modifying AMD's HSA API implementation (known as `ROCR-Runtime` in the ROCm source code) in a manner similar to our HCC change, but, like the driver-based modification, this would risk HCC's bookkeeping becoming inconsistent with actual underlying CU-mask settings.

Modifications to HIP. The modification to HCC is convenient to use in practice, but offers little flexibility within the context of a single process, where we may wish to set different CU masks for different streams (this ended up being necessary for our PyTorch experiments, discussed in Sec. 4.4). To address this need, we chose to add a `hipStreamSetComputeUnitMask` function to the HIP API that sets the CU mask associated with an individual HIP stream.

Each HIP stream is implemented on top of a single underlying HCC-managed queue of GPU commands. Internally, this HCC-managed queue is represented by a C++ object that already provides a `set_cu_mask` method, so we implemented our new HIP function simply by calling this underlying method.

Our change to HIP was straightforward due to HIP's use of HCC, but recall that HIP supports alternate backends. One obvious example is HIP's `nvcc` backend, which targets NVIDIA GPUs by wrapping the CUDA API. NVIDIA GPUs have no concept of hardware-supported CU masking (at least judging by public documentation), so implementing `hipStreamSetComputeUnitMask` for NVIDIA is clearly impossible. As discussed in Sec. 4.2, we anticipate additional difficulties when HIP fully switches to a `clang`-based backend, since we will no longer be able to rely on the availability of the HCC-provided `accelerator_view` interface that provides CU-mask-management functions.

Final words on implementation. All of our CU-management implementation approaches worked as expected. Even though the remainder of our evaluation only required modifying HIP, the fact that so many approaches were viable, even for such a minor feature, shows

```
import torch
stream = torch.cuda.Stream("cuda:0") # Standard PyTorch API to create a stream
stream.set_cu_mask(0xffff0000)       # Invoke our new function
with torch.cuda.stream(stream):      # Make subsequent GPU code use the stream
    # ... evaluate the network, etc
```

Figure 5 Setting a CU mask in a PyTorch script. Note that PyTorch uses the term "cuda" regardless of the fact that it's running on an AMD GPU.

how an open-source GPU-computation software stack can provide significant flexibility for research purposes. Recall that typical research using NVIDIA GPUs is only able to modify one "layer" in NVIDIA's analogue to Fig. 2: the user program!

In the remainder of this section, we discuss how we applied our modifications to improve timing predictability in a PyTorch application, and describe some additional details about AMD GPUs that we encountered in the course of our experiments.

4.4 Evaluation using PyTorch

We carried out a case study using PyTorch [4] to validate our approach using a complex piece of real-world GPU-accelerated software. We chose PyTorch in part due to its popularity; in 2019, it was used in the majority of papers presented at every major computer-vision conference [21]. Additionally, we wanted to design an experiment that accurately reflects a reality where a single GPU may need to concurrently process multiple independent streams of data, e.g., on an autonomous vehicle with multiple cameras or sensors.

PyTorch, as its name may imply, focuses on providing an idiomatic Python interface for creating, training, and evaluating neural networks. This is a convenient interface for computer-vision and AI researchers, but the level of abstraction involved in a Python interface presents a hurdle for real-time management, where fine-grained control is more desirable.

Required modifications to PyTorch. Given PyTorch's complexity and level of abstraction, it comes as no surprise that enabling CU mask support required additional modifications to the PyTorch source code. PyTorch is implemented using CUDA, but ships with a script for converting its CUDA source code to HIP. After obtaining the PyTorch source code and running the provided HIP-conversion script, we were able to add functionality for setting CU masks to the PyTorch API.[15] Fortunately, PyTorch already provides an interface for working with "CUDA" streams,[16] so we were able to add another function to PyTorch's stream-management API to call the `hipStreamSetComputeUnitMask` function we added to HIP. Fig. 5 shows a snippet of a Python script illustrating the usage of the added functionality.

But why go to the trouble of modifying PyTorch source code, when our environment-variable-based HCC modification should already suffice to cause PyTorch to use a CU mask? In fact, our initial attempt was to do exactly that – and indeed, the environment-variable approach appears to correctly limit the CU mask for any given PyTorch process. Unfortunately, this was insufficient for our later experiments requiring contention for GPU resources. We found that AMD GPUs schedule work from separate processes in a different

[15] Our specific procedure for working with PyTorch source code, along with a patch containing our modifications, is included in the repository linked in Footnote 7.

[16] In order to maintain compatibility with existing scripts, PyTorch's user-facing Python API continues to use the term "CUDA" even when compiled for HIP.

manner (namely, one that seems to prevent or reduce concurrent execution) than work coming from multiple streams within a single process. We observed similar behavior on NVIDIA GPUs in our prior work [3, 35, 41], so this analogous finding on AMD GPUs did not come as a particular surprise, but we chose to leave further investigation for future work. In the meantime, we found that using multiple threads and streams from within a single PyTorch script was sufficient for this paper, which only requires generating observable contention between competing GPU workloads.

Our test application. Our experiments were based on an example provided by PyTorch that trains a neural network to classify handwritten digits in the MNIST dataset.[17] We modified the original script in several ways. First, we changed it to only perform classification using pre-trained network weights instead of training a new network. Second, we added an optional competitor thread[18] to provide additional contention for GPU resources. When active, the competitor thread evaluates random data using a separate neural network. The competing network was also based on the MNIST network, but used larger layer sizes and a larger number of layers to generate more GPU computations. Finally, we updated the manner in which the script loads the MNIST image data, which defaults to loading data on demand. To avoid noise involving data transfers or reads from storage, we pre-buffer all of the image data in GPU memory before conducting any measurements.

We instrumented the primary thread to record the duration of each "iteration," consisting of a single forward pass of the network, which classifies a batch of 64 different 32x32-pixel grayscale input images. We determined the duration by recording the time before the first layer of the network was evaluated, and again after a `stream.synchronize()` call returned, indicating the completion of all GPU operations. While some noise is certainly involved when recording times in Python code, any such noise will be negligible compared to the durations being measured in our application. Additionally, our script does not record times for the first several iterations while the GPU "warms up," which is a common practice in GPU research to avoid measurement noise while code or data is lazily loaded onto the device. In total, we record 36,504 time samples for every scenario (this arbitrary-seeming number is due to a combination of factors, including warm-up iterations and the way in which the MNIST dataset is subdivided into batches of 64).

Verifying CU-mask functionality. Our first experiment was designed to verify our ability to successfully limit a task's set of available CUs, and to test whether partitioning CUs improves response-time predictability in the presence of an adversarial workload. Fig. 6 shows the results from four scenarios. The first scenario is represented by the leftmost curve, and shows the response-time distribution when our single MNIST-evaluation PyTorch thread is allowed to execute on all 32 CUs with no competitor. As expected, this configuration exhibits the fastest response times, with a maximum observed response time of 5.02 ms. The second curve shows the response times of the same workload, but in this case it was limited to only use 16 CUs. As seen in Fig. 6, reducing the available CUs from 32 to 16 only increased observed response times by roughly one millisecond, indicating that, absent a competing

[17] Our experimental scripts are included in the link in Footnote 7. We based our experiments on the example from `https://github.com/pytorch/examples/tree/master/mnist`.

[18] Even though Python's threads are notorious for their lack of concurrency when using C/C++ modules, PyTorch takes measures to release the necessary Python locks within its C++ code.

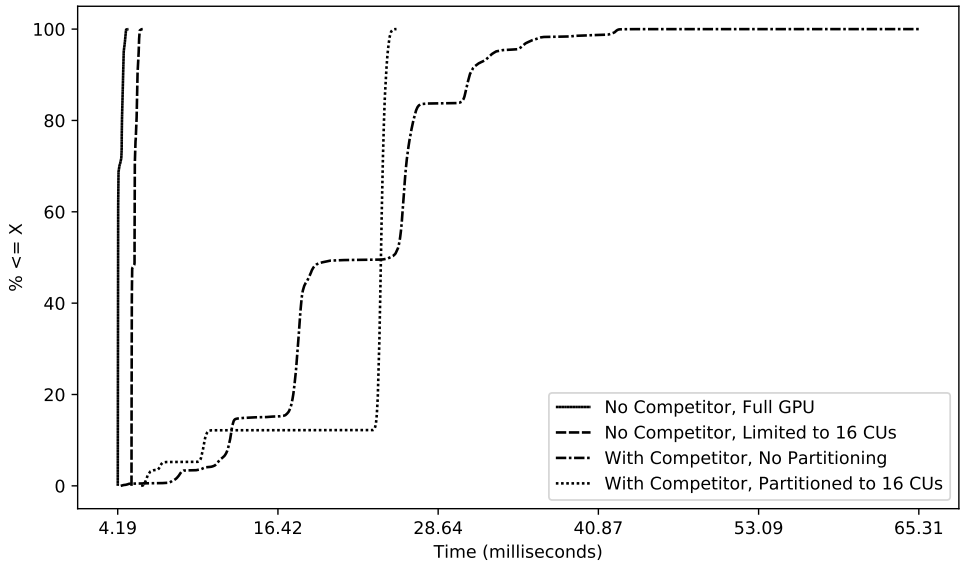


Figure 6 CDFs of times required for a forward pass of PyTorch's MNIST example, with and without a competitor.

workload, overheads or CPU operations comprise a significant portion the MNIST example network's overall execution time (we would expect the slowdown due to the reduction in resources to increase in proportion to the amount of GPU computation required).

Behavior changes drastically with the introduction of a competing thread. The curve in Fig. 6 with alternating dots and dashes shows the distribution of iteration times when the competitor thread is running, and both threads are allowed to access all 32 CUs on the GPU. The dotted curve shows the same scenario, but with the competitor and primary thread partitioned to separate groups of 16 CUs. Both distributions exhibit a very similar mean time of approximately 22 milliseconds, but the worst-case observed time without partitioning, 65 milliseconds, is significantly slower than the time when partitioning is applied, where the worst-case observed time was 25 milliseconds – only slightly worse than the same scenario's median time. While this improvement provides sufficient motivation to use our CU-masking API, the significant increase in mean and median times compared to the times without a competitor remains a concern for future work, as it indicates that threads also spend significant time contending for non-CU resources. Some of this contention may be related to Python threads or PyTorch internals, but there may also be other sources of contention for GPU resources that we can prevent with further modifications to ROCm.

4.5 Additional ROCm Experiments

In addition to our evaluation using PyTorch, we also conducted several microbenchmark experiments using a plugin-based framework we developed for earlier work on NVIDIA GPUs [34], and ported to HIP for this paper.[19] Porting our code to HIP required disabling two NVIDIA-specific features only available in kernel inline assembly: reading the `globaltimer` register, and obtaining the ID of the SM of the currently executing thread block. Losing access to these features is unfortunate, as both were essential for our prior experiments to infer

[19] Source code for this framework is linked in Footnote 8.

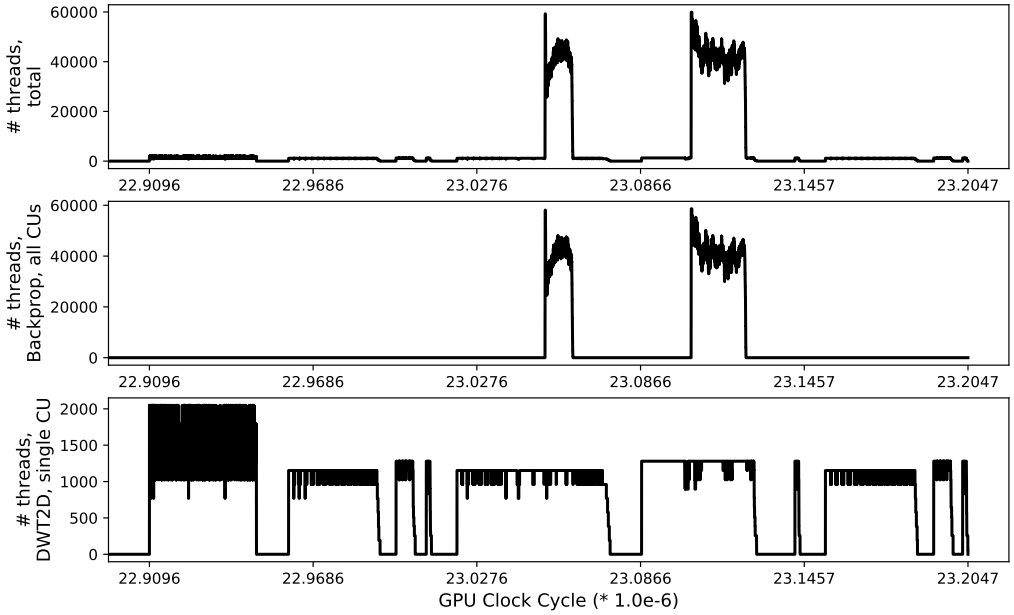


Figure 7 Timelines of the number of in-flight GPU threads during the concurrent execution of two benchmarks.

scheduling rules for NVIDIA GPUs. However, even without them, the highly instrumented microbenchmarks remain useful for observing details at a finer granularity than a large system like PyTorch can provide. Most of the experiments we conducted with this framework simply confirmed the findings we already discussed in Sec. 4.4, but presenting them enables us to report a few additional results.

Revisiting in-flight threads per CU. Recall from Sec. 3.2 that, according to GCN documentation, AMD GPUs support 2,560 in-flight threads per CU. Our experiments, however, contradicted this number. We never observed 2,560 in-flight threads per CU in any experiment, regardless of thread block size, workload, or number of available CUs. Instead, we observed at most 2,048 in-flight threads in all situations. Fig. 7 contains one such example.

Fig. 7 was produced by our plugin framework concurrently running two microbenchmarks, `DWT2D` and `backprop`, that we adapted for our framework from the Rodinia benchmark suite [9]. The choice of these specific microbenchmarks was somewhat arbitrary (as we can observe the same effects with any of the others), however they issue kernels at slightly more irregular intervals than other simpler microbenchmarks (i.e., vector add), potentially allowing us to observe a wider variety of interactions. As with all of the plugins in our framework, we instrumented these microbenchmarks' kernels to record the GPU clock cycle when the first thread in each block starts running, as well as the cycle before the last thread in each block returns, and used this information to compute the number of in-flight GPU threads over the course of the microbenchmarks' execution.

Fig. 7 shows three timelines, each of which represents the active number of GPU threads at a given clock cycle. The top timeline shows the total number of active threads, and the other two show the number of threads running on behalf of the two microbenchmarks. These timelines contain two clear pieces of evidence contradicting a 2,560-thread limit. First, if each of the GPU's 32 CUs indeed supported 2,560 in-flight threads, we would expect to see a maximum of 81,920 concurrent total threads in the top timeline of Fig. 7, but we do

not. Instead, the maximum total number of observed running threads never exceeds 65,536, consistent with a 2,048-thread limit (in fact, this particular experiment never even exceeds 60,000 total concurrent threads, but we observed a clear ceiling of 65,536 threads in many other microbenchmark experiments). The second major piece of evidence is shown in the timeline for `DWT2D` at the bottom of Fig. 7. We limited `DWT2D` to a single CU during this experiment, and its number of in-flight threads clearly never exceeds 2,048. (Note that its vertical axis uses a different scale.) Furthermore, `DWT2D` was using 256-thread blocks, which evenly divides a hypothetical 2,560-thread limit, so the lower-than-expected thread count cannot be a consequence of incompatible block sizes.

This observation likely relates to Drawbacks 2 and 3, where an aspect of the internal GCN architecture may not yet be adequately documented. Regardless of the reason, this observation underscores an important point for real-time researchers: *important hardware details may not be revealed through open-source software or public documentation.* This applies equally to NVIDIA GPUs, so we do not label this as an explicit "drawback" of working with AMD. It is even possible that the "true" 2,560-thread limit can be reached in cases not covered by our microbenchmarks. However, differences between documentation and actual behavior could lead to a fundamentally unsafe model of a real-world system, especially when the difference is in a detail as fundamental as the number of supported parallel threads.

L1 instruction cache impact. For our final experiment, we revisit the question raised in Sec. 3 of the impact of sharing the L1 instruction cache between clusters of four CUs. As mentioned, we were unable to observe any L1 instruction cache interference in our experiments, even in the presence of an attempted adversarial workload, a kernel with 300 KB of code written using inline GCN assembly. There are two factors we suspect make GPU L1 instruction-cache interference only a minor concern in practice:

First, it is reasonable to assume that an L1 cache-line fetch retrieves multiple instructions. This, combined with the fact that well-optimized GPU kernel code tends to have few branches, means that GPU kernel code is unlikely to generate many L1 misses. Second, GPU kernels are unlikely to put much pressure on instruction caches simply by virtue of the small size of typical kernel code.

The second of these two points is supported by the data shown in Fig. 8. In order to obtain this data, we downloaded all of the pre-compiled packages from AMD's ROCm 2.6 respository, and, thanks to the LLVM documentation mentioned in Sec. 4.2 [23], wrote a program that parses ELF binaries containing GPU kernel code in order to report the size of each kernel. In total, we were able to find thousands of kernels in this repository, primarily from GPU-accelerated machine-learning libraries.

As shown in Fig. 8, the average kernel size in AMD's repositories was roughly 2 KB, meaning that up to sixteen such kernels could hypothetically execute concurrently without exhausting the 32 KB L1 instruction cache – an unlikely situation on a single cluster of four CUs. Since many kernels require executing hundreds or thousands of thread blocks, it is far more likely that only a single kernel would be running at a time on each CU cluster.

4.6 AMD GPU Performance

While less relevant for research involving timing predictability, we recognize that fast hardware is still necessary in real applications, and is likely to be of interest to readers. To this end, we ran our PyTorch script from Sec. 4.4 on several hardware platforms, without enabling CU partitioning or the competing thread. The resulting data is summarized in Tbl. 1. Unsurprisingly, one of NVIDIA's flagship GPUs, the Titan V, exhibited the fastest

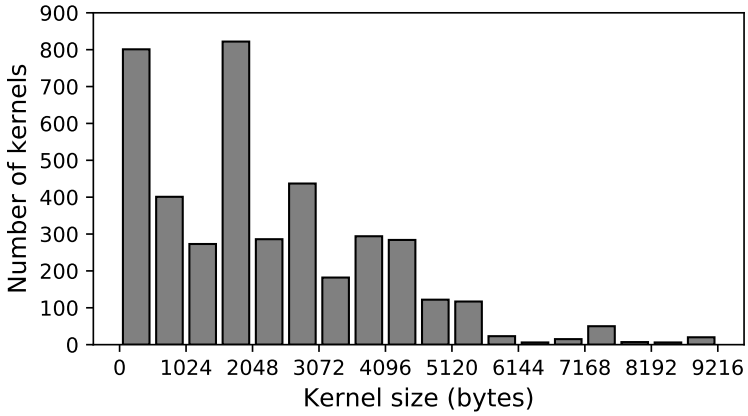


Figure 8 Distribution of GPU kernel sizes found in AMD's pre-compiled ROCm 2.6 repository.

Table 1 Times needed for a single forward pass of PyTorch's MNIST example on several different hardware devices. All times are in milliseconds.

	Min	Max	Median	Mean	Std. Deviation
NVIDIA Titan V	0.51	5.79	0.51	0.52	0.08
NVIDIA GTX 1060	1.40	1.63	1.42	1.42	0.01
NVIDIA GTX 970	1.38	2.77	1.40	1.40	0.02
AMD RX 570	4.19	5.02	4.21	4.32	0.18
CPU (Intel Xeon 4110)	5.54	18.43	8.60	8.40	1.78

performance by far. The other two NVIDIA GPUs also outperformed the RX 570 used in this paper, though even the RX 570 was faster than running the neural network solely on a CPU. It is possible that the RX 570's slower performance relative to its NVIDIA counterparts can be largely attributed to ROCm's lack of maturity compared to CUDA – a possibility strengthened by the fact that the RX 570 performs similarly to the NVIDIA GTX 1060 in graphical workloads [26]. Finally, even though AMD GPUs currently lag behind NVIDIA in our PyTorch benchmark, the slower performance is unlikely to prevent accurately evaluating *management principles*, which remain the focus of real-time research.

5 Conclusion

In this paper, we make the argument that the existence of an alternate platform – AMD GPUs – has the potential to reshape real-time GPU research. The thrust of our argument comes from two key points. First, we described how almost all prior real-time GPU research has, to some extent, worked around the limitations of NVIDIA's closed-source platform, often at the expense of long-term applicability. Second, as we demonstrated using case studies, many of the implementation drawbacks that hamper research on NVIDIA GPUs do not apply to AMD (despite AMD having its own collection of downsides).

In future work, we plan to continue our investigation of AMD GPUs, and, ideally, hope to implement portions of prior NVIDIA-focused real-time research using AMD's open-source software. We also plan to investigate the question of optimal CU assignment in more detail, as well as the impact of contention for non-CU hardware components in software running on AMD GPUs.

Establishing the "best" platform for any complex system will always involve a set of tradeoffs and personal preferences, but it is not necessary to prove that a system is the "best" in order for it to warrant further study. In a research setting, the question is still open as to whether the drawbacks discussed here are an acceptable trade for the benefits offered by an open-source GPU software stack, and the answer will likely vary from one project to another. Nonetheless, with this paper we have at least demonstrated that AMD GPUs warrant further consideration, and, despite some difficulties, offer significant benefits for developers looking for greater control over safety-critical software and hardware.

It is not our goal to provide a definitive answer to the NVIDIA-vs.-AMD question, or even to take a side in the battle. In fact, not only do we refrain from taking a side – we hope the battle for GPU dominance continues! The competition itself has already led to a major hardware vendor investing significantly in open-source GPU software, and it would be wonderful if others would follow AMD's example.

References

1. Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.
2. Waqar Ali and Heechul Yun. Protecting Real-Time GPU Kernels on Integrated CPU-GPU SoC Platforms (Artifact). *Dagstuhl Artifacts Series*, 4(2):3:1–3:2, 2018. URL: `http://drops.dagstuhl.de/opus/volltexte/2018/8971`.
3. Tanya Amert, Nathan Otterness, James Anderson, and F. D. Smith. GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *IEEE Real-Time Systems Symposium (RTSS)*, 2017.
4. PyTorch Authors. PyTorch. Online at `https://pytorch.org/`, 2020.
5. Can Basaran and Kyoung-Don Kang. Supporting preemptive task executions and memory copies in GPGPUs. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.
6. 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.
7. Nicola Capodieci, Roberto Cavicchioli, Paolo Valente, and Marko Bertogna. SiGAMMA: Server based integrated GPU arbitration mechanism for memory accesses. In *International Conference on Real-Time Networks and Systems (RTNS)*, pages 48–57, 2017.
8. Roberto Cavicchioli, Nicola Capodieci, Marco Solieri, and Marko Bertogna. Novel methodologies for predictable CPU-to-GPU command offloading. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2019.
9. Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.
10. AMD Corporation. AMD graphics core next (GCN) architecture. Online at `https://www.techpowerup.com/gpu-specs/docs/amd-gcn1-architecture.pdf`, accessed September 2019., 2011.
11. AMD Corporation. Radeon: Dissecting the polaris architecture (white paper). Online at `https://www.amd.com/system/files/documents/polaris-whitepaper.pdf`, acessed September 2019., 2016.
12. AMD Corporation. ROCm, a new era in open GPU computing. Online at `https://rocm.github.io/`, 2016.
13. AMD Corporation. Radeon's next-generation Vega architecture. Online at `https://www.techpowerup.com/gpu-specs/docs/amd-vega-architecture.pdf`, accessed September 2019., 2017.

14 AMD Corporation. "Vega" instruction set architecture: Reference guide. Online at `https://developer.amd.com/wp-content/resources/Vega_Shader_ISA_28July2017.pdf`, 2017.

15 AMD Corporation. Introducing RDNA architecture. Online at `https://www.amd.com/system/files/documents/rdna-whitepaper.pdf`, accessed February 2020., 2019.

16 NVIDIA Corporation. CUDA C programming guide. Online at `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`, 2019.

17 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.

18 Björn Forsberg, Andrea Marongiu, and Luca Benini. GPUguard: Towards supporting a predictable execution model for heterogeneous SoC. In *Proceedings of the Conference on Design, Automation & Test in Europe*, 2017.

19 HSA Foundation. HSA platform system architecture specification. Online at `http://www.hsafoundation.com/?ddownload=5702`, 2018.

20 HSA Foundation. HSA runtime programmer's reference manual. Online at `http://www.hsafoundation.com/?ddownload=5704`, 2018.

21 Horace He. The state of machine learning frameworks in 2019. Online at `https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/`, October 2019.

22 Přemysl Houdek, Michal Sojka, and Zdeněk Hanzálek. Towards predictable execution model on ARM-based heterogeneous platforms. In *International Symposium on Industrial Electronics (ISIE)*, 2017.

23 LLVM Compiler Infrastructure. User guide for AMDGPU backend. Online at `https://llvm.org/docs/AMDGPUUsage.html`, 2019.

24 Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan (Raj) Rajkumar. Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

25 Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

26 Will Judd. AMD Radeon RX 570 benchmarks: A capable 1080p workhorse. Online at `https://www.eurogamer.net/articles/digitalfoundry-2019-05-01-amd-radeon-rx-570-benchmarks-7001`, June 2019.

27 Shinpei Kato, Karthik Lakshmanan, Aman Kumar, Mihir Kelkar, Yutaka Ishikawa, and Ragunathan Rajkumar. RGEM: A responsive GPGPU execution model for runtime engines. In *IEEE Real-Time Systems Symposium (RTSS)*, 2011.

28 Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *USENIX ATC*, 2011.

29 Michael Larabel. AMDKFD is present for linux 3.19 in open-source HSA start. *Phoronix.com*, 2014. Online at `https://www.phoronix.com/scan.php?page=news_item&px=MTg1MzE`.

30 Haeseung Lee and Mohammed Abdullah Al Faruque. Run-time scheduling framework for event-driven applications on a GPU-based embedded system. In *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*, 2016.

31 Hyeonsu Lee, Jaehun Roh, and Euiseong Seo. A GPU kernel transactionization scheme for preemptive priority scheduling. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018.

32 NVIDIA. CUDA toolkit. Online at `https://developer.nvidia.com/cuda-toolkit`, 2019.

33 NVIDIA. NVIDIA cuDNN. Online at `https://developer.nvidia.com/cudnn`, 2019.

34 Nathan Otterness, Ming Yang, Tanya Amert, James Anderson, and F. D. Smith. Inferring the scheduling policies of an embedded cuda GPU. In *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2017.

35 Nathan Otterness, Ming Yang, Sarah Rust, Eunbyung Park, James Anderson, F.D. Smith Smith, Alex Berg, and Shige Wang. An evaluation of the NVIDIA TX1 for supporting

real-time computer-vision workloads. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

36 Sujan Kumar Saha. Spatio-temporal GPU management for real-time cyber-physical systems. Master's thesis, UC Riverside, 2018.

37 Puja Tayal. NVIDIA loses some discrete GPU market share to AMD. Online at `https://articles2.marketrealist.com/2019/06/nvidia-loses-some-discrete-gpu-market-share-to-amd/`, June 2019.

38 Uri Verner, Avi Mendelson, and Assaf Schuster. Batch method for efficient resource sharing in real-time multi-GPU systems. In *International Conference on Distributed Computing and Networking*. Springer, 2014.

39 Uri Verner, Avi Mendelson, and Assaf Schuster. Scheduling periodic real-time communication in multi-GPU systems. In *IEEE International Conference on Computer Communication and Networks (ICCCN)*, 2014.

40 Uri Verner, Assaf Schuster, Mark Silberstein, and Avi Mendelson. Scheduling processing of real-time data streams on heterogeneous multi-GPU systems. In *ACM International Systems and Storage Conference*, 2012.

41 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.

42 Ming Yang, Shige Wang, Joshua Bakita, Thanh Vu, F Donelson Smith, James H Anderson, and Jan-Michael Frahm. Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

43 Husheng Zhou, Guangmo Tong, and Cong Liu. GPES: A preemptive execution system for GPGPU computing. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

Turning Futexes Inside-Out: Efficient and Deterministic User Space Synchronization Primitives for Real-Time Systems with IPCP

Alexander Zuepke
RheinMain University of Applied Sciences, Wiesbaden, Germany
alexander.zuepke@hs-rm.de

Abstract

In Linux and other operating systems, futexes (fast user space mutexes) are the underlying synchronization primitives to implement POSIX synchronization mechanisms, such as blocking mutexes, condition variables, and semaphores. Futexes allow one to implement mutexes with excellent performance by avoiding system calls in the fast path. However, futexes are fundamentally limited to synchronization mechanisms that are expressible as atomic operations on 32-bit variables. At operating system kernel level, futex implementations require complex mechanisms to look up internal wait queues making them susceptible to determinism issues. In this paper, we present an alternative design for futexes by completely moving the complexity of wait queue management from the operating system kernel into user space, i.e. we turn futexes "inside out". The enabling mechanisms for "inside-out futexes" are an efficient implementation of the immediate priority ceiling protocol (IPCP) to achieve non-preemptive critical sections in user space, spinlocks for mutual exclusion, and interwoven services to suspend or wake up threads. The design allows us to implement common thread synchronization mechanisms in user space and to move determinism concerns out of the kernel while keeping the performance properties of futexes. The presented approach is suitable for multi-processor real-time systems with partitioned fixed-priority (P-FP) scheduling on each processor. We evaluate the approach with an implementation for mutexes and condition variables in a real-time operating system (RTOS). Experimental results on 32-bit ARM platforms show that the approach is feasible, and overheads are driven by low-level synchronization primitives.

2012 ACM Subject Classification Computer systems organization → Real-time operating systems; Software and its engineering → Mutual exclusion

Keywords and phrases Futex, Immediate Priority Ceiling Protocol, Critical Section, Monitor

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.11

1 Introduction and Motivation

A common technique to improve the performance of synchronization mechanisms is to avoid unnecessary system calls [3, 4, 14, 18, 31]. Ousterhout [27] observed for processor architectures of the 1980s that "*[. . .] operating system performance does not seem to be improving at the same rate as the base speed of the underlying hardware*". Today, the situation has not changed much: system calls are an order of magnitude slower than atomic operations. For current Intel CPUs, Al Bahra measures about 15 cycles for an atomic *compare-and-swap (CAS)* operation on an Intel Core i7-3615QM [1], while Soares and Stumm describe a system call overhead of around 150 cycles on an earlier Core i7 generation [33]. For the future, we can assume that system calls remain expensive due to pipeline flushes [33] and mitigation against processor design flaws such as *Meltdown* [21] and *Spectre* [19]. While we can assume that processor design flaws like Meltdown will be fixed in future processor generations, Spectre-like attacks on branch prediction by data cache side channels are expected to stay, along with the corresponding measures of mitigation [23]. Therefore, avoiding unnecessary system calls is still relevant today for efficient synchronization mechanism.

Futexes were originally conceived as a mechanism for *fast mutexes* in Linux [14]. The idea behind futexes is to acquire and release uncontended mutexes by just using atomic operations in the C library in user space. The kernel is only involved in handling blocking and unblocking of threads on contention. This design omits system calls into the operating system kernel in the fast path. Today, futexes are a generic *compare-and-block* mechanism in Linux and used to implement almost all blocking POSIX thread synchronization primitives. A further benefit of futexes is that no pre-registration of synchronization objects in the kernel is needed. The Linux kernel creates in-kernel objects such as wait queues on demand.

From the design point of a real-time system, the basic idea of futexes as an *optimization of the common case* also helps in mixed-criticality environments or when trying to improve *size, weight and power* (SWaP) considerations. If a real-time task finishes early, it is a desirable design goal to leave the remaining processing time to other non-real-time tasks, or to enable power saving strategies earlier. This principle applies to all kinds of optimizations such as processor caches, but only as long as an optimization also shows a deterministic behavior and one can analytically derive a proper *worst-case execution time (WCET)*.

However, design aspects that are important for the flexibility of futexes in Linux also bring problems for their deterministic behavior. In the Linux kernel, blocked threads are kept in a fixed-sized hash table, and previous research has shown that the futex hash table is vulnerable to cause interference to otherwise unrelated processes due to inadvertent or intentional hash collisions [38].

Alternative approaches to prevent this problem of *undesired resource sharing* involve pre-registration of synchronization objects. This allows indexing of wait queues by other means not susceptible to attacks, e.g. by using an index in a descriptor table [34]. Such a limitation is fully compliant to most programming environments for real-time applications, such as ARINC 653 for Avionics, which often require allocation of all resources at start time anyway, to prevent later resource exhaustion at runtime, but this also limits the flexibility of futexes for non-real-time workloads.

Another important design aspect of futexes is the use of 32-bit variables as protocol variables in user space. 32-bit variables were originally chosen because atomic operations on them are available on most major platforms supported by Linux. However, using 32-bit variables in futex-based synchronization protocols requires that all data needed to make a decision whether to wait or to wake up threads must be "compressed" into a single 32-bit variable. While this is possible for most POSIX user space synchronization mechanisms such as mutexes, condition variables, barriers, and semaphores, it is not the case for other, more complex synchronization mechanisms, such as message queues.

To address these problems of futexes in the context of real-time systems, while preserving their desirable properties, we propose to turn the futexes "inside-out" and to move the wait queue management *out* of the kernel *into* user space. An efficient mechanism for *short* critical sections is the enabling factor for queue manipulations, e.g. priority-ordered queues, which cannot be realized as simple atomic operations on modern hardware. Such a mechanism also removes the limitation of 32-bit protocol variables, allowing more complex waiting conditions. For this to work *efficiently* and *deterministically*, we need three key components:

(i) A reliable and efficient mechanism to prevent preemption in user space.

(ii) Fair spinlocks for mutual exclusion between other processor cores.

(iii) A light-weight mechanism to suspend and wake up threads.

The resulting design effectively resembles a *non-preemptive busy-waiting monitor* with *Mesa-style blocking condition variables* [10]. The main challenge of this design is that the overall performance of these mechanisms must be comparable to futexes. For a blocking mutex, we aim for a solution that avoids system calls in the fast path and requires system calls only

to suspend the calling thread in a mutex lock operation, or to wake up a waiting thread in an unlock operation. The baseline is given by a traditional approach of implementing synchronization mechanisms in the kernel, using dedicated system calls for all operations. Here, we must reach a similar level of performance in the worst case.

Components (i) and (ii) are well researched in the context of real-time systems. Specifically, in user space, temporarily disabling preemption while holding a spinlock is necessary to avoid the lock-holder preemption problem [26, 35]. To this end, efficient and predictable implementations of the *immediate priority ceiling protocol* (IPCP) [2, 37] are to be preferred over other non-real-time techniques of controlling preemption [13, 20, 22, 25]. In this paper, the design of component (iii) is based on the key idea of addressing threads directly. Mechanisms for $\mathcal{O}(1)$ look-up of threads by their ID are in fact readily available at OS kernel level, so we can leverage them to construct light-weight suspension and wake-up mechanisms targeting threads directly rather than introducing another level of indirection to look-up wait queues in the kernel. As result, the required kernel mechanisms to support the proposed monitors in user space are much simpler compared to futexes or a traditional approach. This effectively reduces the implementation effort and related WCET concerns inside the operating system kernel and moves the complexity to support common thread synchronization mechanisms into user space.

Our contributions in this paper are:

- A mechanism to suspend and to wake up threads in IPCP for an operating system using partitioned fixed-priority (P-PF) scheduling (Section 4).
- Non-preemptive busy-waiting monitors in user space (Section 5).
- An analysis of synchronization mechanisms w.r.t. their potential for optimization by reducing system calls and related WCET concerns (Section 6).
- An efficient implementation of blocking mutexes and condition variables based on the monitor that require at most one system call. Like futex-based synchronization mechanisms, no pre-registration of synchronization objects in the kernel is required (Section 7).
- An experimental evaluation of the approach in a research RTOS for 32-bit ARM processors to demonstrate the effectiveness of the proposed approach (Section 8).

2 Terminology and System Model

We assume a system comprising a contemporary 32-bit or 64-bit processor with one or more processor cores. On the system, an operating system kernel executes in *supervisor mode*, which is a privileged mode of the processor, while processes host code executing in a non-privileged *user mode*. The processor provides virtual memory or enforces memory protection, such that each process has its own isolated address space. Code in user mode uses *system calls*, a hardware trap mechanism, to call into the operating system kernel. The processor also provides general atomic operation on 32-bit variables based on *compare-and-swap (CAS)* or *load-linked/store-conditional (LL/SC)* instructions. The atomic operations additionally provide *acquire* or *release* semantics, or the processor provides explicit memory barriers to order memory accesses. We also assume that, on average, non-atomic operations (e.g. ALU- and normal load/store-operations) are much faster than atomic operations, and atomic operations are much faster than system calls.

We use the term *thread* instead of *task* when talking about schedulable entities of a process. *TCB* refers to the *thread control block*, which can comprise variables shared between kernel and user space. We denote the user space part of the TCB as *UTCB*. The identifier `SELF` points to the current thread's thread-local storage (TLS) data segment in user space, which also contains the UTCB.

We assume *partitioned fixed-priority* (P-FP) scheduling. We define the *scheduling priority* in the sense that higher values refer to higher priority level, as this is often done in RTOS implementations. The value `max_prio` defines the maximum priority level that a thread in a process can use, while `0` is the minimum priority level in the system.

For resource sharing on each single processor core, we use the *immediate priority ceiling protocol* (IPCP) [12]. When accessing a shared resource, the requesting thread temporarily raises its scheduling priority to the *ceiling priority* of the resource. The ceiling priority of each resource is defined as the maximum priority of all threads accessing the resource. This effectively excludes other threads from accessing the resource at the same time, but only as long as a thread does not block. Compared to the *original priority ceiling protocol* (OPCP) [32], IPCP is simpler to implement, and it is the implementation mandated by e.g. POSIX and found in real-world RTOS implementations. When `max_prio` is used as ceiling priority, the protocol effectively reduces to *non-preemptive critical sections* (NPCS).

We now consider resource sharing between two or more processor cores. For *short* critical sections, we combine non-preemptive critical sections with FIFO spinlocks, like in MSRP [15]. As spinlock implementation, both *ticket locks* and *MCS locks* provide fair FIFO ordering [24]. For *long* resource requests, we use mutexes as suspension-based mechanism instead of busy waiting. The mutexes use priority-ordering, following POSIX as example. This approach is also used in operating system kernels such as Linux. We further assume that synchronization mechanisms are used correctly. Preventing deadlocks should be done in user space and is outside of the scope of this paper.

Note that the presented approach is applicable to both single and multi-processor systems. A single processor system does not require the use of spinlocks. Additionally, the approach can be extended to partitioned EDF by using the *deadline floor protocol (DFP)* instead of IPCP [11, 2]. We plan to investigate this combination in future work.

To analyze the worst case, we do not perform an actual WCET analysis, as this requires detailed knowledge of the underlying processor architecture and the overall system, see e.g. [6]. Instead, we keep the worst-case considerations on an abstract level and identify the worst case for each building block in $\mathcal{O}$ notation. We think this is the right level to decide for or against a mechanism in general.

3 Previous Work

3.1 Efficient Synchronization Mechanisms and Futexes

The observation that system call overheads are expensive is well known. Several approaches were proposed to improve efficiency of synchronization mechanisms: Keedy describes atomic *test-and-increment* and *decrement-and-test* operations to implement uncontended semaphore operations and call into the operating system only to suspend or to wake up threads [18]. Birrell et al. describe an optimization for mutexes, condition variables, and semaphores in the Taos operating system to only call the operating system kernel when there is contention or a thread is waiting on a condition variable [4]. A similar approach is also described for *Benaphores* in BeOS [31]. For synchronization in a Java virtual machine, Bacon et al. proposed *Thin Locks*, based on atomic operations for uncontended cases, with a fall-back to OS provided synchronization primitives [3].

Futexes extend these prior approaches as a generic *compare-and-block* mechanism. Futexes allow a thread to *wait* on a variable in user space or to *wake up* a given number of waiting threads. The kernel dynamically creates an internal wait queue based on the given user space address and keeps the wait queue as long as threads are waiting. The third conceptual futex operation allows to *requeue* waiting threads from one wait queue to another. This helps when

signaling condition variables to prevent *thundering herd effects* [16]. The requeue operation transfers waiting threads from the condition variable's wait queue to the mutex' one instead of waking the threads up and letting them compete on the associated mutex.

Futexes were first introduced in Linux to implement POSIX thread synchronization objects in user space [14], and then later extended to support the priority inheritance protocol (PIP) [16]. Over time, scalability issues were addressed and discussed [8, 9]. Pizlo describes an approach resembling futexes using cascaded locks and hashed wait queues in user space for fine-grained locking and condition variables in the WebKit browser [28]. Spliet et al. evaluated the use of different real-time locking protocols for futexes in the context of LITMUS^{RT}, a Linux-based testbed for real-time scheduling experiments [34]. They use an index-based wait queue look-up and a bitmap of acquired locks that is shared between user space and kernel. Zuepke et al. presented approaches for deterministic futexes with FIFO ordering based on doubly-linked lists and look-up by thread ID [36], by replacing the futex hash table with binary search trees to bound interference effects to logarithmic runtime [38], and by using an index-based wait queue look-up [37].

3.2 Lock Holder Preemption

A problem with spinning synchronization in user space is that a thread can be preempted inside a critical section, as the scheduler is not aware of the critical section, and other threads continue spinning while the lock holder is preempted. In turn, multiple *preemption-safe lock mechanisms* were proposed [25], which either *prevent* preemption or try to *recover* from the fact that a lock holder is preempted, e.g. *adaptive mutexes*, by spinning only for a limited time and then falling back to blocking [26], scheduler hints [5], or *scheduler-conscious synchronization* with liveness indicators [20]. Michael and Scott provide an overview on these techniques [25]. We focus on mechanisms to prevent preemption.

Edler et al.'s *temporary non-preemption mechanism* in *Symunix II* uses two flags shared between user space and kernel [13]. Before starting spinning, a thread indicates its wish to disable preemption in the first flag. When the kernel actually wants to preempt the thread inside the critical section, the kernel sets the second flag to indicate a pending preemption request and let the thread continue. After the thread finishes the critical section, it clears the first flag and checks the second flag if it has to yield the processor. The kernel can set up a short timeout to enforce preemption of uncooperative threads not enabling preemption. An alternative approach is the *two-minute warning* mechanism proposed by Marsh et al. in *Psyche* [22]: the kernel indicates upcoming preemption (i.e. end of time slice) in a user readable flag, and a user space thread then avoids acquiring any spinlocks and rather yields. Holman and Anderson present a third approach in the context of *Pfair-scheduling* [17]: a locking attempt in the *frozen interval* at the end of a time slice implicitly blocks the thread until the next time slice. All three mechanisms were originally designed for systems with quantum scheduling where the time of preemption is known in advance.

3.3 Efficient IPCP Implementations

In the context of real-time systems with P-FP scheduling, lock holder preemption *by lower priority threads* can be also prevented by using real-time locking protocols such as IPCP. To reduce the overhead of frequent scheduling priority changes, Zuepke et al. presented two protocols to change a thread's priority lazily in user space [37]. The protocols use two variables shared between user space and kernel, similar to Edler et al.'s *temporary non-preemption mechanism* [13]. Almatary et al. presented an efficient implementation of IPCP (and DFP as well) using a different protocol with three shared variables [2].

Listing 1 User space per-thread data and UTCB

```
typedef struct {
    tid_t    tid;       // thread ID
    prio_t   uprio;     // user space priority
    prio_t   nprio;     // next thread's priority
    uint32_t ustate;    // user space state variable
    <...>
} thread_t;

#define SELF <...>      // get current thread's per-thread data
```

Listing 2 Priority change operations in user space

```
prio_t prio_raise(prio_t new_prio) {
    prio_t old_prio = SELF->uprio;
    assert(new_prio >= old_prio);
    SELF->uprio = new_prio;
    return old_prio;
}

void prio_restore(prio_t old_prio) {
    SELF->uprio = old_prio;
    if (old_prio < SELF->nprio) {
        sys_preempt();
    }
}
```

Listing 3 Priority change operations in the kernel

```
#define CURRENT <...>   // get current thread's kernel state (TCB)
#define UTCB <...>      // get current thread's user space data

void sys_preempt(void) {
    prio_t uprio = min(UTCB->uprio, CURRENT->max_prio);
    kernel_preempt(CURRENT, uprio);
}

void kernel_wake(<...>) {
    <...>
    prio_t uprio = min(UTCB->uprio, CURRENT->max_prio);
    prio_t nprio = ready_queue_next()->prio;
    UTCB->nprio = nprio;
    if (uprio < nprio) {
        kernel_preempt(CURRENT, uprio);
    }
    <...>
}

void kernel_preempt(thread_t *thr, prio_t prio) {
    // preempt the current thread
    <...>
}
```

All three protocols comprise a priority raise operation and a priority restore operation. The raise operations indicate a temporarily elevated scheduling priority in shared variables without using a system call. The restore operations revert the scheduling priority to the previous value and contain one optional system call to preempt the thread. On scheduling events, e.g. when releasing a suspended thread, the kernel obtains the elevated scheduling priority from the shared variables to consider whether to defer the preemption of the current thread.

In Zuepke et al.'s first protocol [37] and Almatary et al.'s protocol [2], the kernel also updates an in-kernel representation of the current thread's scheduling priority and then indicates that it has *observed* the priority change in the shared variables. In both protocols, the priority restore operations need a system call to update the in-kernel value again, even if the current thread will not be preempted.

Zuepke et al.'s second protocol addresses this shortcoming [37]. Here, the kernel does not update an in-kernel priority and indicates whether it has observed the priority change, but instead provides the priority of the *next* thread eligible for scheduling in a shared variable and updates its value on each scheduling event. With this, the restore operation only issues a system call when the current thread really needs to be preempted. We therefore focus only on this protocol. The two shared variables are named `uprio` (*user priority*) and `nprio` (*next thread's priority*).

Listing 1 shows the shared protocol variables `uprio` and `nprio` among other variables. We keep these variables in the per-thread TLS. Listing 2 shows the priority change operations in user space. The priority raise operation returns the previous scheduling priority for the later restore operation. Listing 3 shows the kernel parts of the implementation for the preemption system call and when waking up a suspended thread. In both cases, the user space priority is bounded to `max_prio` before further use.

Note that the kernel is optimized for frequent priority changes and does not keep the current thread on ready queues, so `nprio` is naturally available from the highest priority thread on the ready queue and also used internally by the kernel to decide whether to preempt the current thread or not. Blackham et al. describe a similar technique for seL4 [6].

From the point of view of their worse-case timing, all three protocols show equal behavior: no system call is needed to raise the priority, but a restore operation might require a system call[1], so we assume the system call is always called as the worst case. Also, all protocols show equal overhead to synchronize and validate user priorities when testing to preempt the currently running thread.

4 Light-Weight Blocking for IPCP

With the IPCP implementation presented in Section 3.3, we can now realize non-preemptive critical sections in user space. Listing 4 shows a simple example. A thread first raises its user space scheduling priority to `max_prio` to become non-preemptive, then acquires a spinlock. The threads previous priority is kept in `old_prio`. At the end of the critical section, the thread unlocks the spinlock and restores its previous scheduling priority. Note that this sequence does not need any system calls in the fast path. The system call to preempt the thread in `prio_restore` is only needed when in the meantime another thread with a medium priority higher than `old_prio` became ready.

[1] In Zuepke et al.'s protocols [37], the preemption system call would be superfluous if the thread is preempted after updating `uprio` but before calling `sys_preempt`. Almatary et al. solve this corner case at the expense of additional protocol variables [2]. In the worst case, one syscall is always needed.

Listing 4 Example non-preemptive critical section in user space

```
spin_t example_lock;

void example_cs(<...>) {
    prio_t old_prio = prio_raise(max_prio);
    spin_lock(&example_lock);
    <...>
    spin_unlock(&example_lock);
    prio_restore(old_prio);
}
```

Listing 5 System call implementation for light-weight waiting in IPCP

```
err_t sys_wait_at_prio(uint32_t *ustate, uint32_t cmp,
                       timeout_t timeout, prio_t wait_prio) {
    <...>
    spin_lock(&ready_queue_lock);
    uint32_t val = safe_user_space_read_access(ustate);
    if (val == cmp) {
        CURRENT->prio = min(wait_prio, CURRENT->max_prio);
        CURRENT->ustate = ustate;
        CURRENT->state = WAIT_USER;
        err = kernel_wait(CURRENT, timeout);
    } else {
        err = EAGAIN;
    }
    spin_unlock(&ready_queue_lock);
    <...>
}
```

We now extend the critical sections with a blocking mechanism that interacts properly with the IPCP implementation and the spinlock-protected critical sections. We opt to manage the wait queue of blocked threads in user space.

Waiting: We can make the following considerations for a waiting operation:

- Suspending the current thread needs help by the kernel. This requires a system call.
- Waiting *after* calling `prio_restore` could trigger unnecessary preemption, as the calling thread is going to suspend itself anyway. Waiting at `max_prio` is advisable.
- Waiting *inside* the spinlock-protected critical section causes problems, as other threads would be unable to acquire the spinlock. A system call to suspend a thread must happen *after* unlocking the spinlock in user space.
- As the spinlock-protected critical section protects any internal state w.r.t. blocking, a system call to suspend a thread *outside* the critical section must prevent *missed wake-ups*.
- A thread's scheduling priority at wakeup time should reflect its original priority. When a thread is woken up at `max_prio`, it would execute only to the point where it restores its original priority and then causes unnecessary context switches if other medium priority threads are ready.
- Spurious wake-ups, e.g. timeouts, require a second critical section *after* waiting to remove the current thread from the wait queue again.

Listing 6 System call implementation for light-weight wake-up of a thread in IPCP

```
err_t sys_wake_set_prio(tid_t tid, uint32_t *ustate, uint32_t cmp,
                        prio_t new_prio) {
    <...>
    thread_t *thr = kernel_lookup_TCB_by_tid(tid);
    spin_lock(&ready_queue_lock);
    new_prio = min(new_prio, CURRENT->max_prio);
    UTCB->uprio = new_prio;
    uint32_t val = safe_user_space_read_access(ustate);
    if ((thr != NULL) && (thr->state == WAIT_USER)
                      && (thr->ustate == ustate) && (val == cmp)) {
        kernel_wake(thr);
    } else if (new_prio < UTCB->nprio) {
        kernel_preempt(CURRENT, new_prio);
    }
    spin_unlock(&ready_queue_lock);
    <...>
}
```

To prevent missed wake-ups, we use a *compare-and-block* mechanism similar to futexes. Inside the spinlock-protected critical section, a thread decides to block and reads a state variable. The state variable encodes a waiting condition and is changed by a wakeup operation. Then the thread unlocks the critical section in user space and calls the kernel to suspend. The kernel reads the state variable again and, if the current value matches the previous value, suspends the thread. An alternative would be to call the kernel from inside the critical section and let the kernel unlock the critical section before suspending the calling thread. This would prevent any ambiguity with parallel wakeup operations. However, the kernel would then need to know the exact semantics of the spinlocks to unlock the spinlock for the caller. We opt to unlock the spinlock in user space instead. This keeps the kernel simple.

To address the problems of the scheduling priority at wakeup time, we temporarily *drop* the priority while waiting. While still executing at `max_prio` in user space, a thread calls a system call to wait at a lower priority `wait_prio`. The kernel then temporarily sets the thread's priority to `wait_prio` while waiting. When the thread is woken up again, it will be enqueued at `wait_prio` on the ready queue. And when the thread is eventually scheduled, the kernel increases the scheduling priority back to `max_prio`, and then returns from the waiting system call. We can easily achieve this by using the following trick in the IPCP implementation of Section 3.3: the kernel lets the thread wait at `wait_prio`, but leaves `uprio` unmodified while waiting. Note that `uprio` was set to `max_prio` before waiting, so the thread is effectively running at `max_prio` again after waiting as well. The thread in user space can then either lock the spinlock again, or leave the IPCP-protected critical section and restore its previous scheduling priority `old_prio`. As no other threads with a higher priority will be ready at that moment, no system call will be needed.

Listing 5 shows the implementation of the `sys_wait_at_prio` system call. With the ready queue locked, the kernel evaluates if the content of a given state variable in user space (`ustate`) matches a compare value (`cmp`). If this is the case (no missed wakeup), the kernel keeps the address of `ustate` for later, and suspends the current thread with the given timeout (`timeout`) on the given waiting priority (`wait_prio`) in an internal waiting state `WAIT_USER`. In case of a missed wakeup, the kernel returns an error condition. The system call does not change `uprio`, so the thread will be immediately boosted to its previous `uprio` after wakeup.

Wakeup: For a wake-up operation, we can discuss similar considerations as for waiting:

- Waking up a waiting thread needs help by the kernel. This requires a system call as well.
- The wakeup system call addresses a blocked thread directly by its thread ID (`tid`).
- Waking a thread up *after* calling `prio_restore` could cause unnecessary delays due to preemption. Again, doing the wake-op operation at `max_prio` is advisable.
- The wakeup system call could happen *inside* the spinlock-protected critical section or be deferred after unlocking the spinlock. In the latter case, a system call to wake up a thread *outside* the critical section must prevent *spurious wake-ups*.
- Waking up a thread with a priority higher than oneself causes preemption when restoring the previous priority.

We opt to wake up a thread outside the critical section. To prevent *spurious wake-ups*, we use same technique as in the waiting operation. User space code passes the address and expected value of a state variable in user space to the system call, and the kernel compares the state variable to the expected value and only unblocks the given thread on a match. This constitutes a *compare-and-unblock* mechanism.

To prevent unnecessary system calls for preemption in `prio_restore`, we defer the wake-up to the latest possible point and *fuse* the system call for wake-up with the system call for preemption. The resulting system call combines wake-up, priority change, and preemption.

Listing 6 shows the implementation of the `sys_wake_set_prio` system call. The wake-up operation directly references a waiting thread by its thread ID (`tid`), so the system call first looks up the TCB in the kernel. With the ready queue locked and executing non-preemptively, the kernel first bounds the given priority to restore and updates `uprio` in user space. Then the kernel evaluates the wake-up condition: the thread must exist, it must be waiting in a call to `sys_wake_set_prio`, it must wait on the same `ustate` variable, and the current value of `ustate` must match a compare value (`cmp`). If the condition is met, it wakes up the thread. Otherwise, it just preempts the current thread, if necessary. Recall that `kernel_wake` in Listing 3 also preempts the current thread when a higher priority thread is woken up.

5 Non-Preemptive Busy-Waiting Monitors in User Space

We now discuss how to construct higher-level synchronization mechanisms based on the IPCP implementation of Section 3.3 to temporarily disable preemption, fair spinlocks, and the light-weight waiting and wake-up primitives of Section 4. We use the term *monitor* for the resulting design because the operations resemble the ones found in monitor implementations.

A monitor protects the *state* of a specific synchronization object of a higher-level synchronization mechanism, e.g. the current lock owner of a mutex, and one or more wait queue heads. We place the according wait queue nodes in each thread's TLS segment, or on the stack in the waiting functions. As a thread can only wait on one wait queue at a time, the space for wait queue nodes is bounded.

A thread *enters* the monitor to gain *exclusive access* to the internal state, and *leaves* the monitor afterwards. When multiple threads try to enter the monitor, they are serialized by the spinlock. This relates to a FIFO-ordered *enter queue*. Within the monitor, a thread can decide to *wait* or to *notify* waiting threads. Waiting effectively comprises enqueuing the thread in a wait queue, leaving the monitor, blocking in the kernel, and entering the monitor again after waiting. To handle spurious wake-ups, a thread must eventually remove itself from the wait queue. For notification, a thread removes a blocked thread from the wait queue and wakes up up the blocked thread when leaving the monitor. The design is

optimized to perform an uncontended *enter* → *leave* sequence without system calls, and both *wait* and *notify* with one system call in the best case. Note that more than one thread can be woken up, but this requires one additional system call for each thread.

Another important aspect is to handle the `ustate` variables correctly. For this, we draw from *eventcounts* and *sequencers* [30]. Firstly, we will use `ustate` as a counter that is incremented *before* a thread suspends or is woken up. The motivation to use a counter instead of a binary state like `WAITING` and `READY` is to prevent spurious wake-ups due to *ABA-problems* when two waiting operations follow each other back to back. Secondly, we will use a dedicated `ustate` variable for each thread. Like a *sequencer*, the increments of a thread's `ustate` variable in user space order the particular wait and wakeup operations of the related thread. The waiting operation in the kernel follows *eventcounts*. As we use a dedicated per-thread counter, a wait operations will observe at most one additional increment from the corresponding wakeup operation. With this, the *compare-equal* condition for blocking in the kernel is sufficient to detect missed wake-ups. The increment before waking up a thread also follows *eventcounts*. As any further waiting attempt would increment `ustate` again, the *compare-equal* condition for unblocking in the kernel is sufficient to prevent spurious wake-ups. Note that we keep `ustate` in the TLS segment of each thread, see Listing 1.

The `ustate` variables are only modified in the critical sections of the related synchronization objects in user space. Nevertheless, an implementation should change `ustate` by using atomic read and write operations to prevent undefined behavior by the compiler, as the kernel reads the current value in parallel.

6 Analysis of Designs for Blocking Synchronization Mechanisms

6.1 Building Blocks

To compare the presented monitor approach to futexes and a traditional implementation using dedicated system calls, we first analyze how blocking synchronization mechanisms are typically implemented in operating systems using fine-grained locking like Linux. For this, we define a *generic blocking mechanism* and decompose it into its internal building blocks. The generic blocking mechanism provides two operations. The *waiting* operation either lets the calling thread continue or suspends it, and the *wake-up* operation can optionally wake up a previously suspended thread. This resembles common operations on mutexes, semaphores, or condition variables, but without specifying the *exact* semantic of the synchronization mechanism. We denote a step where the exact semantics of an actual synchronization mechanism would be required as *semantic operation*. Note that futexes and the monitor approach require two semantic operations, a 1st semantic operation in user space and a 2nd semantic operation in the kernel. The system call approach needs one only in the kernel.

Table 1 shows a comparison of the generic blocking synchronization mechanism implemented as (i) a system call based approach as baseline, (ii) futexes like in Linux, and (iii) the proposed monitors. The upper part of the table shows the layered individual steps to suspend or to wake up a thread from top to bottom. A "•" marks the operations when no blocking is needed, i.e. the fast path. The lower part of the table shows associated data in user space and the kernel. Here, a "◇" indicates global data. Comparable operations and data objects are placed in the same rows.

System call: In the baseline implementation using system calls, user space code calls the kernel with an identifier to a kernel object. In turn, the kernel validates the identifier and retrieves the kernel object comprising all necessary data in the look-up step. Then the kernel

Table 1 Comparison of three implementations of a generic blocking synchronization mechanism. The upper part shows the layered operations from user space down to the kernel. A " • " marks the operations in the fast path when no blocking is needed. The lower part shows the associated data in both user space and the kernel. A "◇" denotes global data.

operations	**system call (baseline)**	**futex**	**monitor (this paper)**
user space			• disable preemption
			• lock user space object
		• 1st semantic op. (atomic)	• 1st semantic operation
			wait queue operation
	• system call	system call	system call
kernel	• look-up kernel object	look-up futex wait queue	look-up thread
	• disable preemption	disable preemption	disable preemption
	• lock kernel object	lock wait queue	
	• semantic operation	2nd semantic operation	
	wait queue operation	wait queue operation	
	lock ready queue	lock ready queue	lock ready queue
			2nd semantic operation
	suspend / wake-up	suspend / wake-up	suspend / wake-up

data model	**system call (baseline)**	**futex**	**monitor (this paper)**
user space	ID of kernel object	futex value (atomic)	user space object lock
		additional semantic state	semantic state
			wait queue
			◇ thread states
kernel	kernel object lock	wait queue lock	
	semantic state	address	
	wait queue	wait queue	
	◇ ready queue lock	◇ ready queue lock	◇ ready queue lock
	◇ thread states	◇ thread states	◇ thread states

disables preemption, locks the data in the kernel object, and performs a semantic operation, such as checking and modifying the internal state. At this point, the semantic operation decides whether the operation is completed or if a wait queue operation is needed. In the latter case, the kernel either enqueues the calling thread on the wait queue, or removes a waiting thread from the wait queue, depending on the desired operation. Then the kernel must lock internal scheduling data (ready queue lock) before it can finally suspend the calling thread or wake up a waiting thread.

Futex: Compared to the system call approach, the main difference of the futex-based implementation is the 32-bit variable in user space expressing the semantic state. Depending on the atomic operation on the variable, the 1st semantic operation either succeeds immediately or requires a system call. In the kernel, the look-up of an associated in-kernel wait queue is based on the user space address of the atomic variable, but the following steps are similar to the baseline approach. That is because futexes *are* a generic compare-and-block mechanism. The 2nd semantic operation in the kernel checks if the futex value has changed in the meantime. This indicates a parallel wake-up operation, and the system call returns in this case, similarly to the baseline version.

Monitor: The monitor-based implementation differs from both approaches. The user space object already comprises a lock, semantic state, and a wait queue. User space code disables preemption and locks the object before it evaluates the internal semantic state in the 1st semantic operation. In the fast path, the 1st semantic operation succeeds and the operation completes. In the slow path, a system call is required to block or to wake up. For blocking or wake-up, the wait queue operation either adds the current thread to the wait queue or removes a thread from the wait queue and then in turn calls into the kernel. The kernel first validates the given thread ID and locates the target thread. Then the kernel locks the necessary scheduling data and performs a 2nd semantic operation. The 2nd semantic operation is serialized by this last lock and detects parallel wake-up or suspend operations. If its semantic check succeeds, a suspend or wake-up operation takes place.

6.2 Analysis of the Fast Paths

Note that the overhead in the fast path in user space for both the futex and the monitor variants is less than a system call, but the monitor shows more overhead than a futex. A futex fast path typically comprises one atomic operation with either acquire or release semantics or equivalent memory barriers. The monitor fast path requires to disable preemption (load and store instructions on the local processor), and one or two atomic operations with both acquire and release semantics or equivalent memory barriers in the spinlock operations.

When comparing the three implementations shown in Table 1, we can see that both the futex and the monitor require additional operations compared to the baseline approach. If we consider the atomic operation in the futex case as a (somewhat "compressed") critical section, then both futex and monitor use a critical section in user space to guard the fast path with the 1st semantic operation. However, there is a second critical section in the kernel as well to guard the 2nd semantic operation that decides whether to block or unblock.

The key technique for the separation into a fast-path in user space and the actual blocking/unblocking in the kernel is to use two serialized critical sections, one in user space and one in the kernel. These critical sections are *loosely coupled* by semantic state data that is *set* in the 1st semantic operation and *checked* in the 2nd one. The benefit of this pattern is that one can determine the WCET of each critical section in isolation.

The 2nd semantic operation in the kernel prevents race conditions between suspend and wake-up operations ongoing in parallel. In the common case, the 2nd semantic operation simply succeeds, but how do the implementations behave if the 2nd semantic operation fails? Non-real-time futexes in Linux use *compare-equal* semantics. If the futex value in user space no longer matches a given previous value, the kernel does not suspend the calling thread, but returns to user space. The user space part then *retries* the whole operation from the beginning. For priority inheritance (PI) mutexes, the Linux kernel solves this situation internally and tries to lock a mutex for the calling thread *again.* In both cases, the loops for this are potentially unbounded. The monitor approach naturally works without any retrying.

6.3 Analysis of the Worst Case

From a worst-case point of view, where we assume that the fast paths are not taken, we see mostly similar operations for all three variants in Table 1. For a typical implementation, we can further assume that a system call, a look-up of a pre-registered kernel object, suspending the current thread, or a wake-up of a blocked thread need $\mathcal{O}(1)$ time. Locking operations usually have $\mathcal{O}(m)$ worst-case behavior for m processor cores. Wait queue operations take at most $\mathcal{O}(\log n)$ time when using balanced binary search trees or $\mathcal{O}(n)$ when using sorted

linked lists for n blocked threads. For the semantic operations, we can also assume $\mathcal{O}(1)$ timing behaviour, e.g. checking the state of a mutex and either making the thread the new mutex owner or suspending the thread.

When we compare the futex approach to the baseline, we see that using futexes adds an atomic operation in user space and requires a more complex look-up operation in the kernel. Hashed wait queues are fast, but have determinism issues and degrade to $\mathcal{O}(n)$ in the worst case. In [38], the author describes a deterministic approach by using a balanced binary search trees with $\mathcal{O}(\log n)$ time for look-up of wait queues. A critical point is to prevent loops if the second semantic operation fails, as this is the case for futex operations on real-time-mutexes using the priority inheritance protocol in Linux. These potentially unbounded loops make futexes hard to assess in a WCET analysis.

The monitor approach moves some steps of the baseline version from the kernel to user space and adds just one additional semantic check before suspending or waking up a thread. The critical point is the non-preemptive critical section in user space. An in-kernel implementation can simply disable interrupts to achieve non-preemptiveness, however, this is not possible in user space. Therefore, in a WCET analysis, extra delays due to interrupt handling have to be accounted for, e.g. see [7] for applicable techniques. Additional overhead is caused by the priority restore operation, where we must account an additional system call to preempt the thread after the critical section in user space. We can model this overhead as a constant. All in all, a WCET analysis of the kernel parts of the monitor approach should be simpler than for the baseline approach, but the user space parts require to account for additional overheads of interrupt handling and preemption.

7 Implementation of Blocking Mutexes and Condition Variables

7.1 Blocking Mutexes

Based on the monitor building blocks of Section 5, we now present a blocking mutex. The data structure representing a mutex comprises a spinlock, an owner field, and a wait queue.

Listing 7 shows a mutex lock operation with timeout in `mutex_lock`. The function first raises the scheduling priority to `max_prio` and locks the internal spinlock. If the mutex is currently unlocked, the function registers the calling thread as mutex owner and returns successfully after unlocking the spinlock and restoring the previous priority. Otherwise, the function adds the current thread to the priority-ordered wait queue using its original priority. Then the mutex function retrieves and increments the current value of its user state variable `ustate`, unlocks the spinlock, and suspends itself in the kernel with the timeout and its original priority. After wake-up, the function locks the spinlock again and tests if `ustate` was incremented. If true, the current thread is now the lock owner. If not, the timeout has expired instead, and the function removes the thread from the wait queue. The function unlocks the spinlock, restores the previous priority and returns the status of lock ownership.

The unblock operation is shown in `mutex_unlock` in Listing 7. Again, the function increases the scheduling priority and locks the spinlock. Then it tries to retrieve the highest priority waiting thread from the wait queue. If no thread is found, the function sets the mutex to unlocked state, unlocks the spinlock, restores the previous priority, and returns. Otherwise, the operation makes the waiting thread the new lock owner, increments its user mode state variable, unlocks the spinlock, and performs a fused wake-up and priority restore operation.

In the best case, when we assume that the priority restore operation does not preempt the thread, then both mutex lock and unlock operations do not need any system calls in the fast path, and only one system call for blocking and wake-up on contention. This is similar

Listing 7 Implementation of a blocking mutex with timeout and a priority-ordered wait queue

```
typedef struct {
    spin_t lock;        // internal spinlock
    tid_t owner;        // current mutex owner or UNLOCKED
    waitq_t waitq;      // priority-ordered mutex wait queue
} mutex_t;

bool_t mutex_lock(mutex_t *m, timeout_t timeout) {
    prio_t old_prio = prio_raise(max_prio);
    spin_lock(&m->lock);

    bool_t success = (m->owner == UNLOCKED);
    if (success == TRUE) {
        m->owner = SELF->tid;
        goto out;
    }

    waitq_add_ordered(&m->waitq, SELF, old_prio);
    uint32_t seq = ++SELF->ustate;
    spin_unlock(&m->lock);
    sys_wait_at_prio(&SELF->ustate, seq, timeout, old_prio);
    spin_lock(&m->lock);

    success = (SELF->ustate != seq);
    if (success == FALSE) {
        waitq_remove(&m->waitq, SELF);
    }

out:
    spin_unlock(&m->lock);
    prio_restore(old_prio);
    return success;
}

void mutex_unlock(mutex_t *m) {
    assert(m->owner == SELF->tid);
    prio_t old_prio = prio_raise(max_prio);
    spin_lock(&m->lock);

    thread_t *next = waitq_remove_highest(&m->waitq);
    if (next == NULL) {
        m->owner = UNLOCKED;
        spin_unlock(&m->lock);
        prio_restore(old_prio);
        return;
    }

    m->owner = next->tid;
    uint32_t seq = ++next->ustate;
    spin_unlock(&m->lock);
    sys_wake_set_prio(next->tid, &next->ustate, seq, old_prio);
}
```

to futexes. However, in the worst case, we must assume that the mutex is contended and the thread is interrupted, and then we need to account two system calls for `mutex_lock` (one for waiting, one for preemption) and one system call for `mutex_unlock` (combined wake-up and preemption). Compared to futexes and the baseline, we have to account one additional system call for preemption in `mutex_lock`.

7.2 Condition Variables for Blocking Mutexes

We present condition variables with POSIX semantics for the mutexes described in Section 7.1. Both `cond_wait` and `cond_notify` operations expect a locked support mutex. The data type of the condition variable comprises just a wait queue. The internal spinlock of the support mutex also protects the wait queues of condition variables. Discussing the implementation in detail exceeds the page limitation of this paper, so we provide only a brief overview.

`cond_wait` enqueues the calling thread on the wait queue of the condition variable, unlocks the mutex, then waits. `cond_notify` simply requeues the given number of threads (one or all) from the wait queue of the condition variable to the wait queue of the mutex. After wake-up, a thread is either the mutex owner (successfully notified and requeued to the mutex), or not (spurious wake-up, e.g. timeout). In the latter case, the thread blocks again on the mutex.

In the best case (no preemption when restoring the previous priority), waiting needs one system call and notifying needs no system call at all, as threads just get requeued to the mutex wait queue and the actual wake-up is done when the notifying thread unlocks the mutex. Therefore, we must include the mutex operations in the discussion as well. Now a *lock* → *wait* → *unlock* sequence takes at most one system call to wait for the condition, and a *lock* → *notify* → *unlock* sequence also needs only one when unlocking, regardless of the number of notified threads. Again, this is similar to futexes. In the worst case (with preemption), we must account *five* system calls for waiting: one to lock the mutex, one when unlocking the mutex and waking up the next mutex owner, one for waiting on the condition variable, another one to block on the mutex again on spurious wake-ups, and the last one for preemption when unlocking the mutex. Futexes require four system calls (the final one for the preemption is not needed). For notification, the number of required system calls is two, one to lock the mutex and one to notify. This is the same for futexes. In both best and worst case, the baseline version always needs three system calls for these sequences. Also, moving threads between queues has the same overhead in all three versions.

8 Experimental Evaluation

For comparison, we have implemented all three approaches (baseline, futex, and monitor) in a small real-time operating system (RTOS) named *Marron*. Marron provides static partitioning of OS resources with fixed-priority scheduling on each processor core. The kernel implements fine-grained locking with a similar implementation complexity as described in Table 1.

Marron currently supports only 32-bit ARM platforms, so we evaluated the approaches on three system-on-a-chip platforms. A *BeagleBone Black* provides a single Cortex A8 core running at 550 MHz, a *Freescale i.MX6Q* has four Cortex A9 cores at 792 MHz each, and the *BeagleBoard-X15* has two Cortex A15 cores at 1 GHz. Note that the Cortex A8, A9, and A15 cores have different microarchitectures. The A8 is an in-order design with a 13-stage pipeline, while the A9 and A15 are out-of-order designs, with a short 8-stage pipeline on the A9 and a longer 15-stage pipeline on the A15. As the working set is small, we expect our experiments to fit into both instruction and data caches and not access any external DRAM. Also, we run our experiments without any interference from other applications. For better

Table 2 Overhead measurements on 32-bit ARM platforms in CPU cycles. The first set of measurements determines the overhead of the building blocks, the last four sets compare the different approaches for synchronization mechanisms in uncontended and contended scenarios.

Test		Cortex A8	Cortex A9	Cortex A15
acquire and release barriers		33	6	20
CAS without barriers		8	22	30
uncontended spin_lock/unlock pair		74	38	108
prio_raise/restore pair		17	12	6
null system call		177	107	205
uncontended mutex lock/unlock pair	system call	711	465	742
	futex	110	87	175
	monitor	218	147	260
contended mutex lock/unlock pair	system call	1557	1137	1503
(same core)	futex	2051	1838	1995
	monitor	1792	1668	1689
contended mutex trylock/unlock pair	system call	—	1172	1696
(2 cores)	futex	—	302	495
	monitor	—	712	1087
contended mutex trylock/unlock pair	system call	—	3182	—
(4 cores)	futex	—	779	—
	monitor	—	2445	—

comparison, we present the results of each CPU core in clock cycles. Measurements were taken with the internal cycle counter of the CPU cores. We ran each measurement in a loop 1024 times and divided the result, therefore all measurements include the loop overhead.

We performed an experiment to evaluate the fast-path performance of the three discussed approaches and their building blocks in different contention scenarios Table 2 shows the results for the three different processor cores. The overhead of the building blocks in isolation and the uncontended mutex runs show stable results, as these tests run in a single-threaded context. We determine the results for the contended case on the same processor core indirectly with the help of a second thread. The presented results show the remaining overhead of the contended mutex operations and two additional context switches. The contended case on different processor cores determines the overhead of the operations in user space, e.g. the internal critical sections of the monitor, on two or four processor cores in parallel. The test uses `mutex_trylock` instead of a blocking system call and effectively spins until it successfully acquires the mutex. The test releases two resp. four cores from a barrier and measures the time until all cores acquire and release the mutex once, and reach the barrier again. The baseline variant implements `mutex_trylock` as system call, futexes use atomic operations in user space, and the monitor approach uses an internal critical section. These results show a great variation between runs and should be treated as a rough indicator of what to expect.

Observation 1: The futex approach is faster than the monitor approach, and both are faster than the baseline approach using system calls. The system call overhead dominates everything else. We expected to see a similar ratio of the performance of atomic operations to system calls on ARM as reported by [1] and [33] for x86 processors, and our experiments show this. Note that the measurement of the a system call includes the overhead of the operating system to save and restore registers on kernel entry and exit. Therefore, our approach to avoid system calls is reasonable.

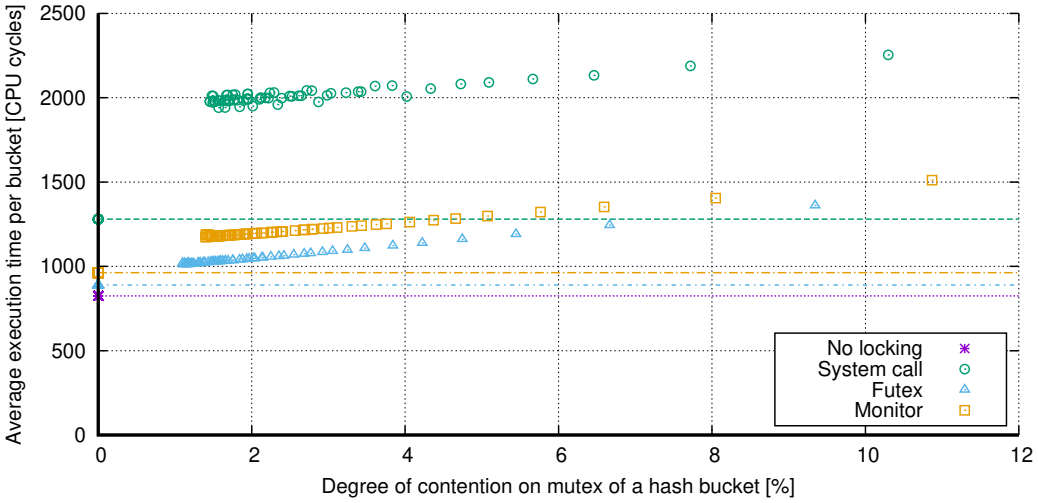


Figure 1 Operations of varying degrees of contention on a shared hash table with 64 hash buckets, each protected by a mutex. Each thread keeps a hash bucket locked for a constant time of 1 µs.

Observation 2: Efficient implementations of IPCP do not contribute much overhead to the fast path. On all three platforms, changing priorities in user space does not cause much overhead. The measurement of the priority raise/restore pair shows this.

Observation 3: On ARM processors, atomic operations cause significant overhead. The measurement of a pair of lock and unlock operations on a ticket spinlock shows significant overhead on the Cortex A15. We did not expect this. As ARM processors use a weakly ordered memory model and require explicit memory barriers to order memory accesses, we investigated this further and measured memory barriers and CAS operations in isolation (first three rows). When using futexes, a mutex lock / unlock pair comprises a sequence of CAS + acquire barrier + release barrier + CAS, and the parts add up correspondingly. Similarly, the monitor requires two pairs of spinlock lock / unlock operations of equal complexity than a futex pair, explaining the more than twice as high overhead of the monitor in the fast path. Note that the baseline version shows a similar locking overhead inside the kernel.

Observation 4: On contention on the internal critical section of the monitor, the monitor shows worse results than futexes. Here, threads in `mutex_trylock` repeatedly spin to lock the internal critical section of the monitor to detect that the mutex is already taken. Futexes fare well here, as they can directly probe the mutex due to the atomic operations. And implementing `mutex_trylock` by repeated system calls is not a reasonable design choice.

Observation 5: In the contended case with blocking, the system call approach shows the least overhead. Futexes and monitors show more overhead, as they first detect contention in user space before calling into the kernel. Monitors are slightly faster than futexes due to the simpler kernel implementation. This stresses the point that the fast-paths in the uncontended case come with extra costs in the contended case.

We conduct another experiment to compare the three discussed approaches in a scenario of varying degrees of contention. For this, we distribute 2^{24} random values following a square distribution to a shared hash table comprising 64 hash buckets. We run this experiment using four parallel threads (one for each core) on the Cortex A9. Each thread atomically draws a

unique value from the random pool and locks the resulting hash bucket for a constant time of 1 µs. Statistics counters in the mutex implementations account contended and uncontended cases. Figure 1 shows the average execution time per bucket operation in CPU cycles (incl. locking overhead) for the varying degrees of contention observed in the hash buckets. We include the results of a run without any locking and without contention as reference shown as dotted horizontal lines. Note that 1 µs relates to 792 cycles on the Freescale i.MX6Q platform. The results show that both futexes and monitors result in less overhead in low contention scenarios compared to the system call approach. Also, futexes show less overhead compared to the monitor. A second effect is that both futexes and monitors show *less* contention than the system call approach. Recall that both approaches comprise *two* semantic checks whether to suspend the current thread. The second semantic check in the kernel provides a *second chance* to acquire the mutex after a brief delay (the system call overhead). Again, this effect is stronger in futexes.

9 Discussion

In general, the evaluation shows that the monitor approach works and saves CPU cycles by avoiding system calls in the uncontended case. The monitor has similar properties as futexes. Synchronization mechanisms built on top do not need initial registration in the kernel, and therefore also no resources or memory allocations in the kernel. The analysis in Section 6.3 shows that the monitors are better than futexes w.r.t. determinism, but they also come with more overhead due to the IPCP and spinlocks to protect internal critical sections, as the evaluation in Section 8 shows.

Note that we only did microbenchmarks to compare specific details in a hot-cache scenario, so the question is how big will the performance win be for a real application. This generally depends on the specific type of application. Both real-time and non-real-time applications originally designed for single-processor systems will probably not benefit much, as they are usually carefully tuned to avoid synchronization overhead in the first place. However, the situation is different if we consider multi-threaded applications with lots of fine-grained locking and low contention, e.g. language environments for Java [3] or JavaScript [28]. While these are *not* typical real-time applications, we can expect that such applications will be deployed in mixed criticality environments, so real-time operating systems should prepare to handle best-effort workloads efficiently as well.

From a WCET point of view, the monitor approach reduces determinism issues compared to futexes, as the analysis in Section 6.3 shows. Mutexes based on monitors do not require loops in the kernel or in user space, and the look-up of a particular thread is simpler than the look-up of a wait queue in futexes. As the monitor building blocks are similar to the baseline version but just shifted in place, the worst-case considerations are similar for both. The monitor adds additional constant overheads for the extra system calls for preemption. However, only the wake-up of one thread is optimized and interacts nicely with IPCP. Waking up an additional thread needs one additional system call each. But this only affects operations waking up multiple threads, e.g. when using a `barrier_wait` operation where the last thread arriving at the barrier wakes up all waiting threads. The system calls to wake-up the additional threads happen inside the critical section of the monitor and must be accounted to the WCET of the monitor as well.

Due to direct addressing of threads, the monitor approach is limited to synchronization of threads in the same process, if we assume that thread ID are local to a single process and not globally accessible. But this is the typical use case for thread synchronization in most applications anyway and therefore not a problem. However, if a system allows access to

threads in other processes, then the monitor approach can also be used for shared memory communication, like futexes. In this case, a thread's waiting state variable (`ustate`) must be placed in the shared memory as well. Note that the robustness considerations here are the same as when using futexes or other synchronization mechanisms, as synchronization over shared memory requires that applications must trust each other. But typically, access to threads in other processes is a source of unwanted interference and therefore not possible. We also expect that threads behave correctly and use the protocols appropriately, but the impact on other processes is bounded by `max_prio`. Still, further mechanisms to detect runaway threads are possible. For example, the kernel could set up a timer when it has to defer a preemption request, and the system call for preemption clears the timer again.

From a security point of view, the efficient implementation of IPCP can leak scheduling information of unrelated processes if processes are not temporally isolated, e.g. by a TDMA scheme like in ARINC 653 for avionics. In the used protocol, `nprio` exposes the priority of the next eligible thread for scheduling on the ready queue to other processes.

Lastly, the presented mutexes do not address priority inversion. The monitor approach can only implement synchronization mechanisms using *anticipatory* locking protocols [34], such as IPCP, where the locking protocol prevents problematic situations a priori. As the monitor approach already uses IPCP internally, a `mutex_lock` operation can easily implement IPCP mutexes by not restoring a thread's previous scheduling priority, but by setting the priority to the ceiling priority of the mutex after successfully locking the mutex. But also other priority-based mechanisms to address priority inversion should work, for example MPCP [29]. For MPCP, the priority space in user space needs to be partitioned into a range of normal priorities and a boosted priority range, with the non-preemptive priority on top. Similar to the IPCP example, a lock holder then uses the boosted ceiling priority of the mutex until release. Note that these approaches also work for the futexes in our setting, as they only depend on the efficient IPCP implementation. *Reactive* locking protocols [34], such as PIP, are not suitable for the monitor approach, as the kernel lacks the necessary information to build a resource allocation graph, and handling the priority changes for PIP in user space would require additional system calls. When using PIP, futexes are the perfect choice, as the protocol activates on resource contention, and this is the case where futexes need system calls anyway.

10 Conclusion

In this paper, we explored the design space of blocking synchronization mechanisms optimized to avoid costly system calls in the fast, uncontended path. We analyzed different design approaches using futexes and monitors and compared them to a system call based approach. For each mechanism, we discussed the impacts on determinism and related WCET concerns. We presented a light-weight monitor implementation in user space comprising efficient implementations of IPCP, spinlocks, and interwoven blocking mechanisms. The proposed monitor approach avoids the determinism issues of the futexes by avoiding unbounded loops and a complex look-up mechanism for wait queues in the operating system kernel. Furthermore, compared to the simple atomic variable state used by futexes, the monitor approach enables a richer semantics in critical sections. The experimental evaluation shows that both monitors and futexes considerably reduce the overhead compared to the baseline system call approach. The extra complexity of the monitor is reflected in the additional overhead of this approach compared to the futex one. The presented approach is suitable as building block to construct other blocking synchronization mechanisms in RTOS implementations for multi-processor real-time systems with P-FP scheduling.

References

1 Samy Al-Bahra. Nonblocking algorithms and scalable multicore programming. *Commun. ACM*, 56(7):50–61, 2013. doi:10.1145/2483852.2483866.

2 Hesham Almatary, Neil C. Audsley, and Alan Burns. Reducing the implementation overheads of IPCP and DFP. In *2015 IEEE Real-Time Systems Symposium, RTSS 2015, San Antonio, Texas, USA, December 1-4, 2015*, pages 295–304. IEEE Computer Society, 2015. doi:10.1109/RTSS.2015.35.

3 David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: Featherweight synchronization for java. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 258–268, New York, NY, USA, 1998. ACM. doi:10.1145/277650.277734.

4 Andrew Birrell, John V. Guttag, James J. Horning, and Roy Levin. Synchronization primitives for a multiprocessor: A formal specification. In Les Belady, editor, *Proceedings of the Eleventh ACM Symposium on Operating System Principles, SOSP 1987, Stouffer Austin Hotel, Austin, Texas, USA, November 8-11, 1987*, pages 94–102. ACM, 1987. doi:10.1145/41457.37509.

5 David L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer*, 23(5):35–43, 1990. doi:10.1109/2.53353.

6 Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. Timing analysis of a protected operating system kernel. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium, RTSS 2011, Vienna, Austria, November 29 - December 2, 2011*, pages 339–348. IEEE Computer Society, 2011. doi:10.1109/RTSS.2011.38.

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

8 Neil Brown. In pursuit of faster futexes. *LWN*, May 2016. URL: https://lwn.net/Articles/685769/.

9 Davidlohr Bueso and Scott Norton. An Overview of Kernel Lock Improvements. *LinuxCon North America, Chicago, IL*, August 2014. URL: http://events17.linuxfoundation.org/sites/events/files/slides/linuxcon-2014-locking-final.pdf.

10 Peter A. Buhr, Michel Fortier, and Michael H. Coffin. Monitor classification. *ACM Comput. Surv.*, 27(1):63–107, March 1995. doi:10.1145/214037.214100.

11 Alan Burns, Marina Gutiérrez, Mario Aldea Rivas, and Michael González Harbour. A deadline-floor inheritance protocol for EDF scheduled embedded real-time systems with resource sharing. *IEEE Trans. Computers*, 64(5):1241–1253, 2015. doi:10.1109/TC.2014.2322619.

12 Alan Burns and Andy J. Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison-Wesley Educational Publishers Inc, USA, 4th edition, 2009.

13 Jan Edler, Jim Lipkis, and Edith Schonberg. Process management for highly parallel UNIX systems. In *Proceedings of the USENIX Workshop on Unix and Supercomputers*, pages 1–17, September 1988.

14 Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Proceedings of the 2002 Ottawa Linux Symposium*, pages 479–495, 2002.

15 Paolo Gai, Giuseppe Lipari, and Marco Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001), London, UK, 2-6 December 2001*, pages 73–83. IEEE Computer Society, 2001. doi:10.1109/REAL.2001.990598.

16 Darren Hart and Dinakar Guniguntalay. Requeue-PI: Making Glibc Condvars PI-Aware. In *Eleventh Real-Time Linux Workshop*, pages 215–227, 2009.

17 Philip Holman and James H. Anderson. Locking in Pfair-scheduled multiprocessor systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02), Austin, Texas, USA, December 3-5, 2002*, pages 149–158. IEEE Computer Society, 2002. doi:10.1109/REAL.2002.1181570.

18 James Leslie Keedy. An outline of the ICL 2900 series system architecture. *Australian Computer Journal*, 9(2):53–62, 1977.

19 Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1–19. IEEE, 2019. doi:10.1109/SP.2019.00002.

20 Leonidas I. Kontothanassis, Robert W. Wisniewski, and Michael L. Scott. Scheduler-conscious synchronization. *ACM Trans. Comput. Syst.*, 15(1):3–40, 1997. doi:10.1145/244764.244765.

21 Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 973–990. USENIX Association, 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

22 Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, page 110–121, New York, NY, USA, 1991. Association for Computing Machinery. doi:10.1145/121132.344329.

23 Ross McIlroy, Jaroslav Sevcík, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *CoRR*, abs/1902.05178, 2019. arXiv:1902.05178.

24 John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991. doi:10.1145/103727.103729.

25 Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51(1):1–26, 1998. doi:10.1006/jpdc.1998.1446.

26 John K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems, Miami/Ft. Lauderdale, Florida, USA, October 18-22, 1982*, pages 22–30. IEEE Computer Society, 1982.

27 John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Usenix Summer 1990 Technical Conference, Anaheim, California, USA, June 1990*, pages 247–256. USENIX Association, 1990.

28 Filip Pizlo. Locking in webkit. online article, May 2016. URL: https://webkit.org/blog/6161/locking-in-webkit/.

29 R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *10th International Conference on Distributed Computing Systems (ICDCS 1990), May 28 - June 1, 1990, Paris, France*, pages 116–123. IEEE Computer Society, 1990. doi:10.1109/ICDCS.1990.89257.

30 David P. Reed and Rajendra K. Kanodia. Synchronization with eventcounts and sequencers. *Commun. ACM*, 22(2):115–123, February 1979. doi:10.1145/359060.359076.

31 Benoit Schillings. Be Engineering Insights: Benaphores. *Be Newsletters*, 1(26), May 1996.

32 Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990. doi:10.1109/12.57058.

33 Livio Soares and Michael Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In Remzi H. Arpaci-Dusseau and Brad Chen, editors, *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 33–46. USENIX Association, 2010. URL: http://www.usenix.org/events/osdi10/tech/full_papers/Soares.pdf.

34 Roy Spliet, Manohar Vanga, Björn B. Brandenburg, and Sven Dziadek. Fast on average, predictable in the worst case: Exploring real-time futexes in LITMUSRT. In *Proceedings of the IEEE 35th IEEE Real-Time Systems Symposium, RTSS 2014, Rome, Italy, December 2-5, 2014*, pages 96–105. IEEE Computer Society, 2014. doi:10.1109/RTSS.2014.33.

35 Alexander Wieder and Björn B. Brandenburg. On spin locks in AUTOSAR: blocking analysis of fifo, unordered, and priority-ordered spin locks. In *Proceedings of the IEEE 34th Real-Time Systems Symposium, RTSS 2013, Vancouver, BC, Canada, December 3-6, 2013*, pages 45–56. IEEE Computer Society, 2013. doi:10.1109/RTSS.2013.13.

36 Alexander Zuepke. Deterministic fast user space synchronisation. In *OSPERT Workshop*, July 2013.

37 Alexander Zuepke, Marc Bommert, and Daniel Lohmann. AUTOBEST: a united AUTOSAR-OS and ARINC 653 kernel. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium, Seattle, WA, USA, April 13-16, 2015*, pages 133–144. IEEE Computer Society, 2015. doi:10.1109/RTAS.2015.7108435.

38 Alexander Zuepke and Robert Kaiser. Deterministic futexes: Addressing WCET and bounded interference concerns. In Björn B. Brandenburg, editor, *25th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2019, Montreal, QC, Canada, April 16-18, 2019*, pages 65–76. IEEE, 2019. doi:10.1109/RTAS.2019.00014.

Modeling and Analysis of Bus Contention for Hardware Accelerators in FPGA SoCs

Francesco Restuccia
TeCIP Institute and Dept. of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Pisa, Italy
francesco.restuccia@santannapisa.it

Marco Pagani
TeCIP Institute, Scuola Superiore Sant'Anna, Pisa, Italy
Université de Lille, CNRS, Centrale Lille, UMR 9189, CRIStAL, Lille, France
marco.pagani@santannapisa.it

Alessandro Biondi
TeCIP Institute and Dept. of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Pisa, Italy
alessandro.biondi@santannapisa.it

Mauro Marinoni
TeCIP Institute and Dept. of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Pisa, Italy
mauro.marinoni@santannapisa.it

Giorgio Buttazzo
TeCIP Institute and Dept. of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Pisa, Italy
giorgio.buttazzo@santannapisa.it

Abstract

FPGA System-on-Chips (SoCs) are heterogeneous platforms that combine general-purpose processors with a field-programmable gate array (FPGA) fabric. The FPGA fabric is composed of a programmable logic in which hardware accelerators can be deployed to accelerate the execution of specific functionality. The main source of unpredictability when bounding the execution times of hardware accelerators pertains the access to the shared memories via the on-chip bus. This work is focused on bounding the worst-case bus contention experienced by the hardware accelerators deployed in the FPGA fabric. To this end, this work considers the AMBA AXI bus, which is the de-facto standard communication interface used in most the commercial off-the-shelf (COTS) FPGA SoCs, and presents an analysis technique to bound the response times of hardware accelerators implemented on such platforms. A fine-grained modeling of the AXI bus and AXI interconnects is first provided. Then, contention delays are studied under hierarchical bus infrastructures with arbitrary depths. Experimental results are finally presented to validate the proposed model with execution traces on two modern FPGA-based SoC produced by Xilinx (Zynq-7000 and Zynq-Ultrascale+ families) and to assess the performance of the proposed analysis.

2012 ACM Subject Classification Hardware → Interconnect; Hardware → Hardware accelerators

Keywords and phrases Heterogeneous computing, Predictable hardware acceleration, FPGA SoCs, Multi-Master architectures

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.12

Supplementary Material ECRTS 2020 Artifact Evaluation approved artifact available at `https://doi.org/10.4230/DARTS.6.1.4`.

1 Introduction

Next-generation *cyber-physical systems* (CPS) require the execution of complex computing workload such as machine learning algorithms and image/video processing. Representative examples include autonomous driving, advanced robotics, and smart manufacturing. In order to perform high-performance computations while matching the timing constraints imposed by

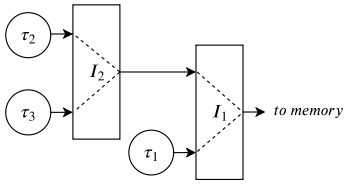


Figure 1 Example bus architecture with three HAs connected using two interconnects.

the physical world, these systems require coupling standard processing units with on-board *hardware accelerators* (HAs), which allow speeding up complex computations, especially those that are prone to large-scale parallelization. Heterogeneous computing platforms, such as system-on-chips (SoC) that integrate a multiprocessor with acceleration-oriented devices like field-programmable gate arrays (FPGAs) or general-purpose graphical processing units (GPGPUs) are de-facto establishing as the reference solutions to develop next-generation CPS. Examples of such platforms are the Zynq Ultrascale+ produced by Xilinx, which includes a large FPGA fabric, and the Xavier produced by Nvidia, which includes a GPGPU and other accelerators for machine learning algorithms.

A key issue when developing safety-critical CPS is to guarantee certain *timing constraints* for the control software. When hardware acceleration is used by critical software, the problem also extends to the consideration of the worst-case timing properties of HAs. Unfortunately, timing analysis for HAs can be particularly challenging, especially if very limited information on their internal architecture and resource management logic is publicly available, as it is the case for Nvidia platforms. To further complicate this issue, note that HAs are typically very memory-intensive. Indeed, they tend to work on a large amount of data (think of real-time video processing) and hence generate a consistent memory traffic that can have a paramount impact on their timing performance, especially when running together with other accelerators that cause contention at some stage on their path towards shared memories (such as buses and memory controllers).

FPGA-based heterogeneous platforms represent very promising solutions to cope with these issues. As a matter of fact, they allow deploying energy-efficient, yet powerful HAs on the FPGA fabric that have a very regular clock-level behavior [3, 21]. FPGA-based HAs are typically implemented as state machines and issue a fairly predictable pattern of bus transactions. As such, the execution times of HAs when running in isolation are characterized by extremely limited fluctuations, and are hence very predictable. The major phenomenon that harms the timing predictability of FPGA-based HAs that are statically programmed on the FPGA and access a shared memory, is the corresponding memory contention they can experience on the bus or at the memory controller.

Nevertheless, differently from other platforms, FPGAs expose a fine-grained control of the bus infrastructure to designers, which are free to organize the bus hierarchy at their own choice in order to match timing constraints, as well as to deploy custom arbitration modules to dispatch memory transactions towards the memory controller [1]. At last, designers are even free to deploy custom on-chip memories on the FPGA fabric for which they can have full control on how contention is regulated [18]. Such strategies can be used to achieve a higher degree of predictability for the memory traffic.

Focusing on most common approaches, FPGA designs for hardware acceleration in COTS SoCs typically consist of a set of accelerators that act as masters on the bus to access the main DRAM memory (off-chip) shared with the multiprocessor(s), e.g., see [27] [10] [33]. Being the number of ports to access the shared memory limited, the typical solution consists

in multiplexing multiple masters on the same port by means of *interconnects*, which are usually available in the standard library of devices offered by FPGA vendors. Interconnects can also be hierarchically connected to form a hierarchical bus network: an example of such networks comprising three HAs (τ_1, τ_2, and τ_3) and two interconnects (I_1 and I_2) is depicted in Figure 1. Clearly, the topology of the bus hierarchy has a primary impact on how the access to memory is regulated, and hence on the corresponding delays due to memory contention. For instance, assuming that both I_1 and I_2 in Figure 1 implement round-robin arbitration with the granularity of one memory transaction per HA per round-robin cycle, it is possible to note that τ_1 is privileged in accessing the memory. Indeed, once every two round-robin cycles I_1 could grant one memory transaction issued by τ_1, while in the other cycle transactions from τ_2 and τ_3 are alternated. At a very high level, τ_1 have a privileged access to the memory controller.

It is crucial to note that, in FPGA SoC, the operating frequency of the FPGA fabric is much lower than the on-chip memory controller (which is realized in hard silicon, i.e., placed outside the FPGA) and the memory itself. For instance, in a Zynq-7000 by Xilinx, the default operating frequency of the FPGA is 100MHz, while the Processing System, which includes the memory controller, runs at 650MHz. As such, the delays introduced by a bus infrastructure realized on the FPGA by means of interconnects are typically of the same order of magnitude of the ones required to access the memory, and hence do not consist of a negligible contribution to the response times of HAs.

Contribution. This paper studies bus contention and proposes a worst-case response time analysis for HAs deployed on FPGA-based SoCs. The AXI open bus standard [2] is considered because of the following reasons: **(i)** AXI is the de-facto standard communication interface for COTS FPGA SoC platforms [31] [13], **(ii)** AXI is widely supported by well-established FPGA design tools such as Xilinx Vivado [30] and Intel Quartus Prime [14], **(iii)** many commercial (closed-source) HAs use AXI interfaces. To begin, a fine-grained model for the AXI bus and AXI interconnects is presented (Sec. 3). The model accounts for several kinds of delays experienced by bus transactions and the behavior of commercial interconnects. Then, the paper presents a response-time analysis to bound the worst-case response time of recurrent HAs that access a shared memory via an arbitrary *hierarchical* network of interconnects (Sec. 4). Finally, three experimental evaluations (Sec. 5) are reported. First, a set of experimental results obtained from a state-of-the-art FPGA SoC by Xilinx are presented to validate the model proposed in this paper. Second, a case study executed on the same platform is discussed by matching measurements extracted from its execution with the bounds provided by the proposed analysis. Third, experimental results obtained with synthetic workload are presented.

2 Essential Background

A typical FPGA SoC architecture combines a *Processing System* (PS), which includes one or more processors, with a FPGA subsystem in a single device. Both subsystems access a shared DRAM controller through which they can access a DRAM memory. Figure 2 illustrates a typical SoC FPGA architecture in which two interfaces allow the communication between the FPGA subsystem and the processing system (PS). The de-facto standard interface for interconnections is the ARM Advanced Microcontroller Bus Architecture Advanced eXtensible Interface (AMBA AXI) [2].

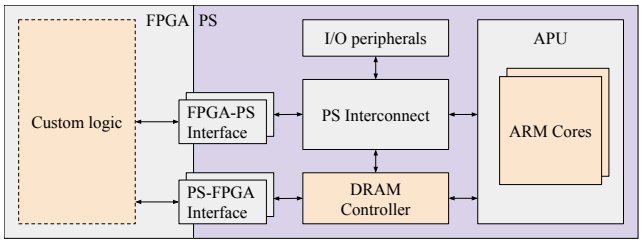


Figure 2 Simplified architecture of a SoC FPGA platform.

The AXI bus. The AMBA AXI standard defines a master-slave interface allowing simultaneous, bi-directional data exchange. An AXI *interface* (also referred to as port) is composed of five independent *channels*: Address Read (AR channel), Address Write (AW channel), Data Read (R channel), Data Write (W channel), and Write Response (B channel). This paper considers that data are transmitted back to the master on the R channel (for read data) or provided to the W channel (for write data) in the same order with which the corresponding requests have been routed to the address channel. In other words, address requests are served in-order, that is, *the access to the output data channels* R *and* W *depends on the order in which requests are routed to the address channels*. Even though this assumption does not directly derive from the standard, it is a popular design choice reported in the documentation of many commercial devices such as those produced by Xilinx [28, 31].

The AXI standard allows masters to issue multiple pending requests. This means that, in principle, each master is allowed to issue an unlimited number of outstanding transactions (typically limited by the designers of devices connected to the bus). AXI offers two methods for transmitting data between masters and slaves: single transactions or transaction *bursts*. When operating in burst mode, the requesting device can issue a single address request to fetch/write up to 256 data words per request.

AXI ports. As it is illustrated in Figure 2, The communication between the FPGA and the PS is allowed by two different types of interfaces: the PS-FPGA interface and the FPGA-PS interface. The first one offers a set of slave interfaces to the FPGA and is used by the processors to control the hardware devices or access data in the FPGA. In a dual manner, the second one offers a set of slave interfaces to the PS and is used by devices deployed on the FPGA (e.g., hardware accelerators) to access the central DRAM memory or the on-chip memory in the PS. Being the number of available ports in the FPGA-PS interface limited, scenarios in which a port is contended by multiple master devices deployed on the FPGA are common in realistic designs. To cope with the case in which bus contention is maximized, this paper is focused on the arbitration required to solve conflicts of requests that target the same output port. Nevertheless, the results of this work can also be easily extended to scenarios in which multiple ports are used.

AXI interconnects. Whenever multiple AXI masters want to access the same output port, an AXI *interconnect* is in charge of arbitrating conflicting requests to the same port. The access to each channel of the output AXI port is managed by a multiplexer. Each multiplexer is controlled by an arbiter that decides, at each time, which slave channel is granted to the master channel. The arbiters are completely independent from each other. Each port (slave or master) of the AXI interconnect is buffered with a FIFO queue (which is typically quite large). For instance, in FPGA SoCs by Xilinx, two implementations of the interconnect are

available: *AXI Interconnect* (deprecated in the latest platforms) and *AXI SmartConnect*. Both the implementations are multiplexer-based and therefore comply with the specification described above.

Arbitration policy. In this work, each arbiter is assumed to implement a round-robin policy. To the best of our records, round-robin is the most common solution in commercial off-the-shelf platforms. For instance, the AXI arbiters for FPGA SoCs by Xilinx implement round-robin (both the AXI interconnect and the AXI SmartConnect, see [35], p.6 and [32], p.7). Note that fixed-priority arbitration has been discontinued in the AXI SmartConnect. Furthermore, even though the AXI standard defines QoS signals to regulate the quality-of-service of transactions, these signals are ignored by state-of-the-art interconnects (see [35], p. 8 and [32], p. 9).

Hierarchical interconnection. State-of-the-art interconnects dispose of a limited amount of slave ports. However, AXI interconnects can even be connected between each other, creating a network tree of interconnects with multiple hierarchical levels. In such a structure, each inner node of the tree represents an interconnect, each leaf represents a master device, and the root node represents the sole interconnect connected to the slave port of the FPGA-PS interface (i.e., the sink of all the traffic towards the FPGA-PS interface). Thanks to such hierarchical structures, it is possible to connect as many devices as desired to a single AXI port of the FPGA-PS interface (provided that there is enough area on the FPGA to deploy all the modules). Clearly, the address requests (both read and write) and the data issued by a device connected at some interconnect I in a hierarchical network must traverse all the interconnects encountered on the path from I to the FPGA-PS. In a dual manner, write responses and the data read by the same device must traverse the same path in reverse order, i.e., from the FPGA-PS interface to I. Note that, due to the intrinsic parallelism of the AXI bus, and the fact that each interconnect is an independent engine that executes in parallel with the others, a network of interconnects exhibits a pipelined behavior.

Read transactions. A general read transaction issued by a master device τ starts with the issue of the address request R_{addr} on the AR channel of its master port M_τ, which is sampled by the corresponding slave port of the AXI interconnect to which τ is directly connected. R_{addr} is then routed through a network of one or multiple AXI interconnects until reaching the FPGA-PS interface (and then the memory controller). After a service delay related to the logic in the Processing System, the memory controller, and the DRAM memory, the requested data R_{data} become available on the R channel of the FPGA-PS interface. Hence, data are routed back to τ through the same interconnect network traversed by R_{addr}, but in reverse order. Once available at M_τ, data R_{data} are sampled by τ, hence completing the read transaction.

Write transactions. A general write transaction issued by a master device τ starts with the issue of the address request W_{addr} on the AW channel of its master port M_τ, which is sampled by the corresponding slave port of the AXI interconnect I to which τ is directly connected. W_{addr} is then routed through a network of one or multiple AXI interconnects until reaching the FPGA-PS interface (and eventually the memory controller). In parallel, once W_{addr} is granted by I, the corresponding data W_{data} are provided by τ to the W channel of its master port and flow through the path reaching the FPGA-PS interface following W_{addr} (i.e., reaching the Processing System and then the memory controller). After a service delay

(introduced by the PS, the memory controller, and the DRAM memory), the Processing System provides a write response W_{resp} on the B channel of the FPGA-PS interface to acknowledge τ. W_{resp} is routed from the FPGA-PS interface through the same network of interconnects traversed by W_{addr} and W_{data}, but in reverse order. Once available at M_τ, W_{resp} is sampled by τ and the write transaction is completed.

3 System model

This section focuses on modeling the components of a system comprising a set of AXI-based hardware accelerators, deployed on the FPGA fabric of a FPGA-SoC platform and connected to a shared DRAM memory on the Processing System through the FPGA-PS interface.

3.1 Hardware task model

Each hardware accelerator implements a specific functionality; therefore, from now on, they are referred to as *hardware tasks* (HW-tasks for short). Each HW-task includes an AXI memory-mapped master interface through which it can autonomously load and store data from the DRAM memory. Each HW-task τ_i is periodically executed every T_i clock cycles, hence generating an infinite sequence of periodic instances referred to as *jobs*. Each job of τ_i **(i)** issues at most N_i^R read transactions and N_i^W write transactions, both with a fixed burst size B; **(ii)** issues at most ϕ_i outstanding transactions per type, i.e., it can have at most ϕ_i pending read transactions and ϕ_i pending write transactions at any time; **(iii)** computes for at most C_i clock cycles; and **(iv)** has a relative deadline equal to T_i (each job must complete before the release of the next one). Read transactions and write transactions are supposed to be independent. Furthermore, note that read and write transactions are routed through independent AXI channels, that is, they do not influence each other when the corresponding data is transmitted.

It is important to observe that no specific memory access pattern for the HW-tasks is assumed, i.e., the requests for memory transactions can be arbitrarily distributed over time across the jobs. This assumption makes the results presented in this paper more general and robust with respect to the HW-tasks' behavior. However, at the same time, it limits the number of timing properties related to bus pipelining that can be used at the stage of analysis as, in the worst-case, transactions can be sufficiently spread far apart such that HW-tasks do not fully exploit pipelining.

3.2 AXI interconnect model

The system can comprise several interconnects connected in a hierarchical fashion. Each interconnect I_j has S_j slave ports and one master port. As each interconnect has a single master port, the incoming traffic (at the master port and) directed to the slave ports does not experience any conflict. On the other hand, address requests of the same type (read or write) issued by different HW-tasks can experience conflicts, which are managed by independent, per-channel, arbiters (see Section 2). The granularity of the round-robin arbiters is ϕ_I, i.e., at each round-robin cycle the master port grants at most ϕ_I read requests (resp., write requests) to each HW-task. To ease the notation in the analysis presented in Section 4, it is assumed that all the interconnects in the system share the same parameter ϕ_I (the analysis can be easily extended to the case of different per-interconnect round-robin granularities). Finally, it is assumed that the FIFO queues associated with the ports of the interconnects

are large enough to never saturate during the execution[1]. Each interconnect introduces a propagation delay in address and data propagation. Specifically, we denote by $d_{\text{Int}}^{\text{addr}}$ the latency introduced in the propagation of address requests, by $d_{\text{Int}}^{\text{data}}$ the latency introduced in the propagation of a word of data (read or write), and by $d_{\text{Int}}^{\text{bresp}}$ the latency introduced in the propagation of a write response. These propagation delays can be derived from the specifications in the official documentation of the considered interconnect (when available) or by employing experimental profiling. The AXI standard defines hold times as the numbers of clock cycles that the address or data must be kept on the corresponding AXI channel while both *valid* and *ready* signals are asserted. Address and data hold times are modeled with the following terms: t_{addr} denotes the hold time of an address request, t_{data} denotes the hold time of a word of data, and t_{bresp} denotes the hold time of a write response.

3.3 Processing System and Memory Controller model

The DRAM memory controller is a global system resource shared among all HW-tasks. Being part of the Processing System, it is accessed from the FPGA fabric through the FPGA-PS interface. Each port of the FPGA-PS interface can be configured to map a contiguous range of addresses, which is referred to as a memory region. As typical for hardware acceleration, it is assumed that each HW-task loads and stores data from a private memory buffer. To address the case in which the maximum contention is experienced, we focus on the case in which all the memory buffers are allocated in the same memory region and accessed through a single AXI port at the FPGA-PS interference. Note that the results of this work can also be extended to the case in which the HW-tasks access the DRAM memory via multiple ports at the FPGA-PS interface: this case is left as future work due to lack of space.

The DRAM memory controller included in the Processing System can be conceptually divided into two main blocks: **(i)** the AXI interface block and **(ii)** the DDR physical core block. The AXI interface block is in charge of receiving and arbitrating the incoming AXI transactions from the AXI slave ports, while the DDR physical core schedules and issues the corresponding read and write requests to the controller's physical layer, which eventually drives the DRAM memory by generating control and data signals.

Typically, the internal architecture of the DDR physical core includes multi-level queues structures, managed with dedicated scheduling policies that reorder transactions to maximize throughput and efficiency [11]. On many commercial platforms, the internals of the DDR physical core block, including the scheduling policies and the queues structure, are not publicly disclosed or are not well documented. For this reason, a fine-grained modeling of the DDR physical core block goes beyond the scope of this paper and it is not addressed here. Rather, being our focus on the conflicts at the interconnects, a coarse-grained modeling of the DRAM-related delays is adopted here: if the internals of the DDR controller are known, then our results can be refined (e.g., by adopting the results from [11]).

From the perspective of the FPGA-PS interface, address requests directed to the DDR memory controller are served *in order* (see [28], p. 297, and [31], p. 440). This means that the order of the data read responses on the data read channel follows the order of the address read requests granted at the address read channel. In the same way, write address requests are served and acknowledged in order. These properties are guaranteed by the

[1] Note that ensuring this condition is an orthogonal problem to timing analysis. That is, it is an a-priori requirement that can be verified independently of the timing performance of the system.

DRAM Memory Controller AXI Interface block. Note that this feature is independent of the internal scheduling policies of the DDR Physical core block, which may include internal reordering, hence affecting the worst-case service time of a request.

Following these considerations, this work assumes that the Processing System and the memory controller introduce the following (cumulative) delays:

- $d_{\text{PS}}^{\text{read}}$ is the maximum time elapsed between the sample of a read transaction at the FPGA-PS interface and the availability of the first word of the corresponding data at the FPGA-PS interface; and
- $d_{\text{PS}}^{\text{write}}$ is the maximum time elapsed between the sample of the last word of data of a write transaction at the FPGA-PS interface and the availability of the corresponding write response at the FPGA-PS interface.

Note that, by definition, these delays include the propagation times introduced by the internal logic of the Processing System and the overall service time at the memory controller. These parameters depend on the internals of the Processing System and can be quantified using the documentation provided by the SoC producer (when available) or through experimental profiling (and over-provisioning).

3.4 Overall architecture

Formally, the system is composed of a set $\Gamma = \{\tau_1, \ldots, \tau_n\}$ of n HW-tasks, a set $\mathcal{H} = \{I_1, \ldots, I_s\}$ of s AXI interconnects, and a memory controller $\mathcal{M}$ included in the Processing System. The HW-tasks in Γ are interconnected through a network of the AXI Interconnects in the set $\mathcal{H}$ that is organized as follows. Each slave port of the AXI interconnects can be directly connected to the master port of a HW-task or, in a hierarchical manner, to the master port of another interconnect. The set of HW-tasks directly connected to interconnect I_j is denoted by $\Gamma(I_j)$. Similarly, the set of interconnects directly connected to the slave ports of I_j (i.e., in input) is denoted by $\mathcal{H}(I_j)$. Furthermore, the set of HW-tasks whose transactions traverse I_j is denoted by $\Gamma^+(I_j)$, i.e., those that are directly or transitively connected to I_j. The interconnect at the bottom of this hierarchy has its master port directly connected to the slave port of the FPGA-PS interface (i.e., to reach $\mathcal{M}$). All the transactions issued by the HW-tasks must pass through this latter interconnect, which is referred to as the *root* interconnect I_{root}. Note that, as interconnects have a single master port, the master port of each interconnect $I_j \neq I_{\text{root}}$ is connected to a slave port of exactly one interconnect, which is denoted by $\beta(I_j)$. For consistency, $\beta(I_{\text{root}}) = \emptyset$. The topology of the whole system resembles a tree where I_{root} is the root node, the HW-tasks in Γ are the leaves, and the interconnects in $\mathcal{H} \setminus \{I_{\text{root}}\}$ are the intermediate nodes (see Figure 3(b)). An interconnect I is said to be placed at the *hierarchical level* L_I if a HW-task directly connected to I has to traverse L_I interconnects before reaching the FPGA-PS interface (I_{root} is at first level, i.e., $L_{I_{\text{root}}} = 1$). The main symbols used in the paper are summarized in Table 1.

4 Response-time analysis

This section proposes an analysis to bound the response times of HW-tasks connected to an arbitrary hierarchical network of interconnects as presented in the previous section.

The analysis is structured in incremental lemmas. First, Section 4.1 proposes a bound on the worst-case response time for a single transaction assuming no contention at the interconnects. Both read and write transactions are considered. Subsequently, Section 4.2 and Section 4.3 propose two different methods to bound the number of interfering transactions that affect a job of a HW-task under analysis. These two bounds are then combined in

Table 1 Main symbols used throughout the paper.

N_i	Number of transactions issued by τ_i (can have superscript R or W)
ϕ_i	Maximum number of outstanding transactions for τ_i
ϕ_I	Max. number of trans. granted per round-robin cycle by interconnects
B	Burst size of a transaction
t_{addr}	Hold time for a single address request on the bus
t_{data}	Hold time for a single data word on the bus
t_{bresp}	Hold time for a single write response on the bus
$d_{\text{PS}}^{\text{read}}$	Max. latency introduced by the PS on a read transaction
$d_{\text{PS}}^{\text{write}}$	Max. latency introduced by the PS on a write transaction
$d_{\text{Int}}^{\text{data}}$	Propagation latency of data word through an interconnect
$d_{\text{Int}}^{\text{addr}}$	Propagation latency of address request through an interconnect
$d_{\text{Int}}^{\text{bresp}}$	Propagation latency of write response through an interconnect
$\Gamma(I_i)$	Set of the HW-task directly connected to I_j
$\mathcal{H}(I_j)$	Set of the interconnects directly connected to slave ports of I_j
$\beta(I_j)$	Interconnect connected to the master port of I_j
$\Gamma^+(I_j)$	Set of HW-tasks whose transactions pass through I_j

Section 4.4. Finally, Section 4.5 presents an iterative algorithm that uses the results of the previous sections to bound the maximum response time of a HW-task of interest.

As the AXI standard defines the same methods to handle both read and write address requests, the bounds derived in this section hold for both read and write transactions. For this reason, in order to keep a compact notation, this section uses just the symbol N_i instead of N_i^R or N_i^W to denote the number of transactions issued by τ_i.

4.1 No contention at the interconnects

This first lemma establishes an upper bound on the response time of a single memory transaction issued by an arbitrary HW-task under analysis τ_i, connected to an interconnect I placed at an arbitrary hierarchical level L, assuming no bus contention from the other HW-tasks in the system[2].

Remember that AXI manages read and write transactions on independent channels: as such, they are separately considered by the following two lemmas.

▶ **Lemma 1.** *Let $\tau_i \in \{\Gamma\}$ be the HW-task under analysis, connected to an interconnect $I_j \in \mathcal{H}$ placed at the hierarchical level L. If all the HW-tasks in $\Gamma \setminus \{\tau_i\}$ are not active, i.e., they do not interfere with τ_i, the worst-case response time for a single read transaction R issued by τ_i is bounded by*

$$d^{NoCont,read}(I_j) = L \cdot (t_{addr} + d_{Int}^{addr}) + d_{PS}^{read} + L \cdot d_{Int}^{data} + B \cdot t_{data}.$$

Proof. Following Section 2, a read transaction R begins with the issue of the address read request R_{addr}, which is then sampled by I_j. The address time is constant and equal to t_{addr}. The latency cost for R_{addr} to traverse the interconnect I_j is bounded by $d_{\text{Int}}^{\text{addr}}$. At this point,

[2] It is worth noting that this contention-free bound does not properly correspond to the case in which the transaction under analysis is served in isolation, but rather just to the case in which no contention is experienced at the interconnects. This is because, for the reasons discussed in Section 3.3, the delay related to the Processing System and the memory controller already cope with conditions of maximum contention.

R_{addr} goes through the interconnect network tree, traversing the remaining $L-1$ interconnects. As for I_j, each of them introduces a latency bounded by $t_{\text{addr}} + d_{\text{Int}}^{\text{addr}}$. Therefore, R_{addr} is available at the master port of the root interconnect I_{root} after an overall propagation delay of $L \cdot (t_{\text{addr}} + d_{\text{Int}}^{\text{addr}})$, where it is sampled from the slave port of the FPGA-PS interface. The Processing System routes R_{addr} to the Memory Controller and provides to the FPGA-PS interface the first word of data after at most d_{PS}^{read} time units (see Section 3.3). At this point, the data words R_{data} corresponding to R traverse the L levels of the interconnect tree, in reverse order with respect to R_{addr}, until reaching τ_i. Since data words are sequentially propagated on the interconnect tree, the propagation latency in the data phase is paid just once on all the burst of data due to pipelining. Hence, considering that t_{data} is the data time (for each word) and that $d_{\text{Int}}^{\text{data}}$ is the latency introduced by each interconnect on data words, the overall latency paid to propagate the data burst on the interconnect tree is $L \cdot (t_{\text{data}} + d_{\text{Int}}^{\text{data}})$. The lemma following by summing up these contributions. ◀

▶ **Lemma 2.** *Under the same hypotheses of Lemma 1, the response time for a write transaction W issued by HW-task τ_i is bounded by*

$$d^{NoCont,write}(I_j) = L \cdot (t_{addr} + \max\{d_{Int}^{addr}, d_{Int}^{data}\}) + B \cdot t_{data} + d_{PS}^{write} + L \cdot (t_{bresp} + d_{Int}^{bresp}).$$

Proof. The write transaction W begins with the issue of the address write request W_{addr} by τ_i, which lasts t_{addr} time units . Following the AXI standard, once W_{addr} is granted at the interconnect I_j, the HW-task τ_i is granted to provide the corresponding data words W_{data} on the write channel. W_{addr} and W_{data} are propagated through the interconnect network tree on the corresponding (parallel) channels, until reaching the FPGA-PS interface. Data can be propagated only after the corresponding address; hence, the latency experienced by W_{addr} and W_{data} to traverse an interconnect is no larger than the maximum between $d_{\text{Int}}^{\text{addr}}$ and $d_{\text{Int}}^{\text{data}}$. Overall, considering all the interconnects up to the FPGA-PS interface, the latency introduced on W_{addr} and the entire burst W_{data} is given by $L \cdot (t_{\text{addr}} + \max\{d_{\text{Int}}^{\text{addr}}, d_{\text{Int}}^{\text{data}}\})$, which must be summed to the time to transmit the data themselves, i.e., $B \cdot t_{\text{data}}$. At this point, the Processing System routes W_{addr} and W_{data} to the memory controller. Following Section 3.3, after at most d_{PS}^{write} time units the write response W_{resp} is available at the FPGA-PS interface. Finally, W_{resp} is propagated through the interconnect tree, until reaching τ_i, experiencing a latency of $L \cdot (t_{\text{bresp}} + d_{\text{Int}}^{\text{bresp}})$. The lemma follows by summing up these contributions. ◀

It is worth noting that the bounds provided by the two above lemmas just depend on the hierarchical level L (identified by the interconnect I_j) at which a HW-task is directly connected.

4.2 First bound on the number of interfering transactions

We proceed in an incremental manner by starting from the simple case in which contention at a single interconnect is considered, say I_{root} for simplicity (see Figure 3(a)). The following lemma establishes a bound on the number of interfering transactions (issued by other HW-tasks) that a transaction issued by the HW-task under analysis can suffer.

▶ **Lemma 3.** *Consider the interconnect I_{root} and let $\tau_i \in \Gamma(I_{root})$ be the HW-task under analysis. In the worst-case, each address request for transaction issued by τ_i grants the access to the master port of I_{root} after at most*

$$\sum_{\tau_j \in \Gamma(I_{root}) \setminus \{\tau_i\}} \min(\phi_j, \phi_I) \tag{1}$$

transactions.

Proof. As mentioned in Section 3, the interconnects implement a round robin arbitration to solve conflicts on address requests issued by different HW-tasks. In the worst-case scenario, τ_i is the last HW-task served in the round robin arbitration cycle, i.e., after all the other HW-tasks in $\Gamma(I_{\text{root}})$. From the model in Section 3.1, each HW-task τ_j can have at most ϕ_j pending transactions. On the other hand, from the model in Section 3.2, the maximum number of transactions granted to each HW-task for each round robin cycle by interconnects is equal to ϕ_I. For these reasons, I_{root} grants at most $\min(\phi_j, \phi_I)$ for each interfering HW-task $\tau_j \in \Gamma \setminus \{\tau_i\}$ per round-robin cycle. The lemma follows by summing up the contribution of each interfering HW-tasks. ◀

With the above lemma in place, we can proceed to bound the number of interfering requests under a general hierarchical network of interconnects. We note that a HW-task τ_i can incur two kinds of interference: **(i)** *direct interference*, which is the one that τ_i's transactions experience at the interconnect to which τ_i is directly connected to; and **(ii)** *indirect interference*, which is the one that τ_i's transactions, or other transactions that generate direct interference to τ_i, experience in other interconnects at shallower hierarchical levels on their way towards the FPGA-PS interface. Further details on both kinds of interference are provided in the following.

Direct interference. The same reasoning used for Lemma 3 can be extended when considering a hierarchical network of interconnects such as the one illustrated in Figure 3(b). Note that, in such a case, a HW-task can also experience contention at an interconnect due to transactions coming from other interconnects placed at higher hierarchical levels. For instance, in Figure 3(b), τ_i (directly connected to I_{root}) can incur in a contention due to a transaction issued by τ_z, which is directly connected to I_1.

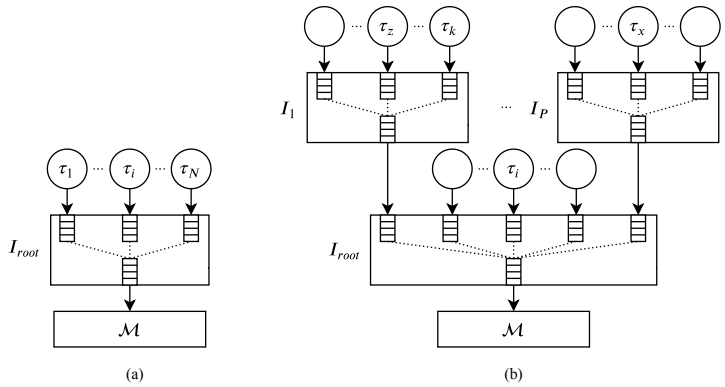


Figure 3 **(a)** A set of HW-tasks directly connected to I_{root}. **(b)** Example hierarchical network of interconnects and HW-tasks with two hierarchical levels. Each circle corresponds to a HW-task (only the ones mentioned in the text are assigned a name).

▶ **Lemma 4.** *Consider an arbitrary interconnect I_j. Also, let $\tau_i \in \Gamma(I_j)$ be the HW-task under analysis. In the worst-case, each address request for transaction issued by τ_i grants the access to the master port of I_j after at most*

$$Y^{direct}(\tau_i, I_j) = \sum_{\tau_j \in \Gamma(I_j) \setminus \{\tau_i\}} \min(\phi_j, \phi_I) + |\mathcal{H}(I_j)| \times \phi_I \tag{2}$$

transactions.

Proof. Following the model of Section 3.2, at each round-robin cycle I_j grants at most ϕ_I transactions per its slave port. Clearly, this consideration is true also when an interconnect I_h, placed at a higher hierarchical level, is connected to a slave port of I_j. Hence, from the perspective of τ_i, any bus traffic coming from I_h can interfere by at most ϕ_I transactions per round-robin cycle, i.e., independently of the number of HW-tasks or interconnects connected to I_h. Hence, the interconnects directly connected to I_j interfere with at most $|\mathcal{H}(I_j)| \times \phi_I$ transactions. The first term of Eq. (2) follows due to the same considerations done for Lemma 3. Hence the lemma follows. ◀

Indirect interference. While propagated through a series of interconnects on their way towards the FPGA-PS interface, the transactions issued by the HW-task under analysis can also incur contention at shallower hierarchical levels. For instance, the transactions issued by τ_z in Figure 3(b) can incur contention at I_{root}, e.g., due to other transactions issued by τ_i or τ_x. Furthermore, note that indirect interference can also affect other transactions that generate direct interference to the HW-task under analysis, hence leading to a *transitive* interference phenomenon. For instance, still considering Figure 3(b), a transaction issued by τ_k that delays τ_z in I_1 can experience contention at I_{root} due to a transaction issued by τ_x, hence in turn delaying τ_z too: in this case, we say that a transaction of τ_x transitively delays τ_z.

In the following, a set of lemmas are presented to account for indirect interference. We proceed in an incremental manner by starting from the consideration of just two *adjacent* hierarchical levels.

▶ **Lemma 5.** *Consider an arbitrary interconnect I_j at hierarchical level $L \geq 2$ that issues Δ transactions in output to its master port. In the worst-case scenario, the Δ transactions can be indirectly interfered by*

$$Y^{indirect}_{2\text{-}level}(\Delta, I_j) = \Delta \times \left(\sum_{\tau_i \in \Gamma(\beta(I_j))} \min(\phi_i, \phi_I) + |\mathcal{H}(\beta(I_j)) \setminus \{I_j\}| \times \phi_I \right) \tag{3}$$

transactions at $\beta(I_j)$ (i.e., at hierarchical level $L-1$).

Proof. Consider one of the Δ transactions, say r. As addressed by Lemma 4, r can incur direct interference at the (only) interconnect $\beta(I_j)$ directly connected to I_j at hierarchical level $L-1$. As such, the interference at $\beta(I_j)$ can be bounded as for Lemma 4. The only differences here are that **(i)** as r comes from another interconnect I_j, it means that it has not been originated by a HW-task connected to $\beta(I_j)$ and hence no HW-task has to be excluded from those that generate interfering transactions (first term in the sum of Eq. (2)); and **(ii)** I_j has to be excluded from the interconnects that generate interfering transactions as it is the one from which the interfered transaction is coming from (second term in the sum Eq. (2)). Note that the second term in the multiplication of Eq. (3) serves this purpose. The lemma follows by accounting for this bound for each of the Δ transactions. ◀

With the above lemma in place, it is possible to generalize the bound of indirect interference to an arbitrary hierarchical structure with $L > 2$ levels.

▶ **Lemma 6.** *Let τ_z be the HW-task under analysis directly connected to interconnect I_j at the hierarchical level $L \geq 2$. The total number of transactions that interfere with those issued by τ_z up to the l-th hierarchical level, with $l \in [1, L]$, is bounded by Y_z^l, which is recursively defined as follows for $l < L$:*

$$\begin{cases} Y_z^l = Y_{2\text{-}level}^{indirect}(N_z + Y_z^{l+1}, I^{l+1}) + Y_z^{l+1} \\ I^l = \beta(I^{l+1}), \end{cases}$$

and as follows for $l = L$ (base case):

$$\begin{cases} Y_z^L = N_z \times Y^{direct}(\tau_z, I_j) \\ I^L = I_j. \end{cases}$$

Proof. The proof is by induction on the hierarchical level $l \in [1, L]$. We also show that I^l is the interconnect traversed by τ_z's transactions at the l-th hierarchical level. **Base case:** At hierarchical level L, τ_z is directly connected to I_j: hence, $I^L = I_j$ and τ_z suffers direct interference only. Therefore, the number of interfering transactions up to the L-th hierarchical level is bounded by accounting for the bound provided by Lemma 4 for each of the N_z transactions issued by τ_z. **Inductive case:** We proceed by assuming that Y_z^{l+1} yields a safe bound for the number of interfering transactions up to the $(l+1)$-th hierarchical level and that I^{l+1} is the interconnect traversed by τ_z's transactions at the same level. Now, we show that Y_z^l provides a safe bound for the l-th hierarchical level. First, note that by definition, $I^l = \beta(I^{l+1})$ denotes the (only) interconnect across which τ_z's transactions can pass at the l-th hierarchical level. Second, observe that the transactions that are received in input by I^l and that affect τ_z's execution are **(i)** those issued by τ_z itself and **(ii)** those that generated interference to τ_z at the interconnects traversed at higher hierarchical levels. The former are no more than N_z (by the model), while the latter are Y_z^{l+1} (by inductive assumption). Such requests are coming from I^{l+1} and can incur indirect interference at I^l, which can be bounded by Lemma 3 as $Y_{\text{2-level}}^{\text{indirect}}(N_z + Y_z^{l+1}, I^{l+1})$. To bound the overall number of interfering requests up to the l-th hierarchical level, it then remains to account for all the (direct and indirect) interference collected at the higher levels, which is given by Y_z^{l+1} (by inductive assumption). Hence the lemma follows. ◀

Thanks to the above lemma, the total number of transactions that interfere with τ_z (under analysis) *across the entire hierarchical network of interconnects* can be bounded by looking at the interference collected up to the root interconnect, i.e., Y_z^1.

4.3 Second bound on the number of interfering transactions

A different approach can be used to derive an alternative bound on the number of interfering transactions by leveraging the observation that the HW-tasks are periodically executed, and hence can only generate a limited number of transactions in a given time window.

▶ **Lemma 7.** *Let τ_i be the HW-task under analysis and let I^l the interconnect traversed by τ_i's transactions at the l-th hierarchical level. In a schedulable system, the number of transactions that can interfere with τ_i up to I^l is bounded by*

$$Y^{time}(\tau_i, I^l) = \sum_{\tau_j \in \Gamma^+(I^l) \setminus \{\tau_i\}} \eta_{i,j}, \qquad \text{where} \quad \eta_{i,j} = \left\lceil \frac{T_i + T_j}{T_j} \right\rceil \times N_j.$$

Proof. Consider HW-task τ_i and assume all HW-tasks never execute after their deadlines[3]. Without loss of generality, suppose that a period instance of τ_i begins at time 0. To interfere with τ_i, a job of another HW-task τ_j must be released after time $-T_j$, otherwise, it would be completed when τ_i is released. In the same way, an interfering job of τ_j must be released before time T_i, otherwise τ_i would already be completed and hence no contention can be generated. As a result, the time window of interest to study the contention generated by τ_j to τ_i is $(-T_j, T_i]$ with length $T_j + T_i$. In this time window there can be at most $\lceil (T_i + T_j)/T_j \rceil$ jobs of τ_j. As each job of τ_j can issue at most N_j transactions, there are at most $\lceil (T_i + T_j)/T_j \rceil \times N_j$ transactions that can interfere with τ_i. The number of interfering transactions is hence bounded by the sum of such contributions from each HW-task that can interfere with τ_i. Note that only the HW-tasks whose transactions traverse I^l can interfere at I^l: according to the system model, the set of such tasks is $\Gamma^+(I^l)$. Clearly, τ_i has to be excluded from $\Gamma^+(I^l)$ as it cannot interfere with itself. Hence the lemma follows. ◀

4.4 Combining the two bounds

This lemma combines the bounds proposed in Section 4.2 and Section 4.3 to introduce a less pessimistic bound on the overall number of interfering transactions for an arbitrary interconnect architecture tree and HW-task set. The proposed formula is iterative on the interconnect levels. Iterating the formula for each interconnect in the path, from the interconnect to which the HW-task under analysis is directly connected to I_{root}, it is possible to find the overall number of interfering transactions a request under analysis issued by the HW-task under analysis suffers.

▶ **Lemma 8.** *In a schedulable system, the same claim of Lemma 6 still holds if Y_z^l is recursively defined as follows for $l < L$:*

$$\begin{cases} Y_z^l = \min\left(Y_{2\text{-}level}^{indirect}(N_z + Y_z^{l+1}, I^{l+1}) + Y_z^{l+1}, Y^{time}(\tau_z, I^l)\right) \\ I^l = \beta(I^{l+1}), \end{cases}$$

and as follows for $l = L$ (base case):

$$\begin{cases} Y_z^L = \min\left(N_z \times I^{direct}(\tau_z, I_j), Y^{time}(\tau_z, I^L)\right) \\ I^L = I_j. \end{cases}$$

Proof. The lemma follows as for Lemma 6 after recalling that both Lemma 6 and Lemma 7 provide a safe bound on the number of transactions that can interfere with τ_z. Hence, the minimum of the two bounds is still a safe bound. ◀

4.5 Response-time analysis algorithm

Leveraging the results of the previous sections, this section presents an algorithm to bound the worst-case response time of HW-tasks connected at arbitrary hierarchical levels. While the lemmas presented in the previous sections allow bounding the *number* of interfering transactions, this section is concerned with assigning a contention cost to them in order to obtain the corresponding temporal interference.

[3] Assuming a schedulable system to bound response times is a typical approach when circular dependencies are present in the response-time equations. The interested reader is invited to refer to [20] (Sec. VI.C) for an explanation about why this is a sound approach to bound response times.

To begin, note that the contention cost associated to each interfering transaction is not constant: indeed, following the model of Section 3, transactions experience a propagation delay each time they traverse an interconnect. Hence, a transaction that interferes at a high hierarchical level generates more delay than another one that interferes at a shallower hierarchical level.

Clearly, given a HW-task τ_z under analysis, a safe bound can be obtained by first computing Y_z^1 from Lemma 8, which provides a bound on the number of interfering transactions across the whole hierarchical network of interconnects (i.e., up to I_{root}), and then multiplying Y_z^1 by the largest contention cost, i.e., the one related to the highest hierarchical level. However, a more accurate bound can be devised if a level-specific contention cost is accounted for each transaction by detecting the highest hierarchical level at which it can interfere.

Algorithm 1 Bounding the worst-case contention delay experienced by τ_z due to interfering transactions across the whole hierarchical network of interconnects.

Input: HW-task $\tau_z \in \Gamma$ directly connected to I_j at level L
Output: d_z^{interf}
$d_z^{\text{interf}} \leftarrow 0$
$I^L = I_j$
$N^{\text{acc}} \leftarrow 0$
for $l = L, L-1, \ldots, 1$ **do**
  $N^l \leftarrow Y_z^l$ from Lemma 8
  $d_z^{\text{interf}} \leftarrow d_z^{\text{interf}} + (N^l - N^{\text{acc}}) \times d^{\text{NoCont}}(I^l)$
  $I^{l-1} = \beta(I^l)$
  $N^{\text{acc}} \leftarrow N^l$
end
return d_z^{interf}

This strategy is implemented by Algorithm 1. As mentioned at the beginning of Section 4, read and write transactions are independently managed by AXI and hence can be separately treated. For this reason, analogously as for the lemmas presented above, Algorithm 1 holds for both read and write transactions. To avoid duplicating its definition, the algorithm considers a contention cost $d^{\text{NoCont}}(I_j)$ that has to be replaced with $d^{\text{NoCont,read}}(I_j)$ or $d^{\text{NoCont,write}}(I_j)$ depending on the type of transactions that are studied. Consequently, the algorithm can be used to produce two outputs, which to keep a compatible notation are named $d_z^{\text{interf,read}}$ and $d_z^{\text{interf,write}}$. In essence, the algorithm iterates over all hierarchical levels interested by the HW-task τ_z under analysis (from $l = L$ to $l = 1$) and copes with the number of interfering transactions collected up to each interconnect traversed by τ_z transactions. For each interconnect I^l traversed at the l-th hierarchical level, the algorithm accounts for the contention delay of the interfering transactions that insist on I^l but have not been accounted at a higher hierarchical level. As said before, this is because the contention cost $d^{\text{NoCont}}(I_j)$ is monotone with the hierarchical level (the higher the level the larger the cost).

Thanks to this algorithm, it is finally possible to bound the worst-case response time of each HW-task, which is given by **(i)** its worst-case execution time, **(ii)** the time required to perform its read and write transactions, and **(iii)** the contention delay experienced by the latter. Hence, for each HW-task τ_z connected to interconnect I_j it is bounded by

$$\mathcal{R}_z = C_z + N_z^R \times d^{\text{NoCont,read}}(I_j) + N_z^W \times d^{\text{NoCont,write}}(I_j) + d_z^{\text{interf,read}} + d_z^{\text{interf,write}}. \quad (4)$$

A system is then schedulable if all HW-tasks meet their deadlines, i.e., if $\mathcal{R}_z \leq T_z, \forall \tau_z \in \Gamma$.

5 Experimental results

This section first presents an experimental evaluation that has been conducted to validate the system model and assess the performance of the proposed analysis (Section 5.3). The experiments have been performed on two state-of-the-art Xilinx SoC FPGA platforms, namely the Zynq-7020 and the ZCU102 Zynq Ultrascale+. On both platforms, one of the high-performance (HP) ports of the Processing System is used in the FPGA-PS interface. Due to the lack of space, this section reports only the results of the experiments performed on Zynq Ultrascale+, since the Zynq-7000 exhibit comparable behavior. Finally, Section 5.4 reports other experimental results obtained with the synthetic workload.

5.1 Experimental setup

In order to perform a clock-level accurate evaluation, two custom IPs have been developed: a programmable traffic generator IP, named *greedy* HW-task (GHW-task for short), and a multichannel *timer* IP. The purpose of the GHW-task IP is to generate, in a controllable way, cycle-accurate patterns of transactions compliant with the AXI standard with arbitrary offsets, spacing, burst size, and maximum number of outstanding transactions. GHW-tasks have been developed to cope with any possible bus behavior of HW-tasks, i.e., they can mimic any kind of pattern of bus transactions issued by real-world, memory-intensive HW-tasks, and are hence useful to stress bus contention. On the other hand, the multichannel *timer* IP is used to perform clock-level accurate measurements of the GHW-tasks' response times without perturbating their execution. Both IPs have been synthesized and implemented using Xilinx Vivado 2018.2. The FPGA clock is set to the default value (100 MHz), while the Processing System runs at the default clock speed of 1.2 GHz.

5.2 Profiling

This set of experiments aims at characterizing the propagation *delay* and the *hold* times, introduced in Section 3.2, for the AXI SmartConnect. To this end, a test setup composed of three GHW-tasks connected to the HP0 port in the FPGA-PS interface through an AXI SmartConnect has been realized. An Integrated Logic Analyzer (ILA) [34] module has been placed to monitor the AXI links that connect each GHW-task to the AXI SmartConnect and the AXI link that connects the AXI SmartConnect to the HP0 port. From the waveform track provided by the ILA, we measured the delays experienced by addresses and data while traversing the AXI SmartConnect (respectively, $d_{\text{Int}}^{\text{addr}}$ and $d_{\text{Int}}^{\text{data}}$, see Section 3.2) and the hold times t_{addr}, t_{data}, and t_{bresp} introduced in Section 3.2. The propagation delays (in clock cycles) observed on both hardware platforms are $d_{\text{Int}}^{\text{addr}} = 12$, $d_{\text{Int}}^{\text{data}} = 11$, and $d_{\text{Int}}^{\text{bresp}} = 9$, while the hold times t_{addr}, t_{data}, and t_{bresp} have been observed to be all constant and equal to one clock cycle. We note that these constant delays may be larger in different settings (not considered in this work) in which HW-tasks are not always ready to sample data or write responses, or when the FIFO queues of the interconnect or the FPGA-PS interface saturate. As mentioned in Section 3.3, the cumulative delays d_{PS}^{read} and d_{PS}^{write} in accessing the DRAM memory from the FPGA-PS interface depend on several aspects and on the masters that insist on the memory controller. In our experiment we did not used memory-intensive workload executed on the processors and we experimentally estimated these delays as $d_{PS}^{\text{read}} = 50$ clock cycles and $d_{PS}^{\text{write}} = 40$ clock cycles.

5.3 Model validation

This experiment aims at validating the assumptions made in Section 4 to characterize the interference that a HW-task may suffer from other HW-tasks. We distinguish between the case of a flat network and the one of a hierarchical network.

Interference in a flat network. The test setup used for these experiments comprises four GHW-tasks $\tau_0, \ldots, \tau_3$ directly connected to a single interconnect I, which is in turn directly connected to the HP0 port exported by the FPGA-PS interface (e.g., likewise as in Fig. 3). The GHW-tasks' activation and finishing times are measured using one of the custom timer IP deployed on the fabric. In this experiment, all the GHW-tasks are simultaneously activated (at the same clock cycle) by the Processing System using a single shared logic signal generated by an AXI GPIO module. With this experimental setup, all transactions issued by the GHW-tasks are subject to a single arbitration step performed by the AXI SmartConnect. The purpose of this evaluation is to experimentally evaluate the behavior of the AXI SmartConnect in the condition of contention. Furthermore, this experiment aims at experimentally measuring the maximum response time of a transaction in the worst-case scenario, i.e., when it loses an entire arbitration cycle, and comparing it with the proposed upper bound on the response time proposed in Section 4.5. To this end, all the GHW-tasks have been programmed to issue a single read (or write) request corresponding to a burst of sixteen 32-bit words. The experiment has been repeated for both read and write transactions. Figure 4 reports the maximum response time measured among all the GHW-tasks, compared with the upper-bound proposed for the flat architecture considered in this experiment.

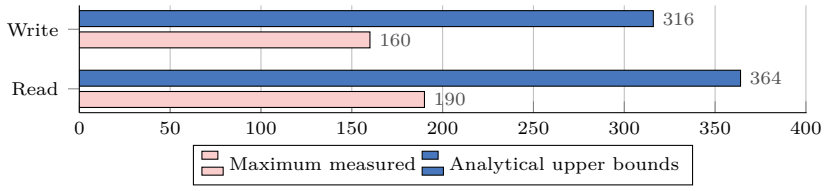


Figure 4 Maximum measured response times for read and write transactions compared with the upper bound proposed in Section 4 (in clock cycles).

The results reported in Figure 4 confirm that in the worst-case scenario stimulated here, i.e., when a HW-task loses an entire arbitration cycle, the measured response times can be safely bounded by the upper bound proposed in Section 4.

Interference in a hierarchical network. This set of experiments aims at validating the assumptions made in Section 4 to characterize the interference that a HW-task may suffer in a hierarchical network of Interconnects, due to the interfering HW-tasks in the system. The test setup used for this set of experiments comprises four GHW-tasks, $\tau_0, \ldots, \tau_3$, and three interconnects, I_0, I_1, I_2, organized as shown in Figure 5.

In this configuration, the transaction requests issued by τ_0 pass through a single step of arbitration occurring at interconnect I_0, while the requests issued by τ_1 traverse two arbitration steps occurring at I_1 and then I_0. Finally, the transaction requests issued by τ_2 and τ_3 pass through three arbitration steps at I_2, I_1, and I_0. The GHW-tasks are programmed and released as for the previous experiment. The first subset of experiments aims at validating the direct interference that a HW-task may suffer due to other HW-tasks connected to the same interconnect and the indirect interference coming from HW-tasks

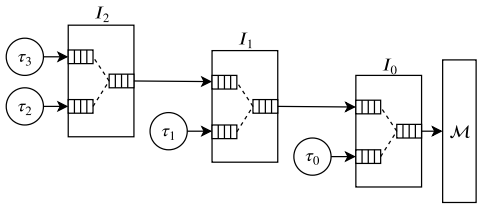


Figure 5 Reference architecture for the model validation in hierarchical network.

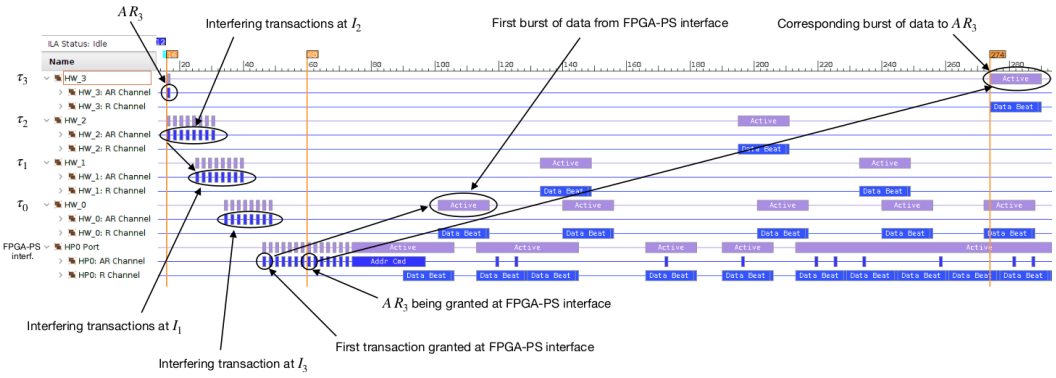


Figure 6 Sample waveform track from the Integrated Logic Analyzer on Zynq Ultrascale+.

connected to the lower-level Interconnects. In this experiment, τ_3 is the HW-task under investigation. τ_3 is programmed to issue a single request for transaction AR_3 (read or write) while the interfering tasks, τ_2, τ_1, τ_0, are programmed to issue eight consecutive interfering requests for transactions of the same type of AR_3. In order to stimulate contention at the interconnects, τ_1 is released with an offset equal to the interconnect propagation delay $d_{\text{Int}}^{\text{addr}}$, while τ_0 is released with a delay equal to $2d_{\text{Int}}^{\text{addr}}$ (offsets are with respect to the release time of AR_3 by τ_3). Each GHW-task-to-SmartConnect AXI link and the SmartConnect-to-HP0 AXI link are monitored by an ILA module deployed on the fabric and the HW-task's response times are measured using the timer module.

Figure 6 reports the ILA waveform track for read transactions acquired on the Zynq-7020 SoC using Vivado 2018.2. At time 15, all GHW-tasks are simultaneously released. Soon afterwards, at time 16, τ_3 issues its address request AR_3. At the same time, τ_2 starts issuing its first transaction request, say AR_2^0, causing a contention at the interconnect I_2. The arbitration round is won by τ_2. Hence I_2 first propagates AR_2^0 to I_1 and then AR_3. The interference at this level is compatible with the direct interference described in Lemma 4. After the propagation delay of the interconnect, I_2 issues the requests at the corresponding slave port of I_1. At the same instant, τ_1 releases its first transaction request, AR_1^0. Hence another contention happens, and the arbitration round at I_1 is won by τ_1. Consequently, I_1 forwards to I_0 the transaction requests in the following order: $AR_1^0, AR_2^0, AR_1^1, AR_3$, *hence according to round-robin arbitration as assumed in our model*. Note also that the amount of interfering requests on AR_3 at this level is compatible with the one found in Lemma 5 for indirect interference. When I_1 propagates this sequence of requests to I_0, τ_0 starts issuing its transaction requests, hence causing contention. τ_0 wins the arbitration round, hence the transaction requests are issued by I_0 to the HP0 port in the following order: $AR_0^0, AR_1^0, AR_0^1, AR_2^0, AR_0^3, AR_1^1, AR_0^3, AR_3$. Therefore, in the worst case, the request AR_3

issued by the GHW-task under investigation is interfered by seven requests coming from interfering GHW-tasks, *as considered by direct and indirect interference phenomena captured by our analysis in Section 4*. Since HP0 serves the incoming requests in order, τ_3 receives its data response only after all the interfering requests have been served with data. At time 274, the first word of data corresponding to AR_3 reaches τ_3 and at time 292 the transaction is completed. It is worth observing that Figure 6 also confirms that the AXI SmartConnect is compatible with the model introduced in Section 3.2 and that is characterized by $\phi_I = 1$.

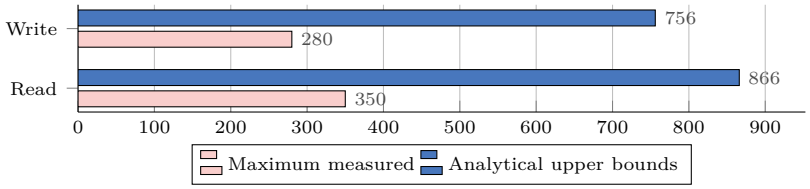


Figure 7 Maximum measured response times for read and write transactions compared with the upper bound proposed in Section 4 (in clock cycles).

Figure 7 compares the maximum measured response times for read/write transactions with the upper bounds computed by our analysis for the architecture under evaluation in this experiment. Also in this case, the results confirm that the delay incurred by transactions can be safely bounded.

5.4 Experiments with synthetic workload

This experimental study has been carried out to evaluate the analysis presented in Section 4 with synthetic workloads. We considered systems with N HW-tasks $(\tau_1, \ldots, \tau_N)$ connected over a binary tree of M interconnects $(I_1, \ldots, I_M)$. Task sets have been generated as follows. First, the period T_i and computation time C_i of each HW-task τ_i have been generated using the *fixedrandsum* algorithm [7] (T_i between $T_{min} = 10ms$ and $T_{max} = 100ms$, using log-normal distribution) by keeping the task set utilization equal to 1 as a reference value (note that the tasks' execution times are not relevant for bus contention in this context). Second, the number of transactions have been generated by first computing the maximum number of transactions that each HW-task τ_i can perform in isolation, as $N_i^{max} = (T_i - C_i)/\max(d^{NoCont,read}, d^{NoCont,write})$. Then, the total number of transactions $N_i^{R+W} = N_i^R + N_i^W$ is computed by multiplying N_i^{max} with a *transaction density factor* $\rho \in (0, 1]$ such that $N_i^{R+W} = \rho \cdot N_i^{max}$. Finally, the N_i^{R+W} transactions are split between reads and writes using a random uniformly-generated ratio in the range $\nu \in [0.4, 0.6]$, such that $N_i^R = \nu \cdot N_i^{R+W}$ and $N_i^W = (1 - \nu) \cdot N_i^{R+W}$. All HW-tasks have been configured with $\phi_i = 6$ (we found it being a typical value from experimental profiling of HAs in the Xilinx IP library) and $B_i = 16$, while all interconnects have $\phi_I = 1$. In order to test realistic configurations, it has been assumed that each interconnect cannot have more than 16 input ports (as it is the case for the Xilinx SmartConnect [35]).

The study considers 16 possible configurations generated by testing combinations of parameters N and M such that $N \in \{4, 8, 16, 24\}$ and $M \in \{1, 2, 4, 8\}$. Unuseful configuration, in which at least one interconnect hosts just a single HW-task, are discarded. This because, in such configurations, that Interconnect(s) would not perform any arbitration, adding only additional latency. For each valid configuration (N, M), 100 random values for ρ are uniformly chosen in the range $[0.1, 1.0)$. Then, for each value of ρ, $K = 50000$ synthetic task sets have been generated, each comprising N HW-tasks evenly distributed over M

interconnects (i.e., each interconnect is directly connected to at most $\lceil N/M \rceil$ HW-tasks). The HW-tasks have been distributed over the interconnect tree according to their slack times $S_i = T_i - C_i$, i.e., tasks with shorter slack times are placed closer to the root interconnect.

Figure 8 reports the results of the experimental study. Please note that, since each interconnect cannot be connected to more than 16 tasks, some configurations are topologically unfeasible. Hence, they are not considered and the corresponding data is not reported. The experimental results show that increasing the number of interconnects not only allows to connect a larger number of HW-tasks, but also can improve the system schedulability ratio by moving HW-tasks with larger slack time to interconnects at higher hierarchical levels, thus reducing their interference on more time-constrained HW-tasks (i.e., HW-tasks with shorter slack times). However, moving HW-tasks to a higher hierarchical level also increases the latency and the worst-case contention experienced by its transactions. The exploration of this trade off requires investigating on allocation strategies for HW-tasks, which is left as future work.

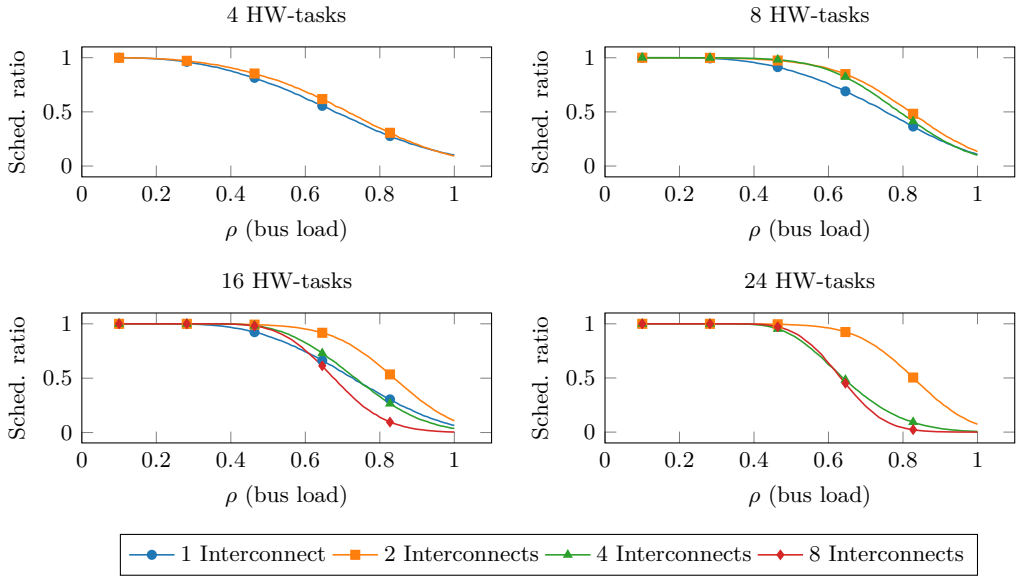


Figure 8 Experimental results with synthetic workload.

6 Related work

Considerable efforts have been spent in bounding and controlling response times in SoCs by addressing the problem from several prospectives. From an architectural point of view, several mechanisms and policies have been proposed, as support for HW prefetch and new arbitration policies [12, 15, 26]. Other works proposed to consider memory inteference in the context of task allocation [16, 17]. Significant work has been dedicated to the integration of memory interference in the schedulability analysis of both COTS and ad-hoc solutions, with a focus on specific elements in the memory tree, like the contribution of caches [9, 19], busses [6, 8], and the memory controller [4, 11]. Also, the explicit effect on the performance of control applications has been investigated [5]. Recently, FPGA-based SoCs have received particular interest, but allocating multiple HW-tasks inside the FPGA requires the use of a shared bus to access the off-chip memory. The AXI bus [2] is the de-facto standard

but has been designed considering flexibility and performance, not time predictability. In fact, the evaluation of bus interference is achieved with hardware monitors in charge of observing HW-tasks performance [29]. Moreover, the standard entrusts several design details to the single implementation and assumes all components behave accordingly [32]. Some mechanisms have been proposed to increase predictability. For example, Pagani et al. [22] proposed an approach to apply bandwidth reservation techniques to HW-tasks memory accesses, while Restuccia et al. designed a mechanism to guarantee fairness among HW-tasks transactions [25], a mechanism to prevent unbounded delays during bus transactions [23], and proposed a predictable, Hypervisor-level AXI interconnect for FPGA SoC [24]. However, these contributions only address single interconnects and are not concerned with a fine-grained timing analysis of bus transactions.

7 Conclusion and future work

This work focused on FPGA-based SoC and presented a fine-grained model for the AXI bus and AXI interconnects. An analysis has been proposed to bound the contention delays experienced by HW-tasks under hierarchical networks of interconnects that allow reaching the FPGA-PS interface (and hence shared memories connected to the Processing System). The model and the effectiveness of the analysis have been validated with experimental results on two modern FPGA SoC by Xilinx. Future work should focus on deriving a more accurate model and analysis of the AXI bus to capture pipelining effects, and on allocation strategies and bus network synthesis for a given set of HAs.

References

1 Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: a predictable SDRAM memory controller. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 251–256. ACM, 2007.

2 ARM. *AMBA AXI and ACE Protocol Specification*, 2011.

3 A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. Buttazzo. A framework for supporting real-time applications on dynamic reconfigurable fpgas. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 1–12, 2016.

4 D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo. A holistic memory contention analysis for parallel real-time tasks under partitioned scheduling. In *Proceedings of the 26th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2020)*, 2020.

5 W. Chang, D. Goswami, S. Chakraborty, L. Ju, C. J. Xue, and S. Andalam. Memory-aware embedded control systems design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(4):586–599, April 2017. doi:10.1109/TCAD.2016.2613933.

6 Sudipta Chattopadhyay, Lee Kee Chong, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. A unified WCET analysis framework for multicore platforms. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):124, 2014.

7 Paul Emberson, Roger Stafford, and Robert I Davis. Techniques for the synthesis of multiprocessor tasksets. In *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010.

8 Gabriel Fernandez, Javier Jalle, Jaume Abella, Eduardo Quiñones, Tullio Vardanega, and Francisco J. Cazorla. Increasing confidence on measurement-based contention bounds for real-time round-robin buses. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2744769.2744858.

9 Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Cache-aware scheduling and analysis for multicores. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 245–254. ACM, 2009.

10 Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. A Survey of FPGA-based Neural Network Inference Accelerators. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 12(1):2, 2019.

11 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):2323–2336, 2018.

12 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.

13 Intel. *Stratix 10 GX/SX Device Overview*, October 2017.

14 Intel FPGA. *Custom IP Development Using Avalon® and Arm AMBA AXI Interfaces*. OQSYS3000.

15 J. Jalle, L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla. Bus designs for time-probabilistic multicore processors. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, March 2014. doi:10.7873/DATE.2014.063.

16 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)*, April 2014.

17 Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding and reducing memory interference in COTS-based multi-core systems. *Real-Time Systems*, 52(3):356–395, May 2016.

18 Jörg Henkel Lars Bauer, Marvin Damschen. Runtime-reconfigurable architectures for WCET guarantees and mixed criticality. In *Special session at ESWEEK 2019: Analyses and Architectures for Mixed-Critical Systems: Industry Trends and Research Perspective*. ACM, 2019.

19 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):05–1–05:48, 2016. doi:10.4230/LITES-v003-i001-a005.

20 Geoffrey Nelissen and Alessandro Biondi. The SRP Resource Sharing Protocol for Self-Suspending Tasks. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 361–372. IEEE, 2018.

21 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.

22 Marco Pagani, Enrico Rossi, Alessandro Biondi, Mauro Marinoni, Giuseppe Lipari, and Giorgio Buttazzo. A Bandwidth Reservation Mechanism for AXI-Based Hardware Accelerators on FPGAs. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:24, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

23 Francesco Restuccia, Alessandro Biondi, Mauro Marinoni, and Giorgio Buttazzo. Safely Preventing Unbounded Delays During Bus Transactions in FPGA-based SoC. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020.

24 Francesco Restuccia, Alessandro Biondi, Mauro Marinoni, Giorgiomaria Cicero, and Giorgio Buttazzo. AXI HyperConnect: A Predictable, Hypervisor-level AXI Interconnect for Hardware Accelerators in FPGA SoC. In *Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC 2020)*, 2020.

25 Francesco Restuccia, Marco Pagani, Alessandro Biondi, Mauro Marinoni, and Giorgio Buttazzo. Is Your Bus Arbiter Really Fair? Restoring Fairness in AXI Interconnects for FPGA SoCs. *ACM Trans. Embedded Computing Systems*, 18(5s):51:1–51:22, October 2019.

26 M. Slijepcevic, C. Hernandez, J. Abella, and F. J. Cazorla. Design and implementation of a fair credit-based bandwidth sharing scheme for buses. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 926–929, March 2017. doi:10.23919/DATE.2017.7927122.

27 Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 65–74. ACM, 2017.

28 Xilinx. *Zynq-7000 All Programmable SoC - Reference Manual*, September 2016. UG585.

29 Xilinx. *AXI Performance Monitor v5.0*, 2017. PG037.

30 Xilinx. *Vivado Design Suite: AXI Reference Guide*, July 2017. UG1037.

31 Xilinx. *Zynq UltraScale+ Device - Reference Manual*, December 2017. UG1085.

32 Xilinx. *AXI Interconnect, LogiCORE IP Product Guide*, 2018. PG059.

33 Xilinx Inc. *The CHaiDNN official github website*. https://github.com/Xilinx/chaidnn.

34 Xilinx Inc. *Integrated Logic Analyzer, LogiCORE IP Product Guide*, 2016. PG172.

35 Xilinx Inc. *SmartConnect, LogiCORE IP Product Guide*, 2018. PG247.

On How to Identify Cache Coherence: Case of the NXP QorIQ T4240

Nathanaël Sensfelder
ONERA, Toulouse, France

Julien Brunel
ONERA, Toulouse, France

Claire Pagetti
ONERA, Toulouse, France

Abstract

Architectures used in safety critical systems have to pass certain certification standards, which require sufficient proof that they will behave as expected. Multi-core processors make this challenging by featuring complex interactions between the tasks they run. A lot of these interactions are made without explicit instructions from the program designers. Furthermore, they can have strong negative impacts on performance (and potentially affect correctness). One important such source of interactions is cache coherence, which speeds up operations in most cases, but can also lead to unexpected variations in execution time if not fully understood. Architecture documentations often lack details on the implementation of cache coherence. We thus propose a strategy to ascertain that the platform does indeed implement the cache coherence protocol its user believes it to. We also apply this strategy to the NXP QorIQ T4240, resulting in the identification of a protocol (MESIF) other than the one this architecture's documentation led us to believe it was using (MESI).

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

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.13

1 Introduction

The ever increasing complexity of aircraft and the market's depreciation of single-core processors are motivating the introduction of multi-core processors in aeronautical systems. While the performance gains offered by a switch to these more recent architectures are enticing, this process is impeded by their seemingly unpredictable nature [34], which is inherently incompatible with safety critical environments and aeronautical certification [9]. Still, a number of works are focusing on determining the means required for aircraft manufacturers to fulfill certification expectations despite the complex internal behaviors of multi-core processors COTS (Commercial Off-The Shelves) [1, 5, 12, 26, 28].

1.1 Cache Coherence – Case of the NXP T4240

Part of this unpredictability can be imputed to the mechanisms that let caches coordinate with one another in order to maintain data coherence without explicit program instructions. There are multiple competing strategies that can be employed to achieve *cache coherence*, and, while the general ideas behind them are known, the details of their implementation tend to be absent from architecture documentations, leaving programmers with the task of finding possibly problematic corner cases and unexpected behaviors.

In this paper, we focus on the NXP QorIQ T4240 [14], a PowerPC architecture featuring twelve e6500 cores, each of which is capable of running two simultaneous threads. The cores are equally distributed among three clusters, with one 2MB L2 cache per cluster. These

three L2 caches coordinate and access memory through a complex interconnect called the CoreNet Coherency Fabric. According to their processor's documentation, [13], these clusters implement the MESI cache coherence protocol. More details can be seen in Figure 1, which displays all the cores, caches, and memory controllers present on that architecture.

To be allowed to embed this architecture in an aeronautical system, the designer must be in control of any *transaction* occurring on the platform, that is, any low level behaviors caused by either explicit requests made by a program or by implicit mechanisms of the platform. This also holds true for cache coherence: it is up to the designer to quantify and control the effects on the application software of any transaction generated by this mechanism.

1.2 Formal Specification and its Validation

Having to keep implicit mechanisms under control is not an easy task for designers. This is especially true in the case of cache coherence, whose impact is difficult to evaluate even when its rules are made known to the designer.

In this paper, we present our analysis of the NXP QorIQ T4240 cache coherence transactions. This first required us to determine the protocol implemented in the architecture. According to its documentation, the protocol is supposed to be MESI [29] (*Modified*, *Exclusive*, *Shared*, *Invalid*). To guarantee the proper coverage of all that is involved, we argue for a formal definition of the cache coherence protocol to be made by the designer, based on their current understanding. Such formal definitions do not leave room for any ambiguities. Thus, we have looked for preexisting models of MESI protocol for split-based transaction buses. As it happens, we found none, making our first contribution (Section 3) a formal definition for a split-transaction bus MESI cache coherence protocol, which also corresponds to what we believed the NXP QorIQ T4240 to be using.

This formal MESI protocol definition describes all the transactions it is supposed to be performing. Thus, through the application of our proposed strategy (Section 4) we are able to make use of appropriate stress testing to observe the platform's behavior and compare with what we expected, in effect validating that the architecture does indeed implement the protocol we believe it to. While we developed this strategy around the T4240 and its limited means of observation, we tried to keep our strategy generic enough that it could be soundly used for other targets.

When we applied the strategy to validate the NXP QorIQ T4240, it became apparent the protocol was not actually MESI. Indeed, thanks to the validation strategy, we observed that there were five stable states instead of four and that one of them behaves in a way that led us to recognize a MESIF protocol [17]. We thus had to formally define a split-transaction bus MESIF protocol (Section 6), as we could not find any preexisting definition for it either. We then applied the validation strategy with this new protocol as the starting point, and this time we only found slight implementation choice differences between the supposed and the observed behaviors (Section 7).

In the sequel, we start with a reminder of hardware components and their contribution to the cache coherence. We then detail the contributions described above. We compare our approach with the related works before concluding the paper.

2 Cache Coherence

This section provides a reminder of the terminology and of the components involved in the description. Figure 2 provides a visual summary of how all these components interact.

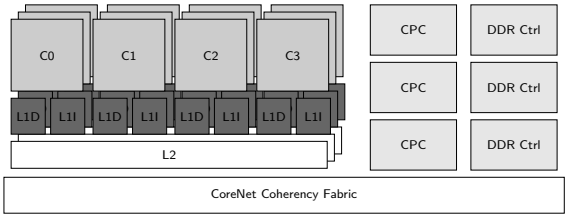


Figure 1 Computation & memory parts of the T4240.

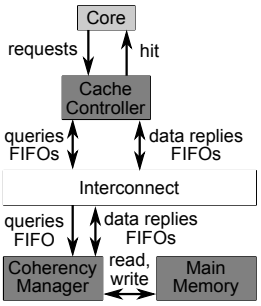


Figure 2 Components involved in coherence.

2.1 The Programs

To keep things simple, we only consider the memory related instructions of programs. Thus, programs are reduced to sequences of `load`, `store`, and `evict` instructions, each being applied to a single given address. Programs do not take into consideration the possibility of either instruction jumping or branching. Addresses are tantamount to *memory element*s (aligned blocks of memory with the size of a cache line), preventing any possible aliasing. Thus, all considered components, including programs, operate on the same memory unit.

2.2 The Caches

Cache controllers keep copies of memory elements to perform core requests. These copies are acquired through queries on the interconnect. Read-only copies are queried using `GetS`, read-and-write copies through `GetM`, and the eviction of a copy can be indicated through `PutM`. A cache controller may reply to the query of another, providing them with data. They are also able to send data to the coherence manager. Each copy of a memory element held in a cache is attributed a state and, optionally, the identifier of a cache controller.

2.3 The Interconnect

The *interconnect* merely arbitrates the order in which queries are broadcasted. Cache controllers do not directly access the interconnect. Instead, interactions between the cache controllers and the interconnect are all done through FIFO queues. There are four in total for each cache controller: incoming and outgoing queries; and incoming and outgoing data messages. The interconnect follows its access policy when choosing which cache controller's outgoing query queue to poll from next, then enqueues that query to every cache controller's incoming query queue (including that of the one from which the query was taken). Thus, all cache controllers and the coherence manager receive all queries and do so in the same order.

We consider interconnects that support split-transactions, meaning that queries and their replies do not block each other, allowing new queries to be sent before the previous ones receive their replies.

2.4 Coherence Manager and Main Memory

Cache controllers do not directly send messages to the system's main memory. Instead, messages meant for the main memory are directed to the *coherence manager*. The coherence manager sees all passing queries. It keeps track of which memory elements are being held by the cache controllers, and has a general idea of their current permissions. In particular, the

coherence manager may consider a cache controller to be the *owner* of a memory element, meaning that this cache controller is tasked with the propagation of the memory element's current value. This lets the coherence manager determine when a query warrants a reply from the main memory.

2.5 Terminology

The term *request* covers all types of communications between a core and its cache controller: *requests* = {`load`, `store`, `evict`}. The term *message* covers both the demands made by cache controllers, and the replies that fullfill them. In other words, *messages* are all communications that pass through the interconnect: *messages* = *queries*∪*data replies*, where *queries* = {`GetM`, `GetS`, `PutM`}, and *data replies* = {`data`, `data-e`, `no-data`}. Note that the actual elements found in *queries* and *data replies* depend on the specified protocol. The values given here being for the procotol described in the very next section.

3 Formal Description of the MESI Protocol

Our first contribution is the formal definition of a split-transaction MESI protocol that relies on a coherence manager. While the general idea behind MESI is available in many existing works, we did not find any that lists all the possible transient states that can be found in a real implementation (i.e. states other than Modified, Exclusive, Shared, and Invalid). These omissions tend to make the protocol much simpler to understand, but they leave ambiguities in the behavior of the protocol. Our description is a conjecture based on [31], which presents a complete definition of the MESI protocol, but that is limited to architectures featuring an atomic bus. Atomic buses only allow a single transaction (query and reply) to occur at any given time, which greatly narrows the number of transient states the system can find itself in.

3.1 Protocol Specification

MESI is based on the MSI protocol, so named because it features three stable states: *Modified*, which indicates read-and-write permissions of a memory element; *Shared*, for read-only permissions; and *Invalid*, the default one, indicating an absence of permissions. Introduced in [29], the MESI protocol adds a fourth stable state, *Exclusive*, which indicates that not only does the cache controller have read-only permissions, but also that no other cache currently holds any permission to access the memory element. This allows the cache controller to upgrade to read-and-write permissions without having to perform a costly communication. Just as it is used to keep track of whether a cache holds a read-and-write copy of a memory element in the MSI protocol, this definition of the MESI protocol uses the coherence manager to detect when a cache can be said to be the sole owner of a memory element.

This version of the MESI protocol uses three types of data replies: `data`, `data-e`, and `no-data`. `data` indicates that the value associated with the memory element is sent. By sending a `no-data` reply, cache controllers can indicate to the coherence manager that the memory element has been discarded (its value is not part of the reply). The coherence manager can send `data-e` replies, which are equivalent to `data`, with the added information that the recipient is its sole owner.

Our description of the MESI protocol can be seen in Table 1. It is split in two tables, one defining the cache controllers' behavior, the other the coherence manager's. In effect, these tables indicate a sequence of actions to be performed when faced with an incoming event (be it a request or a message).

Table 1 Description of the MESI protocol.

Cache Controller									
State	Core Request			Interconnect Access	Data Reply		Received Queries		
	load	store	evict		data	data-e	GetS	GetM	PutM
I	GetS?, IS^{BD}	GetM?, IM^{BD}	hit				-	-	-
IS^{BD}	stall	stall	stall	$IEoS^{D}$	IS^{B}	IE^{B}	-	-	-
IS^{B}	stall	stall	stall	S			-	-	
IS^{D}	stall	stall	stall		r←∅, S	r!data, m!no-data, r←∅, S	-	$IS^{D}I$	
$IEoS^{D}$	stall	stall	stall		S	E	r←s, IS^{D}	r←s, $IS^{D}I$	
$IS^{D}I$	stall	stall	stall		load hit, r←∅, I	load hit, r←∅, r!data, m!no-data, 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		store hit, r!data, r←∅, I		-	-	
$IM^{D}S$	stall	stall	stall		store hit, r!data, m!data, r←∅, S		-	$IM^{D}SI$	
$IM^{D}SI$	stall	stall	stall		store hit, r!data, m!data, r←∅, I		-	-	
S	hit	GetM?, SM^{BD}	hit, 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		store hit, M		r←s, $SM^{D}S$	r←s, $SM^{D}I$	
$SM^{D}I$	hit	stall	stall		store hit, r!data, r←∅, I		-	-	
$SM^{D}S$	hit	stall	stall		store hit, r!data, m!data, r←∅, S		-	$SM^{D}SI$	
$SM^{D}SI$	hit	stall	stall		store hit, r!data, m!data, r←∅, 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			-	-	-
E	hit	hit, M	PutM?, EI^{B}				m!no-data, s!data, S	s!data, I	
IE^{B}	stall	stall	stall	E			-	-	-
EI^{B}	hit	stall	stall	m!no-data, I			m!no-data, s!data, II^{B}	s!data, II^{B}	

Coherence Manager						
State	Received Queries				Data Reply	
	GetS	GetM	PutM (Owner)	PutM (Other)	data	no-data
I	read, s!data-e, r←s, M	s!data, r←s, M		-		
M	r←∅, S^{D}	r←s	r←∅, I^{D}	-	write, IoS^{B}	IoS^{B}
I^{D}	stall	stall	stall	-	write, resume, I	resume, I
S^{D}	stall	stall	stall	-	write, resume, S	resume, S
IoS^{B}	r←∅, S	r←s, M	r←∅, I	-		
S	read, s!data	s!data, r←s, M		-		

In the cache controller's table, columns correspond to the following: *state* refers to the state attributed to the local copy of the memory element by the cache controller. The three *Core request* columns indicate the actions that are performed when receiving a request from the core. *Interconnect access* specifies actions for when the cache controller reads one of its own queries. The *data reply* columns are for when the cache controller receives one of the types of data replies. Lastly, the *received queries* columns are for the reception of queries originating from other cache controllers. The table defining the coherence manager follows the same principles, but does not have columns for core requests, as it cannot receive them, nor for access to the interconnect, as it does not emit queries.

Let us now expand on the semantics of the actions found in these tables. Cache controllers may send queries on the bus (e.g. sending a `GetS` query is noted as `GetS`?). They can also change the state they attribute to a memory element (e.g. moving to the I state, which is noted `I`). If a request coming from their core can be fulfilled without further actions, the table indicates it with `hit`. A similar notation is used to indicate that the oldest request of a given type has just been completed (e.g. `load hit`). As a reaction to an incoming query, cache controllers can mark their copies of memory elements as being associated with the cache controller that sent the query (noted `r`←`s`). This can later be used to send a data message to that cache controller (e.g. `r`!`data`). Data can also be sent as a reply to an incoming query (e.g. `s`!`data`), or to the coherence manager (e.g. `m`!`no-data`). The `stall` action marks that the cache controller is unable to handle the incoming request at the moment. This request is put into a waiting queue until the memory element changes state, at which point it is re-evaluated.

The coherence manager follows a similar syntax, with the exception of the `stall` action, which now blocks *any* incoming query until the next `resume` action (data messages are not blocked, however). The other additions are the `write` and `read` actions, which respectively indicate that the memory controller either writes the received value or reads the current one.

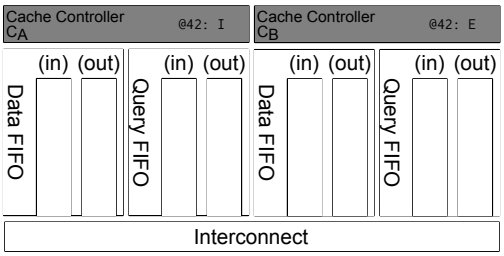
3.2 Examples of Behaviors

Here are some examples of remarkable behaviors exhibited by this definition of the MESI protocol.

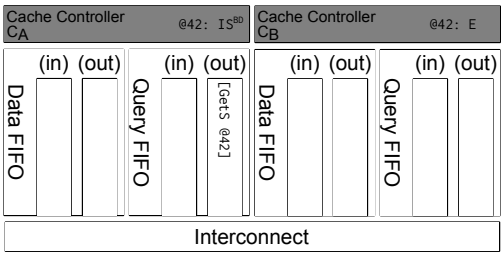
▶ **Example 1 (Reaching `S`).** This example is meant to showcase how exchanges between cache controllers are assumed to take place. To keep things simple, we only consider two cores and a single memory element (whose address is 42). This example is illustrated as a sequence in Figure 3.

▶ **Example 2 (Reaching `E`).** To hold a memory element in the `E` state, a cache must be the only one to have a copy of that memory element. The caches rely on the coherence manager to know when it is the case. The coherence manager uses its `I` state to mark memory elements that are sure to not be in any caches. Thus, if no cache controllers hold the memory element and the coherence manager is in `I`, whenever a core `loads` the data it becomes `E` in its cache. The behavior is similar to Figure 3 except that the main memory will provide the data.

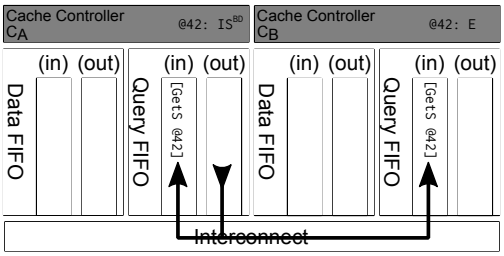
It is important to notice that it is not easy for the coherence manager to detect whether a cache controller is the sole owner. Indeed, the coherence manager is not always able to know that all caches have evicted their copy of a memory element: in Table 1, the cache controller's table indicates that an eviction from `S` does not lead to any message. The only way for the coherence manager to return to the `I` state is for a cache to evict its copy of a memory element in either the `E` or `M` state without another cache asking for a copy.



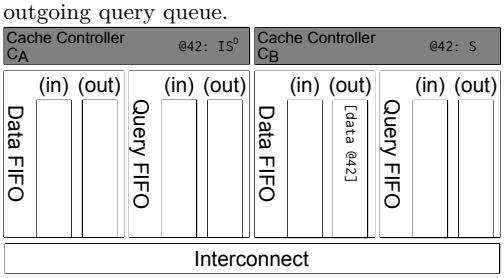
(a) In the initial phase, one cache controller, C_A, holds that memory element in the `I` state, while the other, C_B, holds it in the `E` state.



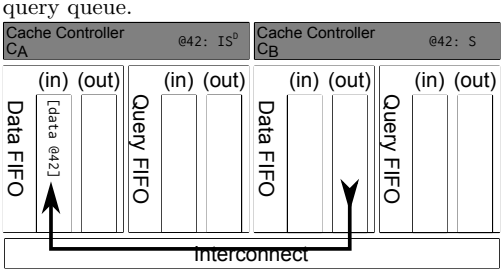
(b) Next, we consider that C_A's core issued a `load` instruction on 42. This leads C_A to move to the IS^{BD} state, and to issue a `GetS` query to its outgoing query queue.



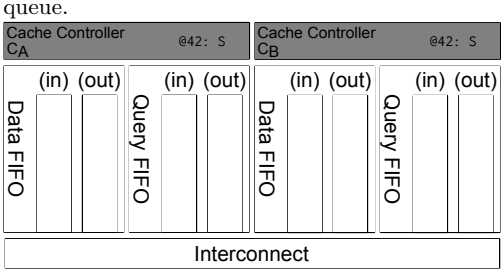
(c) The interconnect broadcasts outgoing queries from caches to all the incoming query queues. As C_A is the only one with an outgoing query, the `GetS` is added to both its own and C_B's incoming query queue.



(d) Consuming the message in their incoming query queues, both C_A and C_B change state, moving to IS^{D} and `S` respectively. In addition, C_B adds its reply for the query to its outgoing data queue.



(e) Data messages are not broadcasted, but instead only added to the incoming data queue of a targeted cache controller (or the coherence manager's). Thus, the reply is moved from C_B's outgoing reply queue to C_A's incoming one.



(f) Finally, C_A consumes the message in its incoming data queue, changing its state to `S` and fulfilling its core's request.

Figure 3 Illustrations for **Reaching `S`**.

▶ **Example 3 (Sharing from `E`).** From the coherence manager's point of view, there is no difference between a cache controller owning a memory element in the `E` state and one in the `M` state. Thus, if there is a cache owning a copy of a memory element in the `E` state, the coherence manager will assume that this cache may have modified the value and that the main memory no longer holds the correct value. As a result, the cache holding the Exclusive copy of the memory element will transfer it to any other cache that asks for it. If this is caused by another cache demanding a read-only copy (`GetS`), the coherence manager will expect an update on the value of the memory element. This update can come in two forms: either the cache that exclusively held the memory element made a modification (in which case it would have moved to the Modified state) and sends a `data` message, or it has not and it sends a `no-data` message.

3.3 System Behavior

The cache coherence protocol is defined for a single address and a single cache controller. However, what we are interested in is a multi-core architecture executing a program. Thus, we need to model the behavior of the overall platform to identify and quantify the transactions generated by the cache coherence. To do so, we use an automaton where each state corresponds to the system state and transitions between states are events produced by one or several components (core, cache controller, interconnect, coherence manager or memory). In this section we formally define such automata.

▶ **Definition 4** (Memory Element State). *Let $\mathcal{S}_s$ (resp. $\mathcal{T}_s$) denote the set of stable (resp. transient) states. From the point of view of a cache controller or a coherence manager, a memory element m can be in any valid stable or transient states, i.e., $m \in \mathcal{S}_s \cup \mathcal{T}_s$. We denote by $\mathcal{G}_s$ the set of states $\mathcal{G}_s = \mathcal{S}_s \cup \mathcal{T}_s$ of a cache controller and $\mathcal{G}_{CM}$ those of the coherence manager.*

▶ **Example 5.** In the MESI protocol defined in Table 1, the states of the cache controllers are $\mathcal{S}_s = \{M, E, S, I\}$ and $\mathcal{T}_s = \{IS^{\mathrm{BD}}, IS^{\mathrm{D}}, \ldots\}$. The states of the coherence manager are $\mathcal{S}_s = \{M, S, I\}$ and $\mathcal{T}_s = \{I^{\mathrm{D}}, S^{\mathrm{D}}, IoS^{\mathrm{B}}\}$.

▶ **Definition 6** (Cache Controller State). *Let $\mathtt{Addr}$ denote the set of all memory element addresses. We define the state of a cache controller CC (resp. of a coherence manager CM) as the function $s_{CC} : \mathtt{Addr} \to \mathcal{G}_s$ (resp. $s_{CM} : \mathtt{Addr} \to \mathcal{G}_{CM}$).*

▶ **Definition 7** (System State). *Let us consider an architecture $\langle CC_1, \ldots, CC_n, CM\rangle$ composed of n cache controllers CC_i and a coherence manager CM. The global state of the architecture consists of the states of all memory elements in all cache controllers and in the coherence manager. Let $\mathtt{Addr}$ be the set of all memory element addresses, the global state $s : \mathtt{Addr} \to \mathcal{G}_s^n \times \mathcal{G}_{CM}$ is defined as $\forall m \in \mathtt{Addr}, s(m) = \langle s_{CC_1}(m), \ldots, s_{CC_n}(m), s_{CM}(m)\rangle$ where s_{CC_i} is the state of the cache controller CC_i as defined in Definition 6. For the sake of simplicity and without loss of the generality, in the sequel, we will only focus on a given address m and define the state of a cache controller as an element in $\mathcal{G}_s$ and of the system as a tuple in $\mathcal{G}_s^n \times \mathcal{G}_{CM}$.*

▶ **Definition 8** (Valid State). *Not all combinations of states are valid, e.g. two cache controllers cannot be in $\mathtt{M}$ for the same address at the same time. We note $\mathcal{V} \subseteq \mathcal{G}_s^n \times \mathcal{G}_{CM}$ the set of valid system states.*

▶ **Definition 9** (Event). *We distinguish between explicit (or controllable) events $E_E = requests = \{\mathtt{load}, \mathtt{store}, \mathtt{evict}\}$, which are made by the user, and the implicit (or uncontrollable) events $E_I = messages \cup \{bus\}$ (where bus corresponds to the cache seeing one of its own query being broadcasted on the interconnect), which are made by the architecture. Thus, on a given cache controller, the possible events are $E_E \cup E_I \cup \{-\}$ where $-$ represents the special event where nothing happens.*

The set of events over the system is denoted by $E \subseteq (E_E \cup E_I \cup \{-\})^n$ (not all combinations of events are possible).

The system event $\langle -, \ldots, -, e, -, \ldots, -\rangle$ consisting of one event e in the cache controller of id i, and nothing in all the other cache controllers, is simply denoted by $\langle e, i\rangle$.

▶ **Definition 10** (Automaton of the system). *The behavior of the system is defined by the automaton $\langle \mathcal{V}, E, s_{init}, Tr\rangle$ where $s_{init} = \langle I, \ldots, I\rangle$ is the initial state and $Tr : \mathcal{V} \times E \to \mathcal{V}$ is the transition function.*

▶ **Example 11.** Using the MESI protocol and 2 cache controllers, we have for instance $\mathrm{Tr}(\langle I, I, I\rangle, \langle load, -\rangle) = \langle IS^{\mathrm{BD}}, I, I\rangle$. As the event is a single component event, we could also use the notation mentioned above $\langle load, 1\rangle = \langle load, -\rangle$ meaning that the cache controller with id 1 does a *load*, whereas all the other do nothing. If we detail all the implicit events leading to the next stable states, we have: $\mathrm{Tr}(\langle IS^{\mathrm{BD}}, I, I\rangle, \langle \mathrm{bus}, GetS, GetS\rangle) = \langle IEoS^{\mathrm{D}}, I, M\rangle$ (the interconnect broadcasts the *GetS*); $\mathrm{Tr}(\langle IEoS^{\mathrm{D}}, I, M\rangle, \langle \text{data-e}, 1\rangle) = \langle E, I, M\rangle$ (the cache coherence triggers the memory which provides the requested data).

▶ **Example 12** (Simultaneous requests)**.** Still using the MESI protocol and 2 cache controllers, there may be several simultaneous requests, e.g. $\langle load, store\rangle$. In such situations, because of the internal dynamics of the interconnect and memory (Round Robin access, delays ...) all combinations of interleaving are envisaged. For instance $\mathrm{Tr}(\langle I, I, I\rangle, \langle load, store\rangle) = \langle IS^{\mathrm{BD}}, IM^{\mathrm{BD}}, I\rangle$ (all local requests are handled). Then, among the possible next steps are both $\mathrm{Tr}(\langle IS^{\mathrm{BD}}, IM^{\mathrm{BD}}, I\rangle, \langle bus, GetS\rangle) = \langle IEoS^{\mathrm{D}}, IM^{\mathrm{BD}}, M\rangle$ (the interconnect chooses the first core first) or $\mathrm{Tr}(\langle IS^{\mathrm{BD}}, IM^{\mathrm{BD}}, I\rangle, \langle GetM, bus\rangle) = \langle IS^{\mathrm{BD}}, IM^{\mathrm{D}}, M\rangle$ (the interconnect chooses the second core first). Thus, several paths leave from $\mathrm{Tr}(\langle I, I, I\rangle, \langle load, store\rangle)$ and they may ultimately lead to separate stable states: $\langle I, M, M\rangle$ if the data reply has reaches core 1 first, or $\langle S, S, S\rangle$ if the data reply reaches core 2 first.

▶ **Definition 13** (Path)**.** *A path in a system automaton corresponds to a succession of transitions, from one state to another, that has been triggered by a controllable event and is followed by a series of adequate implicit events. A path from s to s' triggered by e is denoted by $p : s \rightsquigarrow^{e} s'$.*

▶ **Example 14.** The successive transitions that have been described in Example 11 define the path $\langle I, I, I\rangle \rightsquigarrow^{\langle load,1\rangle} \langle E, I, M\rangle$.

▶ **Definition 15.** *From the transition function Tr, we define the observable transition function $Tr_i^\star$, in the case of single controllable events, as follows: $\forall c \in \mathcal{V}, e \in E_E$ $Tr_i^\star(c, \langle e, i\rangle) = c'$ such that $c \rightsquigarrow^{\langle e,i\rangle} c'$ and no implicit event may be induced by $\langle e, i\rangle$ upon reaching c'.*

For simultaneous explicit events, the observable transition function is defined by composing the observable transition function for each explicit event, taken as a single event. Notice that it is an expected property of the protocol, which we do not study here, that all possible orderings provide the same system state.

▶ **Example 16.** Considering a single event: $\mathrm{Tr}_i^\star(\langle I, I, I\rangle, \langle load, 1\rangle) = \langle E, I, M\rangle$.

Let us now consider multiple events: $\mathrm{Tr}_i^\star(\langle I, I, I\rangle, \langle load, store\rangle) = \{\langle S, S, S\rangle, \langle I, M, M\rangle\}$.

▶ **Definition 17** (Number of Events)**.** *We define the function $NbEvent : \mathcal{P}ath \times 1..n \to \mathbb{N}^3$ (with n the number of cores in the architecture) which associates to each path and core id, the number of accesses to the bus, the number of received queries, and the number of received data replies for the cache controller identified by this id.*

▶ **Example 18.** Let us consider the path $p : \langle I,I,I\rangle \rightsquigarrow^{\langle load,1\rangle} \langle E,I,M\rangle$. Then $NbEvents(p,1) = \langle 1, 1, 1\rangle$ because core 1 has accessed to the bus once, received its own *GetS* and the data reply to its request. $NbEvents(p, 2) = \langle 0, 1, 0\rangle$ because core 2 has simply received the *GetS* generated by core 1.

4 Validation Strategy

This section presents our proposed strategy to assert that a given architecture does indeed implement a given previously defined cache coherence protocol. We have fully defined the

system behavior in Section 3.3 and ideally we would recognize all the automata on the architecture. Unfortunately, we cannot simply observe the states and events as we previously defined them. Instead, we observe flags and performance counters, to which we need to link the notion of states and events. Even worse, our observations are only partial, with some information missing. Thus, in addition to linking the observations to the automaton, we also have to infer the missing elements. We illustrate our ideas on the NXP QorIQ T4240 platform, however, the reasoning could be leveraged for other types of architecture.

Observable States

▶ **Property 1** (T4240 Observable Flags). *We can observe flags with CodeWarrior, the official debugging suite for this architecture. While a lot of information is available, we consider the relevant cache line flags to be: Dirty, Valid, Share, Exclusive and LastReader. Those flags take Boolean value and only provide information on stable states. Indeed, no combination of flags correspond to any transient state. Instead, their value changes upon entering the next stable state following the execution of a request (load, store, or evict) or because of an external query.*

▶ **Definition 19** (Observable Cache Controller State). *Let us consider an architecture with p Boolean flags, an observable state o for a cache controller is a combination of values of the flags $o = \langle f_1, \ldots, f_p \rangle$. Let $\mathcal{R}$ denote the set of cache controller states that can be really observed on a given architecture.*

▶ **Issue 1** (Matching Observable Cache Controller States and Stable States). *For validating that an architecture indeed implements a cache protocol, we need to associate each observed state with a protocol state. More precisely, we need to identify a function* `Decode` $: \mathcal{R} \to \mathcal{G}_s$ *such that* `Decode` *is surjective:* $\forall f \in \mathcal{G}_s,\ \exists r \in \mathcal{R},\ \texttt{Decode}(r) = f$.

Indeed, while having multiple observed states corresponding to the same state is perfectly acceptable at this point (the different states may end up being identical from the cache coherence's point of view), the reverse is not true: if an observed cache coherence state is attributed to multiple formally defined state, the analysis considers that the protocols do not match. This can be caused by missing information (unable to observe the information that would split the observed state into multiple ones). This is the reason why `Decode` has to be surjective.

▶ **Example 20.** For the T4240, the observable state $\langle$*Dirty*=false, *Valid*=false, *Share*=false, *Exclusive*=false,*LastReader*=false$\rangle$ is the initial observable state and corresponds to the I stable state.

The tools at our disposition do not expose anything related to the coherence manager.

▶ **Property 2** (No Observation Available from the Coherence Manager). *We do not have any possibility of observing the coherence manager directly.*

▶ **Definition 21** (Observable System State). *Let us consider an architecture $\langle CC_1, \ldots, CC_n, CM \rangle$ composed of n cache controllers CC_i and a coherence manager CM. The observable system states are the observable states of each CC_i and the CM.*

▶ **Example 22.** On the T4240, the observable system states are the observable states of each cache controller only. Thus, to match the observable state and the real state we have to infer the non observable elements.

▶ **Issue 2** (Matching Observable States and System States). *To validate that an architecture indeed implements a given cache protocol, we need to identify a function* $\texttt{Decode} : \mathcal{R}^n \to \mathcal{V}$ *associating a tuple of observable states with a system state, which is directly defined from the function* `Decode` *of Issue 1 and that is also surjective:* $\forall f \in \mathcal{V},\ \exists r \in \mathcal{R}^n,\ \texttt{Decode}(r) = f$.

Controllable Events and Reachable Observable States

▶ **Property 3** (T4240 controllable events). *On each core, we can execute programs. Thus, to induce implicit cache coherence traffic, we can only trigger some request (load, store or evict) and observe the reached observable states. We have defined a series of benchmarks that can either run a single request on a core or multiple requests on several cores.* $\texttt{Reach}(C, \langle \text{instr}, k\rangle)$ *denotes the observable state after executing the single request* instr *on the core* k *from the observable state* C*. In addition,* $\texttt{Reach}_m(C, \langle \text{instr}_1, \ldots, \text{instr}_n\rangle)$ *denotes the observable states after executing the simultaneous requests* $\text{instr}_i \in \{load, store, evict, -\}$ *on each core from the observable state* C.

▶ **Definition 23** (Reachable System States). *From an architecture in an initial state in which no memory elements are stored in the caches, we can explore the reachable system states by executing benchmarks that trigger requests.*

▶ **Definition 24** (Step 1: Reachability Analysis). *Starting from the initial situation where all cache controllers consider the memory element to be invalid, we compute the reachable observable system states by observing the effect of a single core instruction and the associated transition relation* `Reach`*. The idea is to run a benchmark and observe the reached state. If this state has not been visited, it is added to* $\mathcal{R}^n$*, otherwise it is not. This is a basic reachability algorithm.*

```
R^n ← {init}
Candidates ← {⟨init⟩}
while (Candidates ≠ ∅):
    C ∈ Candidates;
    Candidates ← Candidates/C;
    foreach k ≤ n
        foreach instr ∈ {load, store, evict}
            ObservedState ← benchmark(C, ⟨instr, k⟩)
            Reach (C, ⟨instr, k⟩) ← ObservedState
            if ObservedState ∉ R^n
                R^n ← R^n ∪ {ObservedState}
                Candidates ← Candidates ∪{ ObservedState}
                Events[C, ObservedState] ← PerformanceCounters
```

▶ **Issue 3.** *To be valid, the matching between observable states and system states must be consistent with the transitions of both protocols. That is, the function* `Decode` *has to be a* simulation relation*:* $\forall o \in \mathcal{R}^n,\ \forall c \leq n,\ \forall i \in$ requests, $\texttt{Decode}(\texttt{Reach}(o, \langle i, c\rangle)) = Tr_i^\star(\texttt{Decode}(o), \langle i, c\rangle)$.

▶ **Example 25.** For the T4240, the observable state $f_0 = \langle Dirty$=false, $Valid$=false, $Share$=false, $Exclusive$=false, $LastReader$=false$\rangle$ is the initial observable state and corresponds to the I stable state; whereas $f_1 = \langle Dirty$=false, $Valid$=true, $Share$=false, $Exclusive$=true, $LastReader$=false$\rangle$ seems to be E. When running the benchmark $\langle$`load`$, 1\rangle$ on core 1 from f_0, the reached observable state is f_1, which allows for the possibility of f_1 being E.

Observable Events

The performance registers can count the number of occurrences of predefined events. While the name and identification code for each performance event is indicated in the architecture's documentation, the meaning behind their name is not always obvious.

▶ **Property 4** (T4240 Performance Counters). *Below is a list of the events of interest, as well as their meaning based on our understanding.*

- ***L2 Data Accesses*** *Accesses made to the L2 cache.*
- ***L2 Snoop Hits*** *External queries on a memory element held by this cache.*
- ***L2 Snoop Pushes*** *Replies given to snooped queries.*
- ***External Snoop Requests*** *External queries.*
- ***L2 Reloads From CoreNet*** *Replies received.*
- ***L2 Snoops Causing MINT*** *Replies to a snooped query when holding the memory element in a dirty (modified) state.*
- ***L2 Snoops Causing SINT*** *Replies to a snooped query when holding the memory element in a clean (unmodified) state.*
- ***CPU Cycles***

▶ **Definition 26** (Observable Cache Controller Associated Events). *Let us consider an architecture with p Integer counters, an observable event f for a cache controller is a combination of values of the counters $o = \langle f_1, \ldots, f_p \rangle$. The set of cache controller associated events that can be really observed on a given architecture is denoted by $\mathcal{N}$.*

▶ **Issue 4.** *To be valid, the matching between observable states and system states must be consistent with the events associated with the transitions of both protocols. That is, for single instructions,* $\forall o \in \mathcal{R}^n,\ \forall c \leq n,\ \forall i \in \text{requests},\ \forall j \in 1..n,$
$\text{Events}[o, \texttt{Reach}(o, \langle i, c\rangle), j] = \text{NbEvents}(\texttt{Decode}(o) \rightsquigarrow^{\langle i,c\rangle} \texttt{Decode}(\texttt{Reach}(o, \langle i, c\rangle)), j)$ *where* Events *stores the performance counters, seen from core j, in the benchmark going from o to* $\texttt{Reach}(o, \langle i, c\rangle)$ *(as done in the algorithm of step 1).*

For multiple simultaneous instructions: $\forall o \in \mathcal{R}^n,\ \forall e_1, \ldots, e_n \in \text{requests} \cup \{-\},\ \forall j \in 1..n,\ \forall s \in \texttt{Reach}_m(o, \langle e_1, \ldots, e_n\rangle),\ \text{Events}[o, s, j] = \text{NbEvents}(\texttt{Decode}(o) \rightsquigarrow^{\langle e_1,\ldots,e_n\rangle} \texttt{Decode}(s), j)$.

▶ **Example 27.** Continuing Example 25, when running the benchmark $\langle \texttt{load}, 1\rangle$ on core 1 from f_0, we also collect the observable events and we observe that: on core 1, there are 2 *L2 Data Accesses* (1 for the *data-e*, all such messages are duplicated as explained in the next section) and 1 *L2 Reloads From CoreNet* (1 for the *GetS*); on core 2, there is 1 *External Snoop Requests* (1 for the *GetS*); which still allows f_1 to be E.

▶ **Definition 28** (Step 2: Reachability Analysis with Simultaneous Requests). *In addition to step 1 (see Definition 24), for the multiple simultaneous requests, we need to run additional benchmarks. The idea is similar, except that instead of running benchmark(C, ⟨instr, k⟩), we apply benchmark(C, ⟨instr$_1$, . . . , instr$_n$⟩) for the tuples ⟨instr$_1$, . . . , instr$_n$⟩ in a pre-computed list.*

▶ **Example 29.** Consider that we have run the benchmark $\langle load, store, -\rangle$ from the initial state and we believe that this coincides with the path $\langle I, I, I\rangle \rightsquigarrow^{\langle load,store,-\rangle} \langle I, M, M\rangle$ then we have to count on the core 1: 1 access to the interconnect, 2 queries and 2 data replies.

5 Evaluating Cache Coherence on the T4240

In this section, we illustrate our proposed process by attempting to validate the MESI protocol we defined on the NXP QorIQ T4240 architecture. Much to our surprise, the results quickly conclude that this architecture does not implement MESI.

5.1 The NXP QorIQ T4240 Experimental Setup

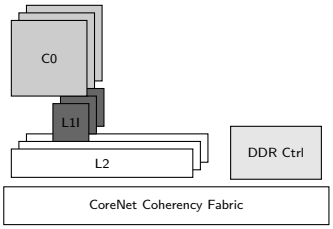


Figure 4 Configuration of the T4240.

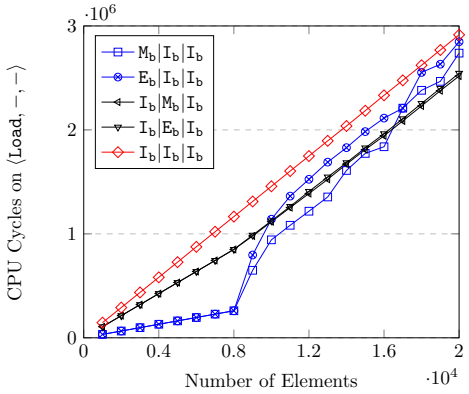


Figure 5 Exposing Cache Eviction.

In order to limit the mechanisms observed to the L2 cache coherence, we chose to disable the architecture's L1 Data caches. Furthermore, we only consider a single core (and execution thread) per cluster, and thus, per L2 cache. In an attempt at reducing the impact of instruction fetching, we keep the L1 Instruction caches enabled. Lastly, our system only uses a single memory controller. Thus, our configuration resembles the one shown in Figure 4, the remaining hardware configuration being left to what it is by default.

The NXP QorIQ T4240 architecture does not feature the `evict` instruction. The closest available instruction (`dcbi`, *Data Cache Block Invalidate*) results in the element being evicted from all the caches, which is significantly different, unless that element has been marked as ignored by cache coherence (which is then pointless for our purposes). Since our benchmarks are very small programs dealing almost exclusively with the set of experimental memory elements, we replaced the application of an `evict` on all of the memory elements with a simple invalidation of the whole local cache, which does still involve cache coherence.

While related to caches and an important factor of their performance, the issue of replacement policy is orthogonal to the cache coherence protocol. Thus, we do not want its effects to be mixed in our benchmarks. Through testing (see Figure 5), we concluded that caches started evicting cache lines when holding somewhere between 8000 and 9000 of them. From the information given in [13], we speculate that this corresponds to the 8192 cache lines held in a bank.

5.2 Partial Matching of States with Step 1

We listed and named every combination we have encountered in Table 2. We made an initial matching that seems coherent with the flags name, but that still needs to be checked by looking at the transitions. We denote by a $_b$ suffix the states observed on the platform.

We check the property required by Issue 3 that applying any request (`load`, `store`, and `evict`) from a matched state leads to the correct matched state. Table 3 shows the original and destination state for a memory element on each of the three clusters according

Table 2 Stable States of the T4240 L2 Caches Protocol.

State	*Dirty*	*Valid*	*Share*	*Exclusive*	*LastReader*
$\mathtt{M_b}$	✓	✓			
$\mathtt{E_b}$		✓		✓	
$\mathtt{I_b}$					
$\varphi_\mathtt{b}$		✓			✓
$\chi_\mathtt{b}$		✓	✓		

Table 3 State Changes.

	⟨Load,-,-⟩		⟨Store,-,-⟩		⟨Evict,-,-⟩	
Origin	**Destination**		**Destination**		**Destination**	
	Observ	**Match**	**Observ**	**Match**	**Observ**	**Match**
$\langle \mathtt{I_b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \mathtt{E_b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \mathtt{E},\mathtt{I},\mathtt{I}\rangle$	$\langle \mathtt{M_b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \mathtt{M},\mathtt{I},\mathtt{I}\rangle$	$\langle \mathtt{I_b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \mathtt{I},\mathtt{I},\mathtt{I}\rangle$
$\langle \mathtt{E_b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \mathtt{E_b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \mathtt{E},\mathtt{I},\mathtt{I}\rangle$				
$\langle \mathtt{M_b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \mathtt{M_b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \mathtt{M},\mathtt{I},\mathtt{I}\rangle$				
$\langle \mathtt{I_b},\mathtt{I_b},\mathtt{M_b}\rangle$	$\langle \varphi_\mathtt{b},\mathtt{I_b},\chi_\mathtt{b}\rangle$	$\langle \mathtt{S},\mathtt{I},\mathtt{S}\rangle$			$\langle \mathtt{I_b},\mathtt{I_b},\mathtt{M_b}\rangle$	$\langle \mathtt{I},\mathtt{I},\mathtt{M}\rangle$
$\langle \mathtt{I_b},\mathtt{I_b},\mathtt{E_b}\rangle$	$\langle \varphi_\mathtt{b},\mathtt{I_b},\chi_\mathtt{b}\rangle$	$\langle \mathtt{S},\mathtt{I},\mathtt{S}\rangle$			$\langle \mathtt{I_b},\mathtt{I_b},\mathtt{E_b}\rangle$	$\langle \mathtt{I},\mathtt{I},\mathtt{E}\rangle$
$\langle \varphi_\mathtt{b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \varphi_\mathtt{b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \mathtt{S},\mathtt{I},\mathtt{I}\rangle$			$\langle \mathtt{I_b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \mathtt{I},\mathtt{I},\mathtt{I}\rangle$
$\langle \chi_\mathtt{b},\varphi_\mathtt{b},\mathtt{I_b}\rangle$	$\langle \chi_\mathtt{b},\varphi_\mathtt{b},\mathtt{I_b}\rangle$	$\langle \mathtt{S},\mathtt{S},\mathtt{I}\rangle$			$\langle \mathtt{I_b},\varphi_\mathtt{b},\mathtt{I_b}\rangle$	$\langle \mathtt{I},\mathtt{S},\mathtt{I}\rangle$
$\langle \chi_\mathtt{b},\chi_\mathtt{b},\varphi_\mathtt{b}\rangle$	$\langle \chi_\mathtt{b},\chi_\mathtt{b},\varphi_\mathtt{b}\rangle$	$\langle \mathtt{S},\mathtt{S},\mathtt{S}\rangle$			$\langle \mathtt{I_b},\chi_\mathtt{b},\varphi_\mathtt{b}\rangle$	$\langle \mathtt{I},\mathtt{S},\mathtt{S}\rangle$
$\langle \varphi_\mathtt{b},\chi_\mathtt{b},\chi_\mathtt{b}\rangle$	$\langle \varphi_\mathtt{b},\chi_\mathtt{b},\chi_\mathtt{b}\rangle$	$\langle \mathtt{S},\mathtt{S},\mathtt{S}\rangle$			$\langle \mathtt{I_b},\chi_\mathtt{b},\chi_\mathtt{b}\rangle$	$\langle \mathtt{I},\mathtt{S},\mathtt{S}\rangle$
$\langle \varphi_\mathtt{b},\chi_\mathtt{b},\mathtt{I_b}\rangle$	$\langle \varphi_\mathtt{b},\chi_\mathtt{b},\mathtt{I_b}\rangle$	$\langle \mathtt{S},\mathtt{S},\mathtt{I}\rangle$			$\langle \mathtt{I_b},\chi_\mathtt{b},\mathtt{I_b}\rangle$	$\langle \mathtt{I},\mathtt{S},\mathtt{I}\rangle$
$\langle \mathtt{I_b},\mathtt{I_b},\varphi_\mathtt{b}\rangle$	$\langle \varphi_\mathtt{b},\mathtt{I_b},\chi_\mathtt{b}\rangle$	$\langle \mathtt{S},\mathtt{I},\mathtt{S}\rangle$			$\langle \mathtt{I_b},\mathtt{I_b},\varphi_\mathtt{b}\rangle$	$\langle \mathtt{I},\mathtt{I},\mathtt{S}\rangle$
$\langle \chi_\mathtt{b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \chi_\mathtt{b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \mathtt{S},\mathtt{I},\mathtt{I}\rangle$			$\langle \mathtt{I_b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \mathtt{I},\mathtt{I},\mathtt{I}\rangle$
$\langle \mathtt{I_b},\mathtt{I_b},\chi_\mathtt{b}\rangle$	$\langle \varphi_\mathtt{b},\mathtt{I_b},\chi_\mathtt{b}\rangle$	$\langle \mathtt{S},\mathtt{I},\mathtt{S}\rangle$			$\langle \mathtt{I_b},\mathtt{I_b},\chi_\mathtt{b}\rangle$	$\langle \mathtt{I},\mathtt{I},\mathtt{S}\rangle$
$\langle \mathtt{I_b},\varphi_\mathtt{b},\chi_\mathtt{b}\rangle$	$\langle \varphi_\mathtt{b},\chi_\mathtt{b},\chi_\mathtt{b}\rangle$	$\langle \mathtt{S},\mathtt{S},\mathtt{S}\rangle$			$\langle \mathtt{I_b},\varphi_\mathtt{b},\chi_\mathtt{b}\rangle$	$\langle \mathtt{I},\mathtt{S},\mathtt{S}\rangle$
$\langle \mathtt{I_b},\chi_\mathtt{b},\chi_\mathtt{b}\rangle$	$\langle \varphi_\mathtt{b},\chi_\mathtt{b},\chi_\mathtt{b}\rangle$	$\langle \mathtt{S},\mathtt{S},\mathtt{S}\rangle$			$\langle \mathtt{I_b},\chi_\mathtt{b},\chi_\mathtt{b}\rangle$	$\langle \mathtt{I},\mathtt{S},\mathtt{S}\rangle$
$\langle \chi_\mathtt{b},\chi_\mathtt{b},\mathtt{I_b}\rangle$	$\langle \chi_\mathtt{b},\chi_\mathtt{b},\mathtt{I_b}\rangle$	$\langle \mathtt{S},\mathtt{S},\mathtt{I}\rangle$			$\langle \mathtt{I_b},\chi_\mathtt{b},\mathtt{I_b}\rangle$	$\langle \mathtt{I},\mathtt{S},\mathtt{I}\rangle$

to what instruction was applied to the first cluster. This figure covers all the possible sets of stable states for the coherence of a single memory element on the system's clusters, since the permutation of two clusters does not impact the cache coherence's mechanisms. The transitions, however, are limited to those relevant when only a single operation is applied across the whole system. Furthermore, this does not account for any state of the coherence manager, since we are unable to observe them.

According to Table 3 we match each observed stable state with one of the formal ones. The $\mathtt{M_b}$, $\mathtt{E_b}$, and $\mathtt{I_b}$ states we observed perfectly match their `M`, `E`, and `I` counterparts from the MESI protocol. The `S` state, however, seems to match our observations of both the $\varphi_\mathtt{b}$ and $\mathtt{I_b}$ states. Indeed, when starting from $\langle \mathtt{I_b},\mathtt{I_b},\mathtt{M_b}\rangle$ and performing a `load` operation on the first cluster, we end up with two different states, $\varphi_\mathtt{b}$ and $\chi_\mathtt{b}$, where we would have expected to see two of the `S` state equivalent. The same occurs when starting from $\langle \mathtt{I_b},\mathtt{I_b},\mathtt{E_b}\rangle$. By itself, this observation is not sufficient to conclude that there is a discrepancy between the protocol we defined and the one observed on the architecture.

As we go through the different transitions from one stable state to another, we observe that performing an evict on either $\varphi_\mathtt{b}$ or $\chi_\mathtt{b}$ does not affect the other caches' state, which means that reaching either $\langle \chi_\mathtt{b},\mathtt{I_b},\mathtt{I_b}\rangle$ or $\langle \varphi_\mathtt{b},\mathtt{I_b},\mathtt{I_b}\rangle$ (or any permutation of these clusters)

is possible. In addition, the previous step showed that there is no way to have a system in which two clusters hold the same memory element in the φ_b state: the first cluster to reach the φ_b moves to the χ_b state upon seeing the other's query. Neither is it possible to have all three clusters in the χ_b state: the last cluster to load from I_b always enters φ_b, and there is no way to reach φ_b other than doing exactly that.

5.3 Consolidated Matching of States with Observable Events

Table 4 Unexpected Behaviors.

	⟨load, -, -⟩	
Origin	**Behavior**	
	Expected	**Observed**
$\langle I_b, I_b, I_b \rangle$	8000 L2D Accesses, 8000 Reloads From CoreNet	16000 L2D Accesses, 8000 Reloads From CoreNet, 1166700 CPU Cycles
$\langle I_b, I_b, \varphi_b \rangle$	8000 L2D Accesses, 8000 Reloads From CoreNet	16000 L2D Accesses, 8000 Reloads From CoreNet, 850600 CPU Cycles
$\langle I_b, I_b, \chi_b \rangle$	8000 L2D Accesses, 8000 Reloads From CoreNet	16000 L2D Accesses, 8000 Reloads From CoreNet, 1172600 CPU Cycles

	⟨-, -, load⟩	
Origin	**Behavior**	
	Expected	**Observed**
$\langle I_b, I_b, I_b \rangle$	8000 External Snoop Requests	8000 External Snoop Requests
$\langle \varphi_b, I_b, I_b \rangle$	8000 L2 Snoop Hits, 8000 External Snoop Requests	8000 L2 Snoop Hits, 8000 L2 Snoop Pushes, 8000 External Snoop Requests, 8000 SINTs
$\langle \chi_b, I_b, I_b \rangle$	8000 L2 Snoop Hits, 8000 External Snoop Requests	8000 L2 Snoop Hits, 8000 External Snoop Requests

While observing the existence of the χ_b and φ_b states may not have been sufficient to contradict a MESI protocol, they definitely did put it into question and so we prioritized furthering their analysis.

Table 4 shows our observations when loading a dataset of 8000 unique memory elements from the I_b state. The upper table indicates what is recorded on the cluster performing the `load` operations and the bottom table corresponds to what is recorded on the farthest cluster, hence the symmetry of origin state and of operation between the two tables. The $\langle I_b, I_b, I_b \rangle$ is given as a reference point. Indeed, the other lines involve either χ_b or φ_b, which we have so far assumed to be equivalent to an `S` state, meaning that the results ought to have been the same in all the lines of this first table.

The first surprising result is that we consistently observed twice the amount of expected L2D accesses. While it is odd, we do not consider it to be a sufficient contradiction of our proposed definition, as it holds true for every single one of our benchmarks.

Much more interesting is the hint of a truly unexpected behavior found in the upper table, where the $\langle I_b, I_b, \varphi_b \rangle$ benchmarks is performed using less CPU cycles than the others. Looking at what happens on the bottom table for the symmetrical line, we can see that the

cache holding the memory elements in the $\varphi_{\mathtt{b}}$ is actually providing them to the demanding cluster. This is in clear contradiction with our understanding of the architecture's protocol. Furthermore, this is not simply a case of having a different behavior for what should be the `S` state: the $\langle \chi_{\mathtt{b}}, \mathtt{I}_{\mathtt{b}}, \mathtt{I}_{\mathtt{b}} \rangle$ line of the bottom table indicates that no such thing is happening for memory elements in the $\chi_{\mathtt{b}}$ state. This allows us conclude that $\varphi_{\mathtt{b}}$ and $\chi_{\mathtt{b}}$ are, in fact, two completely separate stable states. This confirms that the NXP QorIQ T4240 architecture does not use MESI as its coherence protocol.

6 Formal MESIF Description

From the observations we made, we believe the implemented protocol to be MESIF. Table 5 shows our formal definition of the MESIF protocol.

The MESIF protocol [17] adds a *Forward* stable state. This state is equivalent to a *Shared* state with the added constraint of being responsible for the propagation of the memory element's current value. Thus making it possible to avoid reading from the system's main memory even when multiple caches hold the same memory element. Unlike the *Exclusive* state, it does not allow the cache to upgrade to a *Modified* state by itself, since the other caches still have to be informed that their copies are out-of-date.

As with any stable state that gives a cache the responsibility of propagating the memory element's current value, the challenge lies in determining when a cache can enter that state, and making sure that the responsibility is properly transferred when the cache leaves it.

The coherence manager keeps track of which cache holds memory elements in the *Forward* state. As this cache cannot actually make modifications while in this state, informing the coherence manager that it was left does not require sending any kind of `data` message: a simple `PutM` query broadcast is sufficient.

A cache moving from *Forward* to *Modified* still has to broadcast a `GetM` query and process all the queries that preceded before proceeding. We assume that if the cache still is responsible for the propagation of the memory element when it sees its own `GetM` query (meaning that it stayed in the $\mathtt{FM}^{\mathtt{B}}$ state), then it should be able to simply move to the *Modified* state without receiving any `data` reply. However, if the responsibility was lost (because of either an external `GetS` or `GetM` query), then it will need to re-acquire the current value of the memory element as a `data` reply before entering the *Modified* state.

7 Validating MESIF on the T4240

We apply again our validation strategy, this time with the MESIF protocol. First, we match the observable states with the stables: we now identify $\varphi_{\mathtt{b}}$ as corresponding to the `F` state, making the name $\mathtt{F}_{\mathtt{b}}$ more appropriate. Likewise, the $\chi_{\mathtt{b}}$ state is now named $\mathtt{S}_{\mathtt{b}}$, as it does appear to correspond to the `S` state.

Overall, our results confirm a MESIF protocol, albeit differing in some of the implementation choices. For the sake of brevity, we omitted all the results that were exactly as expected.

No `store` Optimization on `F`

Our MESIF protocol formalization considers that performing a `store` on `F` does not require a `data` reply if no other query occurs simultaneously, since that particular cache is the one in charge of distributing the value. However, the performance monitors on the T4240 show that the memory elements were actually received again (CoreNet Reloads) and that the `F`

Table 5 Description of the MESIF protocol.

Cache Controller									
State	Core Request			Interconnect Access	Data Reply		Received Queries		
	load	store	evict		data	data-e	GetS	GetM	PutM
I	GetS?, IF^{BD}	GetM?, IM^{BD}	hit				-	-	-
IF^{BD}	stall	stall	stall	$IEoF^{D}$	IF^{B}	IE^{B}	-	-	-
IF^{B}	stall	stall	stall	F			-	-	
$IEoF^{D}$	stall	stall	stall		F	E	r←s, IS^{D}	r←s, $IS^{D}I$	
IS^{D}	stall	stall	stall		r!data, r← ∅, S	r!data, m!no-data, r← ∅, S	-	$IS^{D}I$	
$IS^{D}I$	stall	stall	stall		load hit, r!data, r← ∅, I	load hit, r!data, r← ∅, m!no-data, 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		store hit, r!data, r← ∅, I		-	-	
$IM^{D}S$	stall	stall	stall		store hit, r!data, m!data, r← ∅, S		-	$IM^{D}SI$	
$IM^{D}SI$	stall	stall	stall		store hit, r!data, m!data, r← ∅, I		-	-	
S	hit	GetM?, SM^{BD}	hit, I				-	I	
F	hit	GetM?, FM^{B}	PutM?, FI^{B}				s!data, S	s!data, I	
SM^{BD}	hit	stall	stall	SM^{D}	SM^{B}		-	IM^{BD}	
FM^{B}	hit	stall	stall	M			s!data, SM^{BD}	s!data, IM^{B}	
SM^{B}	hit	stall	stall	M			-	IM^{B}	
SM^{D}	hit	stall	stall		store hit, M		r←s, $SM^{D}S$	r←s, $SM^{D}I$	
$SM^{D}I$	hit	stall	stall		store hit, r!data, r← ∅, I		-	-	
$SM^{D}S$	hit	stall	stall		store hit, r!data, m!data, r← ∅, S		-	$SM^{D}SI$	
$SM^{D}SI$	hit	stall	stall		store hit, r!data, m!data, r← ∅, 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			-	-	-
E	hit	hit, M	PutM?, EI^{B}				m!no-data, s!data, S	s!data, I	
IE^{B}	stall	stall	stall	E			-	-	-
EI^{B}	hit	stall	stall	m!no-data, I			m!no-data, s!data, II^{B}	s!data, II^{B}	
FI^{B}	hit	stall	stall	I			s!data, II^{B}	s!data, II^{B}	

Coherence Manager						
State	Received Queries				Data Reply	
	GetS	GetM	PutM (Owner)	PutM (Other)	data	no-data
I	read, s!data-e, r←s, M	s!data, r←s, M		-		
M	r←s, F^{D}	r←s	r← ∅, I^{D}	-	write, IoF^{B}	IoF^{B}
I^{D}	stall	stall	stall	-	write, resume, I	resume, I
F^{D}	stall	stall	stall	-	write, resume, F	resume, F
IoF^{B}	r←s, F	r←s, M	r← ∅, I	-	write	-
S	read, s!data, F	s!data, r←s, M		-		
F	r←s	r←s, M	r← ∅, S	-	write, IoF^{B}	IoF^{B}

cache is not sending them to itself (Snoop Pushes). This may be a standard implementation choice for MESIF, and exactly the kind we believe important for the architecture's user to know about.

	⟨store,-,-⟩	
Origin	**Behavior**	
	Expected	**Observed**
⟨E_b,I_b,I_b⟩	8000 L2D Accesses	16000 L2D Accesses, 248532 CPU Cycles
⟨F_b,I_b,I_b⟩	8000 L2D Accesses	16000 L2D Accesses, 8000 CoreNet Reloads, 252900 CPU Cycles

Odd Results with `evict` on `M`

Eviction from `M` yielded surprising results. Indeed, if not for the absence of any External Snoop Requests, these values are what one would expect to see when a cache in the `M` state sees another cache's `GetM` query. The number of L2D Accesses are not significant in this benchmark since, as previously indicated, we do not perform separate `evict` operations on each memory element but rather a general eviction of that particular cache.

	⟨evict, -, -⟩	
Origin	**Behavior**	
	Expected	**Observed**
⟨E_b,I_b,I_b⟩	8000 L2D Accesses	42 L2D Accesses, 22400 CPU Cycles
⟨M_b, I_b, I_b⟩	8000 L2D Accesses, 8000 Snoop Pushes	42 L2D Accesses, 8000 Snoop Hits, 8000 Snoop Pushes, 8000 MINTs, 65700 CPU Cycles

Better Coherence Manager

While we are unable to see the coherence manager, we still tried to expose the issue we mentioned in Example 2, where the coherence manager is unable to grant the Exclusive state when all caches evicted from `S`. As it happens, our benchmark showed that the Exclusive was indeed reached, pointing to either a better coherence manager being used, or some other co-ordination strategy.

Simultaneous Events Behaviors

Considering the limited control and observation points available to us on the platform, performing benchmark to validate the simultaneous events behaviors is particularly difficult. We have observed multiple observable states for a same combination of multiple events as expected but we are unable to detect whether the transactions that fulfilled the request of each core interlaced or if they were simply resolved in a sequence. In the latter case, all the behaviors correspond to single event ones instead. Furthermore, the possibility of an optimization being present on the architecture for certain scenarios is hardly detectable.

8 Related Works

Cache Coherence Error Detection

[11] proposes the detection of design issues in the architecture by automatically generating and performing tests on a simulation of that architecture. Indeed, while the protocol itself may be correct, its implementation and interaction with other components can still be a source of issues. In effect, this also performs a validation of the protocol on the architecture through tests, but it requires a valid model of the architecture to already be available. [20] also proposes a framework for automated test generation, this time focused on validating memory controllers according to models of the behaviors they are supposed to follow.

A number of papers propose the inclusion of hardware to implement redundant coherency mechanisms which are continuously compared with the primary ones. [6] is such a paper: it runs a simplified (stable states only) coherence protocol alongside the real one, reacting to every query and instruction. It detects local errors by comparing the cache line states according to the simplified protocol with their states according to the real protocol. It also detects errors related to the system entering an invalid coherence state by having each cache broadcast its state according to the simplified protocol so that the others can react if it is incompatible with theirs (e.g. a cache in the `M` state seeing the broadcast of another cache signaling they entered `M` as well). [33]'s solution is similar, with the exception that the states are not broadcasted. Instead, a centralized checking unit simply accesses them to check whether the system entered an invalid coherence state. In terms of detected mismatches, this is the equivalent of continuously performing the flags matching step of our strategy.

[22] proposes keeping a short backlog of relevant coherence mechanisms information (state of outgoing/incoming messages, state of cache lines) at each cycle. The information captured at each cycle is first studied in isolation, using invariants to check that the protocol would indeed allow the system to reach such a state. Then, the protocol is applied to the system state that was logged at each cycle to check that the result is compatible with the system state that was logged in the next cycle. [10] presents CoSMa, another solution making use of a backlog, but this time it is stored within the caches themselves instead of a separate component. The system periodically stops its activities to go perform a coherence check and detect if any error have occurred. The authors point out that this is not meant to be ran in production, but instead as a post-silicon validation process, which should be done prior to the product's release. Compared to the approaches from the previous paragraph, these two go further, by including the validation of behaviors.

Cache Coherence Profiling & Modeling

By successfully validating that the formally defined protocol indeed matches what is implemented on the architecture, a model of part of the platform can be created so as to verify properties relevant to the user. An important such property being the WCET, and its computation for multi-core systems is difficult and the subject of many publications. [25] provides a general survey for WCET in multi-core systems, and [24] provides another survey, this time focused on caches.

Much like our own approach, [4] and [16] make use of benchmarks and monitors to learn the characteristics of a given architecture. Their focus is on exposing unexpectedly shared resources by overwhelming them through stress testing.

We ourselves have used timed automata for the modeling of platform and program in [30], with a focus on the identification of interferences (negative impact caused by external queries). Timed automata were also used in models aimed at WCET computation [8, 19].

WCET is also strongly impacted by cache eviction policies, which we did not address in this paper. Solutions to analyze the impact of cache eviction are plenty [15, 18, 23, 32], especially since this problematic predates multicore processors.

To ease WCET computation, some cache coherence protocols are designed to be predictable in their impact on runtime, such as [21]. This does not, however, remove the need for the coherence protocol implementation to be properly identified and validated. Another way to make WCET computation easier is to limit what is being affected by cache coherence. For example, [7] leverages sensible scheduling so that parts of the program that require access to the bus are less likely to be happening simultaneously; [27] also makes use of careful scheduling, this time so that tasks can simply leave their results in cache so that it will be used by the next task without needing to be fetched; [2] suggests making use of the platform capabilities to better control what is kept in what cache, and what should be affected by coherence mechanisms.

Even if not interested in an easy to predict WCET, good understanding of the impact of cache coherence can be used to improve performance and/or reduce wasteful operations. [3] adds hardware that will consider each cache line as being write-through or write-back depending on what is preferable.

9 Conclusion

In this paper we presented a strategy to validate the user's understanding of the cache coherence mechanisms implemented on an architecture. To illustrate our process, we applied it to the NXP QorIQ T4240 architecture, which we understood to be running MESI. We thus proposed a formal definition for a split-transaction bus MESI protocol, which we tried to validate using the aforementioned process. To our surprise, where we expected to only see differences in implementation choices, we learned that the architecture is in fact implementing MESIF. We validated this by proposing a formal definition for that protocol and re-applying the process a second time. This time, the results indicated a match, with the exception of a few implementation choices.

In the future, we will make measure on more temporal behavior from the NXP QorIQ T4240 architecture relative to the cache coherence to quantify the impact induced by cache coherence on software running on the cores. We will also extend our Uppaal model from [30] to integrate several cache protocol, including MESIF, and to use real delay values so as to be able to offer the formal model of a validated system.

References

1 Jyotika Athavale, Riccardo Mariani, and Michael Paulitsch. Flight safety certification implications for complex multi-core processor based avionics systems. In *25th IEEE International Symposium on On-Line Testing and Robust System Design, IOLTS 2019, Rhodes, Greece, July 1-3, 2019*, pages 38–39, 2019.

2 Ayoosh Bansal, Jayati Singh, Yifan Hao, Jen-Yang Wen, Renato Mancuso, and Marco Caccamo. Cache where you want! reconciling predictability and coherent caching, 2019. `arXiv:1909.05349`.

3 Pedro Benedicte, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. HWP: Hardware Support to Reconcile Cache Energy, Complexity, Performance and WCET Estimates in Multicore Real-Time Systems. In *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, volume 106 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:22, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.ECRTS.2018.3`.

4 Jingyi Bin, Sylvain Girbal, Daniel Gracia Perez, Arnaud Grasset, and Alain Merigot. Studying co-running avionic real-time applications on multi-core cots architectures. In *Embedded Real Time Software and System Conference (ERTS2)*, February 2014.

5 Frédéric Boniol, Youcef Bouchebaba, Julien Brunel, Kevin Delmas, Thomas Loquen, Alfonso Mascarenas Gonzalez, Claire Pagetti, Thomas Polacsek, and Nathanaël Sensfelder. PHYLOG certification methodology: a sane way to embed multi-core processors. In *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*, Toulouse, France, January 2020. URL: `https://hal.archives-ouvertes.fr/hal-02441323`.

6 Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. *Dynamic Verification of Cache Coherence Protocols*, pages 25–42. Springer New York, New York, NY, 2004. `doi:10.1007/978-1-4419-8987-1_3`.

7 Thomas Carle and Hugues Cassé. Reducing Timing Interferences in Real-Time Applications Running on Multicore Architectures. In *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*, volume 63 of *OpenAccess Series in Informatics (OASIcs)*, pages 3:1–3:12, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/OASIcs.WCET.2018.3`.

8 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`.

9 Certification Authorities Software Team. Multi-core Processors - Position Paper. Technical Report CAST 32-A, Federal Aviation Administration, November 2016.

10 A. DeOrio, A. Bauserman, and V. Bertacco. Post-silicon verification for cache coherence. In *2008 IEEE International Conference on Computer Design*, pages 348–355, October 2008. `doi:10.1109/ICCD.2008.4751884`.

11 M. Elver and V. Nagarajan. Mcversi: A test generation framework for fast memory consistency verification in simulation. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 618–630, March 2016. `doi:10.1109/HPCA.2016.7446099`.

12 Hakan Forsberg and Andreas Schwierz. Emerging cots-based computing platforms in avionics need a new assurance concept. In *the 38th Digital Avionics Systems Conference (DASC'19)*. IEEE Press, 2019.

13 Freescale. e6500 core reference manual, rev 0, 2014.

14 Freescale. T4240 QorIQ: Integrated multicore communications processor family reference manual, 2014.

15 Michele Garetto, Emilio Leonardi, and Valentina Martina. A unified approach to the performance analysis of caching systems. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 1(3), May 2016. `doi:10.1145/2896380`.

16 Sylvain Girbal, Jimmy le Rhun, and Hadi Saoud. METrICS: a measurement environment for multi-core time critical systems. In *9th European Congress on Embedded Real Time Software and Systems (ERTS'18)*, 2018.

17 James Goodman and Hhj Hum. Mesif: A two-hop cache coherency protocol for point-to-point interconnects (2004), 2004.

18 Daniel Grund and Jan Reineke. Toward Precise PLRU Cache Analysis. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 23–35, 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.2010.23`.

19 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`.

20 M. Hassan and H. Patel. Mcxplore: Automating the validation process of dram memory controller designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(5):1050–1063, May 2018. doi:10.1109/TCAD.2017.2705123.

21 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.

22 B. Kumar, A. K. Bhosale, M. Fujita, and V. Singh. Validating multi-processor cache coherence mechanisms under diminished observability. In *2019 IEEE 28th Asian Test Symposium (ATS)*, pages 99–995, 2019.

23 Benjamin Lesage, David Griffin, Sebastian Altmeyer, Liliana Cucu-Grosjean, and Robert I. Davis. On the analysis of random replacement caches using static probabilistic timing methods for multi-path programs. *Real-Time Syst.*, 54(2):307–388, April 2018. doi:10.1007/s11241-017-9295-2.

24 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):05–1–05:48, 2016. doi:10.4230/LITES-v003-i001-a005.

25 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 Comput. Surv.*, 52(3), June 2019. doi:10.1145/3323212.

26 Laurence Mutuel, Xavier Jean, Vincent Brindejonc, Anthony Roger, Thomas Megel, and E. Alepins. Assurance of Multicore Processors in Airborne Systems, 2017.

27 Viet Anh Nguyen, Damien Hardy, and Isabelle Puaut. Cache-conscious Off-Line Real-Time Scheduling for Multi-Core Platforms: Algorithms and Implementation. *Real-Time Systems*, pages 1–37, 2019. doi:10.4230/LIPIcs.ECRTS.2017.14.

28 Jan Nowotsch and Michael Paulitsch. Leveraging multi-core computing architectures in avionics. In *Proceedings of the 2012 Ninth European Dependable Computing Conference*, EDCC '12, pages 132–143, Washington, DC, USA, 2012. IEEE Computer Society.

29 Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. *SIGARCH Comput. Archit. News*, 12(3):348–354, January 1984. doi:10.1145/773453.808204.

30 Nathanaël Sensfelder, Julien Brunel, and Claire Pagetti. Modeling Cache Coherence to Expose Interference. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:22, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECRTS.2019.18.

31 Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011.

32 Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. Fast and exact analysis for lru caches. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290367.

33 Hui Wang, Sandeep Baldawa, and Rama Sangireddy. Dynamic error detection for dependable cache coherency in multicore architectures. *21st International Conference on VLSI Design (VLSID 2008)*, pages 279–285, 2008.

34 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.

Simultaneous Multithreading and Hard Real Time: Can It Be Safe?

Sims Hill Osborne
University of North Carolina, Chapel Hill, NC, USA
http://www.cs.unc.edu/~shosborn/
shosborn@cs.unc.edu

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

Abstract

The applicability of Simultaneous Multithreading (SMT) to real-time systems has been hampered by the difficulty of obtaining reliable execution costs in an SMT-enabled system. This problem is addressed by introducing a scheduling framework, called CERT-MT, that combines scheduling-aware timing analysis with a cyclic-executive scheduler in a way that minimizes SMT-related timing variations. The proposed scheduling-aware timing analysis is based on maximum observed execution times and accounts for the uncertainty inherent in measurement-based timing analysis. The timing analysis is found to work for tasks with and without SMT, though some adjustments are required in the former case. A large-scale schedulability study is presented that shows CERT-MT can schedule systems with total utilizations approaching 1.4 times the core count, without sacrificing safety.

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

Keywords and phrases real-time systems, simultaneous multithreading, hard real-time, scheduling algorithms, probability, statistics, timing analysis

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.14

Related Version Longer version with all graphs, code, and data available at http://jamesanderson.web.unc.edu/papers/

Supplementary Material ECRTS 2020 Artifact Evaluation approved artifact available at https://doi.org/10.4230/DARTS.6.1.1.

Funding Work was supported by NSF grants CNS 1563845, CNS 1717589, and CPS 1837337, ARO grant W911NF-17-1-0294, ONR grant N00014-20-1-2698, and funding from General Motors.

Acknowledgements We thank Prof. Alex Mills (Zicklin School of Business, Baruch College, New York, NY) for reviewing an early draft and offering suggestions, particularly with regards to Sec. 4. We thank Joshua Bakita (UNC-Chapel Hill) for contributions to the code used to execute our benchmark tests. Finally, we thank our anonymous reviewers and Shepherd for the improvements they suggested to the final version of this paper.

1 Introduction

Simultaneous Multithreading (SMT) is a technology that allows a single physical computing core to act as two or more logical cores, or hardware *threads*. Ideally, enabling SMT allows multiple jobs to execute in parallel on a single core in significantly less time than would be required for the same jobs to execute sequentially. If this ideal can be achieved, the potential benefits are clear. Users in industry are eager to make more use of SMT; in particular, multiple developers have expressed interest to the U.S. Federal Aviation Administration (FAA) in using SMT in safety-critical systems [55].

Unfortunately, with SMT enabled, a task's execution time is dependent on code that executes on other thread(s) of the same core. This fact makes it difficult to obtain worst-case execution times (WCETs) with SMT enabled, to the point that the real-time community has largely given up on SMT for hard real-time systems. However, multicore systems face problems with interdependence between scheduling and execution times even without SMT. Such problems make static timing analysis on multicore platforms extremely difficult, which has led to interest in Measurement Based Probabilistic Timing Analysis (MBPTA), a family of methods that determine probabilistic worst-case execution times (pWCETs) by sampling observed execution times [17, 20].

Considered problem. We claim the following: *if a hard real-time task system τ can be guaranteed a given level of safety on a modern multicore architecture via MBPTA, enabling SMT can, in many cases, allow τ to run on fewer cores without sacrificing safety.* We provide a formal definition of safety in Sec. 2. We justify this claim by providing and examining CERT-MT (**C**ontrolled **E**xecution of **R**eal **T**ime with **M**ulti-**T**hreading), a scheduling framework based on a cyclic-executive scheduler and MBPTA timing analysis.

The CERT-MT framework. CERT-MT is a framework that includes both scheduling analysis and timing analysis designed to minimize uncertainties resulting from SMT. The overall structure of CERT-MT is illustrated in Fig. 1. The timing analysis of CERT-MT is based on MBPTA. We add to the MBPTA family of techniques by analyzing the probabilistic and statistical implications of basing timing analysis on worst observed execution times. As part of this analysis, we point out a potential source of error: the measurements used in MBPTA to build estimates may themselves be subject to stochastic variation.

Historically, there has been a separation of concerns in the real-time community between timing analysis and schedulability analysis. There are good reasons for this separation: both topics are complex on their own, making it difficult to address both within a single work. However, maintaining this separation comes at the cost of imprecise timing analysis that can be made safe only by resorting to more pessimism than needed. To avoid both unsafe conditions and excessive pessimism, CERT-MT is built around two key rules: first, conditions when testing tasks with MBPTA should match runtime conditions as closely as possible; second, timing analysis and scheduling decisions should be considered at the same time, since scheduling can affect task execution times. These are well-known, standard principles of measurement-based analysis; we consider scheduling and timing analysis simultaneously in deference to these principles.

Contribution and organization. We give an MBPTA method that is straightforward to use and, given basic assumptions about the underlying data, provides mathematically guaranteed results. Using this method, we analyze the execution times of tasks employing SMT and thereby show that using SMT can bring substantial benefits to hard real-time systems without sacrificing safety.

More specifically, in Sec. 3 we define the CERT-MT scheduler, which is designed to minimize variations of job execution times caused by SMT, and in Sec. 4, we give our new MBPTA method, which upper-bounds the likelihood of an observed maximum execution time being exceeded. This method, which is also applicable to systems without SMT, breaks new ground by accounting for the fact that if measurements contain any random elements, then any measurement-based estimate is itself a random variable. The theoretical work of Secs. 3 and 4 is backed by empirical results.

In Sec. 5, we present our experiments. First, in Sec. 5.1, we show that our timing analysis works very well in practice, even when certain assumptions underlying our analysis do not hold. Second, in Sec. 5.2, we discuss how enabling SMT affects the timing requirements of individual tasks. Finally, in Sec. 5.3, we give the results of a large-scale schedulability study built around the understanding of SMT behavior we have developed. In this study, CERT-MT enabled systems with total utilizations approaching 140% of the core count to be scheduled with no loss of safety. In Sec. 6, we conclude and discuss future research directions.

We add to existing work on MBPTA, but there are many elements within MBPTA we do not address. In order to focus on the role of SMT, we do not consider the role of interference from tasks on other cores, nor do we address the complex question of how to determine appropriate task inputs for timing analysis. For our benchmark tasks, we used the built-in inputs and proceeded to work under the assumption that they are appropriate. Finally, we do not consider how the probability of individual jobs exceeding stated costs affects the probability of larger system failures. We plan to address these topics in future work.

2 Background

In this section, we provide some basic background information about SMT, define our task and platform models, and give a brief overview of concepts related to MBPTA and pWCETs.

SMT basics. Modern superscalar cores execute multiple instructions per cycle, using instruction-level parallelism within jobs to decrease execution time. SMT extends this behavior by allowing multiple jobs to execute instructions within a single cycle. An overview is given in Ex. 1 below. More details can be found in Eggers et al. [22].

▶ **Example 1.** Fig. 2 shows the effects of enabling SMT. At the top of Fig. 2, jobs of tasks τ_1 and τ_2 execute sequentially without SMT on a core that can accept two instructions per cycle. When fewer than two instructions are ready, as in cycles 2 and 3, execution resources are wasted. τ_1 finishes in cycle 6 and τ_2 in cycle 12.

In the middle of the figure, the same jobs execute in parallel with SMT enabled, reducing the number of lost cycles. Both tasks finish in cycle 8. In this case, SMT has the effect of delaying the completion of the darker-colored τ_1, but speeding up the completion of the lighter τ_2, since it does not have to wait for τ_1 to complete before beginning its own execution.

The bottom shows SMT execution with the start of τ_2 delayed; this delay actually causes τ_1 to require more time to execute due to different interactions between the two tasks. For a detailed discussion of factors that can affect SMT execution, see Bulpin [10, 11].

Hardware platform. Our hardware platform π consists of m identical cores, each supporting two hardware threads. Different threads on the same core are referred to as *sibling threads*. Jobs scheduled on sibling threads are *sibling jobs*. Sibling jobs are said to be *co-scheduled*. It is not possible to adjust the priority of sibling jobs relative to one another; in particular, switching which job executes on which sibling thread has no effect. These conditions match those of Hyperthreading, Intel's SMT implementation [25]. Jobs that do not use SMT are referred to as *solo jobs*.

Task model. We consider a hard real-time task system τ that consists of n independent synchronous periodic, implicit-deadline tasks. More specifically, we assume there are no inter-task precedence constraints, critical sections, or other synchronization requirements. Each

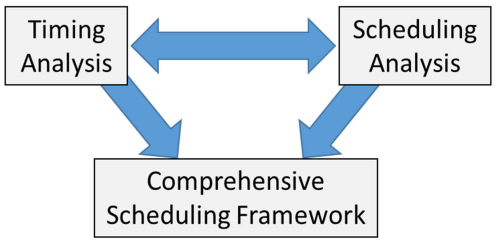


Figure 1 The CERT-MT framework.

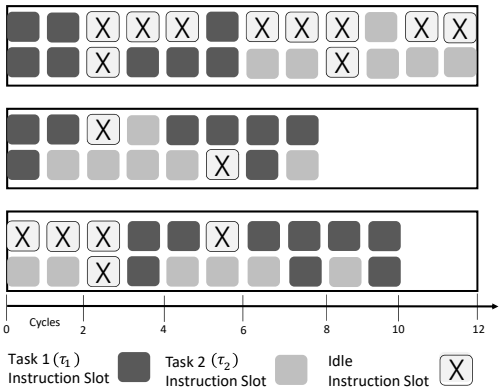


Figure 2 Top: task execution without SMT. Middle: execution with SMT. Bottom: execution with SMT, different starting times.

task $\tau_i = (C_i, T_i)$ is defined by its execution *cost,* C_i, and its *period,* T_i. Time is continuous. 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 a^{th} job released by τ_i denoted by $\tau_{i.a}$. Jobs of τ_i are released every T_i units of time and have a relative deadline of T_i; the a^{th} job is released at time $r_{i.a} = T_i(a-1)$ and has an absolute deadline at time $d_{i.a} = T_i \cdot a$. The system is scheduled correctly if it can be shown that no job will ever miss a deadline if all task execution costs are upper-bounded by C_i.

Typically, C_i is said to give a task τ_i's WCET. Our C_i values are determined based on pWCETs, or probabilistic WCETs. We give a more precise definition of C_i below, after establishing some necessary statistical concepts.

We require task periods to be harmonic, i.e., every period must be an integer multiple of every smaller period. The least common multiple of all periods is referred to as the *hyperperiod* and denoted H. We add to this task model random variables that correspond to the execution times of individual jobs:

▶ **Definition 2.** *E_i is a random variable corresponding to the execution time of a randomly selected job of τ_i.*

pWCETs and system safety. There are two problems with modeling a task based on its WCET. First, determining the true WCET may be prohibitively difficult, especially on a multicore machine. Second, if the WCET can be determined, it may be a very rare event that a job actually requires that many units of execution time, leading to very pessimistic scheduling if it is assumed that every job of τ_i will require its WCET; depending on the application, this pessimism may be unnecessary. As an alternative to WCETs, we could consider pWCETs instead.

▶ **Definition 3.** *τ_i has the* pWCET C_i^p *if and only if execution times for τ_i follow a probability distribution*[1] *such that*

$$\Pr(E_i \leq C_i^p) = p \tag{1}$$

holds. We refer to p as the provisioning level *of τ_i.*

[1] In some sources, the distribution itself is referred to as the pWCET

However, finding a task's pWCET has its own problems. Validating that C_i^p is a pWCET for a task τ_i requires a detailed understanding of the task's behavior in many possible execution states and of the likelihoods of those states occurring. It can be argued that determining a pWCET is no more practical than determining a WCET. Furthermore, it is potentially dangerous to determine a pWCET based only on observed run-times. To understand why, consider a simpler problem: rolling a die. Finding exact probabilities of a given die roll result, which is analogous to a pWCET, requires detailed knowledge of the die's physical properties, which is analogous to perfect knowledge of a task's behavior. Lacking detailed knowledge of a die or task, we can make predictions based on observed past rolls or execution times. This method is practical, but does not produce a pWCET for the task.

▶ **Example 4.** We wish to know Pr(an arbitrary future roll ≤ 5) for a given die, but we cannot examine the die directly. We only know the results of rolling the die in the past. Suppose that out of ten roles, the maximum was 5. Any of the following are possible: the die has no faces greater than 5, the die has faces greater than 5 but these are unlikely to be rolled, or the die has faces greater than 5, which are reasonably likely to be rolled, but our set of ten rolls was not representative of the die's true long term behavior. To illustrate the last case, rolling a fair six-sided die ten times gives

$$\Pr(\text{ten rolls} \leq 5 \mid \text{a fair six-sided die}) = \left(\frac{5}{6}\right)^{10} \approx 0.162.$$

Consequently, a naive interpretation of observed results, such as the rule that getting zero 6s out of ten rolls means a 6 is less likely than other roles, will frequently be wrong.

Any estimate we can collect from measured random data – such as the observed maximum given ten rolls – is a random variable, but to make probabilistic statements of the form Pr(arbitrary future roll or execution time $\leq Y$) where Y is a constant rather than a random variable, we need more information than can be obtained by observing results. For a die, this would mean physically examining it. For a task, we would need static analysis, which is impractical for tasks utilizing SMT or in multicore systems. Since static timing analysis is impractical, we are left with measurement-based methods. With MBPTA, each trial is used to gather data, similarly to each roll of a die. A set of execution times is called a *trace*.

▶ **Definition 5.** *A trace is an ordered set of execution times for a task and constitutes a finite-sized sample from the population of all possible execution times for that task. For trace R with $|R|$ elements, let $\{R_1, R_2, ...R_{|R|}\}$ be the ordered sample times within the trace. Let the maximum of the trace be denoted by R_{max}. Note that the R_k values refer to times within a finite-sized sample, while E_i is from the entire population of execution times.*

The trace is analyzed to produce an estimate of future behavior. How to do so safely lies at the heart of MBPTA analysis. EVT methods attempt to use traces that are representative of possible task behaviors to produce an upper bound on C_i^p, typically much larger than the observed maximum execution time, for a given value of p. By doing so, practitioners can avoid many hazards associated with estimates based on purely random data. Unfortunately, how to obtain data that is sufficiently representative is very much an open question; there is no universal agreement on how to do so [17, 20, 36]. Consequently, it is not always possible to correctly apply EVT.

We use a different approach. We analyze a trace to produce a *safe WCET*, which accounts for both randomness in future behavior and randomness involved in creating the estimate.

▶ **Definition 6.** *Given a task τ_i, a* safe WCET S_i^q *is a random variable such that*

$$\Pr(E_i \leq S_i^q) = q \tag{2}$$

holds given trace R of τ_i, where R constitutes a random sampling of τ_i's possible execution times. We refer to q as the safety level[2] *of τ_i.*

The difference between Defs. 3 and 6 is that, while C_i^p in Def. 3 is a constant, S_i^q is a random variable. In practice, the execution times forming R may not *actually* be random, but we analyze R *as if* they were. We formally state our assumptions regarding R in Sec. 4. Later, in Sec. 5, we show empirically that our analysis holds even when applied to execution traces for which our assumptions of randomness may not. This approach of analyzing non-random data as if it were random may seem strange, but it is not original to us. In particular, pseudo-random number generators (pRNGs) actually produce values that are purely deterministic, but are then used to produce "randomess" that powers a vast variety of applications across many domains. Our traces do not exhibit the same degree of random-like behavior that is expected of widely-used pRNGs, but our experiments in Sec. 5 show the power of assuming randomness to gain insight into task behaviors.

Costs and correctness revisited. We define the execution-cost parameter C_i in our model so that $C_i = S_i^q$ holds for a specified q. The traditional notion of correctness – all jobs are guaranteed to complete on time – cannot be adhered to without known upper bounds on all job execution times. For this reason, we supplement the binary idea of correctness with a quantifiable level of *safety*.

▶ **Definition 7.** *Task system τ is q-safe if all tasks have sWCETs with safety level at least q, and the system would be correctly scheduled if all tasks had true WCETs no greater than the stated sWCETs.*

Some may question the wisdom of using sWCETs within a safety-critical context. However, probabilistic reasoning is already present within hard real-time contexts. In particular, FAA standards for commercial aircraft state acceptable failure rates, essentially giving a probabilistic bound. *We are not weakening hard real-time correctness; we are making explicit a dependence on timing analysis that is often left implicit.*

Determining an appropriate safety level is an application-specific decision. Our concept of safety is applicable to any system designed around an acceptable failure rate, which could include applications ranging from streaming media to aviation. We highlight aviation rules to show that probabilistic timing analysis does not preclude systems that have both hard real-time and safety-critical requirements.

Related work. The foundations of SMT were given by Tulsen et al. [67] and Eggers et al. [22] in the mid 1990s. SMT first became widely available in 2002, when it was made available on Intel processors [50], prompting works that gave detailed descriptions of SMT on Intel hardware [10, 11, 34, 64, 66], albeit outside of a real-time context.

Jain et al. [35] did early work on applying SMT to real time; they showed experimentally that significant performance gains in soft real-time systems are possible. Kato et al. [37, 38] gave algorithms for determining which tasks should share a core to maximize efficiency, provided that it is possible to statically determine execution costs of SMT-enabled tasks.

[2] Note that q is unrelated to safety integrity levels used in risk analysis, despite the similar name.

Cazorla et al. [16] and Gomes et al. [26, 27] have proposed ways to exert greater control over SMT timings by modifying the interaction between the operating system and the hardware, while Zimmer et al. [73] and Suito et al. [65] have proposed purpose-built hardware aimed towards real-time work with multithreading. Lo et al. [48] gave methods to limit real-time work to a small number of threads, executing non-real-time work only when safe to do so. Mische et al. [53] suggested using SMT to hide context-switch costs by using threads to switch task state in and out in the background. More recently, Osborne et al. [56, 58] have benchmarked SMT performance and given schedulability results within a soft real-time context based on worst-case observed execution times. Detailed analysis of Intel's microarchitecture, including the resource constraints that are relevant with SMT, have been performed by Fog [25]. Scheduling with SMT is related to the more general case of scheduling in parallel any set of tasks that may influence one another's execution times, as discussed by Andersson et al. [4].

Cyclic-executive (CE) scheduling was developed to combine high predictability with low run-time overheads [5, 47]. The essence of the method is to pre-compute a table stating when every job should execute. Recent work on CE scheduling for multicore platforms has focused on mixed-criticalty scheduling [14, 12] and more efficient schedule computation [21].

The difficulties with guaranteeing execution costs in a multicore context arise from the possibility of tasks on different cores interfering with one another. An overview of these problems, and possible solutions, is given by Kim in his dissertation [40]. More specific concerns include cache conflicts [2, 9, 19, 41, 49, 69, 70], DRAM conflicts [29, 30, 68, 71, 72], memory bus conflicts [54, 63], general OS support [3, 18, 31], and I/O conflicts [39, 59].

MBPTA was first proposed by Burns and Edgar in 2000 [13], in part as a means to address the difficulty of timing analysis on complex processors; concerns over the practicality of static timing analysis predate multicore computing. Traditionally, EVT has required that execution times be identically and independently distributed [6, 24, 60]. This requirement has been an obstacle to the use of EVT, since observed execution times will often include statistical dependencies. One way real-time practitioners have worked around this limitation by has been by requiring that executions be performed with randomized caches [1, 7, 8, 32, 42, 43, 51, 52]. Others have explored ways to introduce randomness to execution traces after measurement is complete [45, 46]. Alternatively, the works of Leadbetter et al. [44] and Hsing [33] imply that variables need not be identically distributed so long as they are *stationary* – a data set is stationary if it maintains the same distribution over time – and either all values are independently distributed over time or extreme values are independent with respect to time (extremal independence). This fact has been used by Guet, Santinelli, and various co-authors in [28, 61, 62] and has been noted by Carzola et al. and Davis and Cucu-Grosjean in recent survey papers [17, 20].

3 The CERT-MT Scheduler

In this section, we describe the CERT-MT scheduler, which is a CE scheduler that allows for SMT. We do so in two parts. First, we describe the restrictions we place on jobs using SMT that will make our timing analysis possible. Second, we discuss the correctness conditions of CE schedulers and outline the construction of a mathematical optimization program that will produce a correct schedule for SMT-enabled tasks that adheres to our rules if one exists. We will return to the topic of timing analysis in Sec. 4.

3.1 Rules for Using SMT

Again, the fundamental rule of CERT-MT is that *jobs employing SMT must execute as they are tested in determining execution times.*[3] To enforce this rule, we implement the following two sub-rules:

1. All jobs employing SMT must be *simultaneously co-scheduled,* as defined in Def. 8 below.
2. All jobs employing SMT must be executed non-preemptively. We include scheduler interruptions as preemptions.

▶ **Definition 8.** *We say that two jobs are* simultaneously co-scheduled *if both begin execution simultaneously on sibling threads of the same core, and when one job completes, the remaining job continues on the same core with no sibling job until complete. We use $\tau_{i.a:j.b}$ to denote the simultaneously co-scheduled jobs $\tau_{i.a}$ and $\tau_{j.b}$ and $\tau_{i:j}$ to denote non-specific simultaneously co-scheduled jobs of τ_i and τ_j.*

With these sub-rules in place, we can enforce our primary rule – execution as tested – without creating an excessive timing-analysis burden. The first sub-rule allows us to limit our testing to simultaneously co-scheduled jobs, as opposed to considering, for example, what happens if τ_i begins executing as a solo job and τ_j begins later on the same core, similar to what we saw in the final case of Ex. 1 and Fig. 2. We thereby ensure that our timing analysis accounts for hardware resource sharing between jobs on the same core. From here onwards, we use the term "co-scheduled" to mean "simultaneously co-scheduled" since that is the only scheme we consider. Banning preemptions for SMT-enabled jobs is a precautionary step; while there exists substantial work on accounting for preemptions without SMT, no such work exists for SMT. In its absence, we choose to use a conservative approach. We allow for preemptions when not using SMT.

Disadvantages of non-preemptive scheduling. While requiring non-preemptive execution simplifies our timing analysis, it can make otherwise schedulable systems unschedulable, as shown in Ex. 9 below. The example does not explicitly reference SMT or co-scheduled jobs; we explain later how co-scheduled jobs can be treated as a single schedulable unit.

In a CE schedule, jobs are assigned to execute within specific time intervals called *frames* [5]. A correct schedule is built by assigning jobs to frames. We give a precise definition of frames – our definition elaborates on the standard one – in Def. 10 below, following Ex. 9.

▶ **Example 9.** Let τ be a task system with periods in the set $\{10, 20\}$, with some jobs having $T_i = 20$ and $C_i > 10$. Consider the frame size f needed to execute τ non-preemptively. If $f > 10$ holds, then no job with $T_i = 10$ can be scheduled, since the first frame will not complete until after the deadline of the first job, so such a job could miss its deadline. If $f \leq 10$ holds, then no job with $C_i > 10$ can be scheduled, since the frame boundary causes a preemption by the scheduler.

To facilitate the scheduling of non-preemptive jobs, we allow the frame size to be defined per-core, rather than requiring one frame size for the entire platform. We replace the notation f for a platform-wide frame size – the standard in CE scheduling – with $f(\ell)$.

[3] Recall that we consider only independent tasks; we defer more complex systems to future work.

▶ **Definition 10.** *A* frame *is a time interval of length $f(\ell)$ on core ℓ. Frames are indexed using g, starting from $g = 1$, such that the g^{th} frame on core ℓ starts at time $(g-1) \cdot f(\ell)$ and ends at time $g \cdot f(\ell)$. We require that no frame extends beyond the hyperperiod boundary,*[4] *allowing the schedule to repeat every hyperperiod. The last frame per hyperperiod on core ℓ has index $g = \left\lfloor \frac{H}{f(\ell)} \right\rfloor$.*

As a consequence of allowing multiple frame sizes, frames with the same indices may cover different lengths of time on different cores. For example, suppose that on a two-core system, $f(1) = 10$ and $f(2) = 20$. Frames 1, 2, and 3 of core 1 would start at times 0, 10, and 20, but frames 1, 2, and 3 of core 2 would start at times 0, 20, and 40. In addition, the maximum valid frame index varies based on $f(\ell)$; since we only need to define a schedule for the first hyperperiod, we do not consider frames for which $g \cdot f(\ell) > H$ holds.

Scheduling co-scheduled jobs. We define three scheduling parameters for co-scheduled job pairs: *joint cost*, *joint release*, and *joint deadline*. These parameters allow us to treat pairs of jobs as schedulable entities.

▶ **Definition 11.** *The* joint cost *to simultaneously execute jobs of τ_i and τ_j is given by $C_{i:j}$, defined as the execution time for both jobs assuming they begin simultaneously. If $i = j$, then $C_{i:j} = C_i$, indicating solo execution for τ_i. Jobs with nothing co-scheduled are* solo jobs*.*

As with single-task costs, determining $C_{i:j}$ such that both jobs are absolutely guaranteed to complete is impractical. For this reason, we define $C_{i:j} = S^q_{i:j}$ for a specified value of q, with $S^q_{i:j}$ analogous to S^q_i (Def. 6).

▶ **Definition 12.** *Let $E_{i:j}$ be a random variable corresponding to the time required to simultaneously co-schedule a pair of randomly selected jobs of τ_i and τ_j. We define the* joint safe WCET *(jsWCET) $S^q_{i:j}$ of $\tau_{i:j}$ to be a function of a trace R of $\tau_{i:j}$ such that*

$$\Pr(E_{i:j} \leq S^q_{i:j}) = q \tag{3}$$

holds. As with solo tasks, we assume R is a random sample, making $S^q_{i:j}$ a random variable.

Our definition of safety (Def. 7) can be easily expanded to accommodate jsWCET values. Along with defining a joint cost for each pair, we define a *joint release* and *joint deadline*.

▶ **Definition 13.** *Given $\tau_{i.a:j.b}$, the* joint release *and* joint deadline *are given, respectively, by*

$r(i.a, j.b) = \max(T_i \cdot (a-1), T_j \cdot (b-1))$ *and*
$d(i.a, j.b) = \min(T_i \cdot a, T_j \cdot b)$.

For the case where $i = j$ and $a = b$ – i.e., a solo job – the r and d terms are simply the job's release time and absolute deadline.

In order for both jobs of a pair to finish on time, the pair must begin no sooner than $r(i.a, j.b)$ – both jobs must have been released – and must finish no later than $d(i.a, j.b)$ – prior to either jobs' deadline.

[4] This requirement weakens the typical CE requirement that $f(\ell)$ divide H while still allowing the schedule over each hyperperiod to repeat.

3.2 Creating a Schedule

In this subsection, we review what it means for a CE scheduler to be correct and expand upon standard CE rules to apply to SMT and when frame size can vary. For now, we assume we have already determined safe execution costs for all tasks and pairs of co-scheduled tasks. We will discuss how to determine such costs in Secs. 4 and 5. A CE schedule is correct if the rules stated in Def. 14 below, adapted from Baker and Shaw [5], hold.

▶ **Definition 14.** *A CE schedule is* correct *if over the course of each hyperperiod: (i) all jobs are scheduled; (ii) any non-preemptable job is scheduled in exactly one frame; (iii) every job completes in a frame that ends no later than its deadline; (iv) no job executes in a frame that begins before its release; (v) the total execution time scheduled in each frame is no greater than the frame size; and (vi) no job executes in parallel with itself.*

If we allow a particular job to execute on multiple cores, rule (vi) of Def. 14 becomes quite challenging. For this reason, we require that each job, with or without SMT, executes on only one core. However, a task may execute different jobs on different cores.

In a conventional CE scheduler, not permitting SMT and requiring one frame size across all cores, the assignment of jobs or job portions to frames so that the system is schedulable is the primary decision to make. With CERT-MT, additional decisions are needed: what frame size should be used for each core, on what core should each job be scheduled – with different frame sizes, cores are not interchangeable – and how, if at all, should jobs be co-scheduled? Even so, the requirements given for correctness in Def. 14 are unchanged.

However, these correctness requirements are more complicated, particularly since the scheduling decisions needed cannot be made independently. To make them, we employ a mathematical optimization program. To convert our correctness conditions into mathematical form, we define a variable for every possible job pair, core, and frame.

▶ **Definition 15.** *For $i \neq j$, let the variable $x(i.a, j.b, \ell, g)$ be defined as*

$$x(i.a, j.b, \ell, g) = \frac{\textit{time budgeted for } \tau_{i.a:j.b} \textit{ on core } \ell \textit{ in frame } g}{C_{i:j}}.$$

This actually defines two variables for every pair of distinct jobs: one for which $i < j$ holds, and a second for which the opposite is true. For $i = j$, we define the variable only for $a = b$ and keep the same definition with the provision that a solo job is said to be "simultaneously co-scheduled" with itself.[5] *We refer to these variables as the x variables.*

▶ **Example 16.** Let the task system τ consist of $\tau_1 = (7.5, 10)$, $\tau_2 = (5, 20)$, $\tau_3 = (5, 20)$, $\tau_4 = (10, 20)$, and $\tau_5 = (20, 40)$. Furthermore, let $C_{1:2}$ and $C_{1:3} = 10$. Fig. 3 shows a possible schedule for τ on two cores (note that τ has total utilization of 2.25; it cannot be scheduled on two cores without SMT), along with the non-zero corresponding x variables. Frame borders are indicated by dashed lines; we have $f(1) = 10$ and $f(2) = 20$. The first two variables, $x(1.1, 2.1, 1, 1)$ and $x(2.1, 1.1, 1, 1)$, show that the first jobs of τ_1 and τ_2 are co-scheduled in frame 1 of core 1 for 10 time units, their joint cost. The last variable listed, $x(5.1, 5.1, 2, 2)$, shows that job 1 of τ_5 is scheduled in frame 2 of core 2 for half of its total cost. Note that the preemption of $\tau_{5.1}$ by the scheduler at time 20 is permitted, as our non-preemption requirement only applies to jobs using SMT.

[5] We do not propose scheduling a job with a second copy of itself; we overload the term simultaneously co-scheduled to avoid constantly addressing solo jobs as a special case.

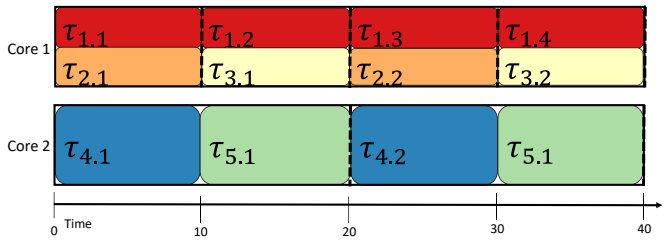


Figure 3 A possible schedule and corresponding x variables for the task system of Ex. 16.
x(1.1, 2.1, 1, 1)=x(2.1, 1.1, 1, 1)=1
x(1.2, 3.1, 1, 2)=x(3.1, 1.2, 1, 2)=1
x(1.3, 2.2, 1, 3)=x(2.2, 1.3, 1, 3)=1
x(1.4, 3.2, 1, 4)=x(3.2, 1.4, 1, 4)=1
x(4.1, 4.1, 2, 1)=1
x(5.1, 5.1, 2, 1)=0.5
x(4.2, 4.2, 2, 2)=1
x(5.1, 5.1, 2, 2)=0.5 .

We outline the constraints needed to build an optimization program that will produce a correct schedule for τ on platform π if such a schedule exists. Rather than giving the full program here, we outline the constraints needed to fulfill the conditions of Def. 14. The full program is available online [57].

Our first constraint is needed to address the existence of two variables for every pair of jobs. We require that

$$x(i.a, j.b, \ell, g) = x(j.b, i.a, \ell, g) \tag{4}$$

holds. In Fig. 3, we see the first eight variables listed as pairs for this reason. Our remaining constraints fulfill the requirements of Def. 14.

(i) All jobs are scheduled. To guarantee that all jobs released over the hyperperiod are scheduled, we require that for all jobs $\tau_{i.a}$ released within the first hyperperiod, the x variables corresponding to each job must sum to 1. Mathematically,[6]

$$\forall i, a \sum_{j=i}^{n} \sum_{b=1}^{\frac{H}{T_j}} \sum_{\ell=1}^{m} \sum_{g=1}^{\left\lfloor \frac{H}{f(\ell)} \right\rfloor} x(i.a, j.b, \ell, g) = 1.$$

In our example, τ_1 through τ_4 fulfill this requirement by having one variable valued at 1 for each job within the hyperperiod; τ_5 has two non-zero variables for its one job, which is scheduled in two different frames of core 2; each variable is valued at 0.5

(ii) Any non-preemptable job must be scheduled in exactly one frame. For $i \neq j$, we require that x be an integer variable, which can be seen with the variables corresponding to τ_1 through τ_3 in Ex. 16. While each job of τ_4 is also scheduled in a single frame, doing so is not necessary for a correct schedule, since τ_4 does not use SMT.

[6] Note that variables for which $i = j$ and $a \neq b$ do not exist per Def. 15. Here and elsewhere when $i = j$, $\sum_{b=1}^{\frac{H}{T_j}}$ would more properly written as $\sum_{b=a}^{a}$. We leave it as is for the sake of brevity.

(iii) **Every job completes in a frame that ends no later than its deadline.** Frame g of core ℓ ends at time $g \cdot f(\ell)$. We make the per-core frame size into a variable – $f(\ell)$ for all $\ell \leq m$ – and require that if a job is scheduled in a given frame, then the job's deadline may be no sooner than the end of the frame. In our optimization program, we express this rule using the following constraint:

$$\forall i.a, j.b, \ell, g, \lceil x(i.a, j.b, \ell, g) \rceil \cdot g \cdot f(\ell) \leq d(i.a, j.b).$$

If $\lceil x(i.a, j.b, \ell, g) \rceil = 1$ holds, then the restriction is true if and only if the frame's end time, $g \cdot f(\ell)$, is no more than the deadline. If $\lceil x(i.a, j.b, \ell, g) \rceil = 0$ holds, i.e., the job(s) is (are) not scheduled in frame g of core ℓ, making the frame size irrelevant, then the constraint is always true. While the ceiling operator is meaningless for non-preemptable co-scheduled jobs, which already have $x(i.a, j.b, \ell, g) \in \{0, 1\}$, it is necessary for the solo jobs to test whether *any* portion of the jobs is within frame g.
This constraint holds in Fig. 3; using the first jobs of τ_1 and τ_2 as an example, we have

$$\begin{aligned} \lceil x(1.1, 2.1, 1, 1) \rceil \cdot 1 \cdot f(1) &\leq d(1.1, 2.1) \\ 1 \cdot 1 \cdot 10 &\leq 10. \end{aligned}$$

Observe also that scheduling $\tau_{1.1}$ anywhere other than frame 1 of core 1, given that $f(2) = 20$, would violate this constraint. In addition, constraint (v) below guarantees that every job can execute for its scheduled time within each frame.

(iv) **No job executes in a frame that begins before its release.** Frame g of core ℓ begins at time $(g-1) \cdot f(\ell)$. Consequently, we require

$$\lceil x(i.a, j.b, \ell, g) \rceil \cdot r(i.a, j.b) \leq (g-1) f(\ell).$$

The logic here is similar to that in (iii) above; if $x(i.a, j.b, \ell, g) = 0$ holds, then the constraint will always be true. Otherwise, the job's release must fall no later than the beginning of the frame for the constraint to hold. In Fig. 3, using the third job of τ_1 and the second job of τ_2 as examples, we have

$$\begin{aligned} \lceil x(1.3, 2.2, 1, 2) \rceil \cdot r(1.3, 2.2) &\leq (3-1) \cdot f(1) \\ 1 \cdot 20 &\leq 2 \cdot 10. \end{aligned}$$

(v) **The total execution time scheduled in each frame is no greater than the frame size.** Each x variable requires that $x(i.a, j.b, \ell, g) \cdot C_{i:j}$ units of time be allocated to the corresponding job in the selected frame in order for the corresponding job or job pair to complete. Clearly, we must avoid overscheduling frames. The following constraint accomplishes that goal:

$$\forall\ l, g: \sum_{i=1}^{n} \sum_{a=1}^{\frac{H}{T_i}} \sum_{j=i}^{n} \sum_{b=1}^{\frac{H}{T_j}} x(i.a, j.b, \ell, g) \cdot C_{i:j} \leq f(\ell).$$

For example, $C_{1:2}$ and $f(1)$ are equal to 10 in Ex. 16, giving us

$$\begin{aligned} x(1.1, 2.1, 1, 1) \cdot C_{1:2} &\leq f(1) \\ 1 \cdot 10 &\leq 10 \end{aligned}$$

for frame 1 of core 1; note that all other variables connected to this frame, apart from $x(2.1, 1.1, 1, 1)$, are equal to 0.

(vi) No job executes in parallel with itself. For non-preemptable jobs, this rule holds automatically, since every job is scheduled in exactly one frame. For preemptable solo jobs, we add the constraint

$$\forall i, a, \ell \sum_{g=1}^{\left\lfloor \frac{H}{f(\ell)} \right\rfloor} x(i.a, i.a, \ell, g) \in \{0, 1\},$$

which states that for every solo job, either all or none of it must be on a given core. In Ex. 16, this constrains each job of τ_4 and τ_5 to a single core.

If a correct schedule exists, then an optimization program that follows the above constraints can find one. Despite the name, there is no objective function that needs to be optimized; if a set of decision variables that fulfills all restrictions exists, a correct schedule can be formed by making the scheduling choices corresponding to those variables. In our experiments, we found that when executing this program on a platform with 24 cores, most systems containing in the range of 20 tasks on 4 or 8 cores could be scheduled within a few seconds; we counted as a failure any system for which we had not found a solution within 60 seconds. Since solving the optimization program is done offline, this time is acceptable for many applications.

4 Timing Analysis

In this section, we present the timing-analysis portion of CERT-MT. While using the observed maximum for task costs, as we do here, is somewhat common practice, we are not aware of any prior work that considers exactly what the observed maximum is telling us, nor that addresses the fact that the observed maximum may itself be a random variable.

We use our results here to analyze our benchmark tests, presented in Sec. 5. All discussions of job execution costs in this section also apply to costs for simultaneously co-scheduled pairs of jobs. As mentioned in Sec. 2, we analyze execution times *as if* they were random. More specifically, we assume for now that for any given task, the execution times of individual jobs follow Premise 1 below. In Sec. 5 we show empirically that we can safely predict task behavior even without Premise 1 in most cases.

▶ **Premise 1.** The following properties hold for E_i and all $R_k \in R$: First, E_i and all values within R are drawn from the same probability distribution; second, individual R_k values are not dependent on k; and third, individual R_k values are not dependent on other values within R.

If we schedule our system using R_{max} as the task cost, how safe is that? More formally, consider Premise 2 below:

▶ **Premise 2.** Given task τ_i and trace R, assume that S_i^q is defined by $S_i^q = R_{max}$.

Combining Premise 2 and Def. 6 gives

$$\Pr(E_i \leq R_{max}) = q. \tag{5}$$

We want to determine the value of q. To do so, we need to consider both the relationship between E_i and R_{max} and that between $R_{max} = S_i^q$ and C_i^p for an arbitrary value of p. We make use of three basic rules of probability to determine q, given below without proof. Using these, rules, we give a lower bound for q in terms of p and $|R|$ in Lemma 20 below.

▶ **Proposition 17.** *Let events B_1 through B_v partition a probability space, and let A be an event belonging to the same probability space. The law of total probability states that the following holds:* $\Pr(A) = \sum_{i=1}^{v} \Pr(A|B_i) \cdot \Pr(B_i)$.

▶ **Proposition 18.** *Let A be a possible outcome of a repeated random trial. It follows for a series of trials, where each trial's result is independent from all previous results, that* $\Pr(A \text{ holds for some trial}) = 1 - \Pr(A \text{ holds for no trials})$ *holds.*

▶ **Proposition 19.** *Let A be a possible outcome of v repeated, independent trials. Then,* $\Pr(A \text{ holds for all trials}) = \Pr(A)^v$.

▶ **Lemma 20.** *Assume Premises 1 and 2 hold. Given a trace R and an arbitrary p associated with some value for C_i^p, the following holds:*

$$q \geq p \cdot \left(1 - p^{|R|}\right). \tag{6}$$

Note that since (6) holds for an arbitrary p, q is not a function of p.

Proof. The lemma is established by the following derivation:

$$
\begin{aligned}
& q \\
= \; & \{\text{by Exp. (5)}\} \\
& \Pr\left(E_i \leq R_{max}\right) \\
= \; & \{\text{by Prop. 17}\} \\
& \Pr(E_i \leq R_{max} | R_{max} \geq C_i^p) \cdot \Pr(R_{max} \geq C_i^p) + \Pr(E_i \leq R_{max} | R_{max} < C_i^p) \cdot \Pr(R_{max} < C_i^p) \\
\geq \; & \{\text{since } \Pr\left(E_i \leq R_{max} \mid R_{max} < C_i^p\right) \cdot \Pr\left(R_{max} < C_i^p\right) \geq 0\} \\
& \Pr\left(E_i \leq R_{max} \mid R_{max} \geq C_i^p\right) \cdot \Pr\left(R_{max} \geq C_i^p\right) \\
\geq \; & \{\text{since } \Pr\left(E_i \leq R_{max} \mid R_{max} \geq C_i^p\right) \geq \Pr(E_i \leq C_i^p)\} \\
& \Pr(E_i \leq C_i^p) \cdot \Pr\left(R_{max} \geq C_i^p\right) \\
= \; & \{\text{by Exp. (1)}\} \\
& p \cdot \Pr\left(R_{max} \geq C_i^p\right) \\
= \; & \{\text{by the definition of } R_{max} \text{ (Def. 5)}\} \\
& p \cdot \Pr(R_k \geq C_i^p \text{ holds for some } R_k \in R) \\
= \; & \{\text{by Prop. 18}\} \\
& p \cdot (1 - \Pr(R_k < C_i^p \text{ holds for all } R_k \in R)) \\
= \; & \{\text{by Prop. 19}\} \\
& p \cdot (1 - \Pr(R_k < C_i^p)^{|R|}) \\
= \; & \{\text{since } R_k \text{ terms are independent and from the same distribution as } E_i, \text{ per Pre. 1}\} \\
& p \cdot (1 - \Pr(E_i < C_i^p)^{|R|}) \\
\geq \; & \{\text{since } \Pr(E_i < C_i^p) \leq \Pr(E_i \leq C_i^p) \; \} \\
& p \cdot (1 - \Pr(E_i \leq C_i^p)^{|R|}) \\
= \; & \{\text{by Exp. (1)}\} \\
& p \cdot (1 - p^{|R|}).
\end{aligned}
$$

◀

We can now lower-bound q in terms of $|R|$ alone.

▶ **Theorem 21.** *Assume that Premises 1 and 2 hold. Then it follows that*

$$q \geq \left(\frac{1}{|R|+1}\right)^{\frac{1}{|R|}} \left(1 - \frac{1}{|R|+1}\right). \tag{7}$$

Proof. We first define the lower bound of q given in Exp. (6) as q^*, i.e. $q \geq q^*$ holds, where

$$q^* = p(1 - p^{|R|}).$$

Maximizing q^* will give a lower bound for q in terms of $|R|$ alone. To maximize q^*, we find the value of p for which the first derivative of q^* with respect to p equals 0 and the second derivative is negative. The first derivative is given by

$$\frac{dq^*}{dp} = 1 - (|R|+1) \cdot p^{|R|},$$

and the second by

$$\frac{d^2q^*}{dp^2} = -|R| \cdot (|R|+1) \cdot p^{|R|-1}.$$

Observe that $\frac{d^2q^*}{dp^2} < 0$ holds for all $p > 0$. Consequently, q^* is maximized when $\frac{dq^*}{dp} = 0$.

Solving $\frac{dq^*}{dp} = 0$ for p gives the value of p that maximizes q^*. Inserting this value into Exp. (6) in the place of q gives the result. ◀

▶ **Definition 22.** *We refer to the lower bound of Exp. (7) given trace size $|R|$ as $q_{b(|R|)}$.*

To attach some concrete values to (7), $|R| = 1000$ gives $q_{b(|R|)} = 0.992$ and $|R| = 10^5$ gives $q_{b(|R|)} = 0.9999$. Using these results, we can make comparisons between tasks with and without SMT based on maximum observed execution times. While we could have compared maximum observed times without the preceding proofs, determining $q_{b(|R|)}$ as we have allows us to meaningfully say that tasks with SMT are as safe as those without.

Violating Premise 1. The greatest potential shortfall of this approach is the reliance on Premise 1, which may not hold in practice. However, this obstacle is not unique to us; as mentioned in Sec. 2 under "Related works," EVT methods also must contend with data that may not be as independent as desired. In Sec. 5.1, we empirically test what happens when Premise 1 does not hold.

5 Experimental Results

In this section, we present our experimental results. The benchmark tests were conducted on an Intel Xeon Silver 4110 2.1 GHz (Skylake) CPU running Ubuntu 16.04.6. All code related to our experiments is available online [57]. We begin, in Sec. 5.1, by evaluating how well defining costs to be R_{max} works when Premise 1 may not hold. We also determine how many execution-time samples are needed to safely determine R_{max} within the context of our experiments. Using these sample counts, we then evaluate the benefits of allowing SMT in Sec. 5.2 by examining benchmark data involving individual tasks and task pairs. Finally, in Sec. 5.3, we evaluate these benefits more holistically on a system-wide basis via a schedulability study. The benchmark data discussed in Sec. 5.2 was used to inform parameter choices underlying this study.

Experimental studies are typically done within a framework that defines an "artificial world," and definitive conclusions can only really be drawn with respect to that "world" – further conclusions concerning the "real" world, though often interesting and relevant, are necessarily speculative. In our context, we need to be able to compare R_{max} values determined from relatively small traces to entire populations. Thus, in Secs. 5.1 and 5.2, we assume an artificial world in which 18 programs are of interest from the TACLeBench sequential benchmarks [23], which consist of functions commonly found in embedded and real-time systems. Furthermore, we assume that the entire population of execution times for the task in this world is given by a sequence of 100,000 job executions, which we denote as R^+. Note that, in the "real" world, we typically would not have the entire population of possible execution times – if we did, we would have no need for MBPTA. With this setup in place, our world thus consists of 18 solo tasks and 171 task pairs, each with 100,000 jobs or job pairs. In determining SMT execution costs in this world, we simultaneously co-scheduled task pairs per Def. 8. In all cases, we invalidated the cache between jobs or job pairs in determining such costs.

In the "real" world, appropriately dealing with task inputs in timing analysis is a complex issue. In our setting here, the initial "input" for each benchmark is hard-coded, but the program structure causes the inputs processed by each loop to vary, primarily due to the presence of non-resetting global variables. This aspect of the benchmarks allows for patterns similar to real-world applications where a task's input, and therefore its execution time, exhibits dependencies between one job and the next. For example, in many image-processing applications, the complexity of one job, and therefore the processing time needed, is predictive of the next. As our purpose here is to assess the viability of SMT, rather than fully addressing the issue of timing analysis in multicore systems, we believe these assumptions regarding input data are reasonable.

5.1 Timing-Analysis Assessment

So far, our timing-analysis has been theoretical; we have shown that if Premises 1 and 2 hold, then we can safely place a lower bound on the value of q, which we denote as $q_{b(|R|)}$ per Def. 22 ("b" denotes our theoretical bound). In practice, however, Premise 1 may not hold. What can we say about q in a more realistic setting? To avoid overloading q, we define an additional term for the value we actually compute given a trace and population.

▶ **Definition 23.** *Given a trace size $|R|$ and a population R^+, and assuming that every possible block of $|R|$ consecutive execution times taken from R^+ forms a trace and is equally likely to occur, let the result of computing q per Def. 6 be denoted $q_{c(|R|)}$ ("c" denotes a computed value of q).*

The way we have created R^+ and defined our possible traces means that Premise 1 will not typically hold; individual R_k values within each trace will tend to be dependent on both k and on other values within the same trace. However, if we find that $q_{c(|R|)} \geq q_{b(|R|)}$ holds, we then have evidence that building a system on the assumption that a proportion $q_{b(|R|)}$ of all tasks will have execution times no more than R_{max} is safe even without Premise 1. In our experiments, we use $|R| = 1000$ (i.e., we consider using 1,000 samples from the population of size 100,000), giving $q_{b(1000)} = 0.992$ using Exp. (7).

Results. When we calculated $q_{c(1000)}$ for each of our 18 solo tasks, we found a minimum $q_{c(1000)}$ of 0.997 and a mean and maximum of 0.999. We conclude that for these tasks in our experimental context, calculating costs via R_{max} is safe even without Premise 1. Our

$q_{c(1000)}$ values are closer to 0.999 than to $q_{b(1000)}$; the former is the value we would expect to see given that $R_{max} \approx C_i^{0.999}$. Our $q_{c(1000)}$ values for all solo jobs, along with additional summary statistics, are available in an online appendix [57]. For our 171 task pairs using SMT, the mean $q_{c(1000)}$ across all pairs was found to be 0.998. Further highlights are given in Table 1. Full results are in [57].

Cases where $\mathbf{q_{c(1000)} < q_{b(1000)}}$. Of the 171 task pairs, five had $q_{c(1000)} < q_{b(1000)}$, but if we exclude pairs for which C_i and C_j (defined as the R_{max} for the first 1,000 jobs of each solo task) differ by a factor of more than 10, we can eliminate these five problem pairs. In Sec. 5.2, we will see that there are additional reasons to not consider pairs with dramatically different solo execution times, making this exclusion an appealing option.

An alternative solution is to increase our trace size until we find a new value for $|R|$ such that $q_{c(|R|)} \geq q_{b(1000)}$. When we did this, our new $|R|$ values ranged from 1,250 to 1,500, in all cases giving $q_{c(|R|)} > 0.992$. An example can be seen in Table 1; the pair of tasks "cjpeg_wrbmp" and "susan" have $q_{c(1000)} = 0.987$, but if we inflate the trace size by a factor of 1.5 (column "inf."), we then have $q_{c(1500)} = 0.992$ (column "$q_{(inf.|R|)}$"). Note that inflation factors are not applicable for task pairs where $q_{c(1000)} \geq q_{b(1000)}$ holds. Within the context of our experiments, this result suggests that even when SMT is challenging from a timing-analysis perspective, reliable timings can be obtained by using a trace size that is a slightly larger than the trace size that would be needed for a given level of safety if Premise 1 held. We plan to investigate this aspect of our results further in future work, with a particular emphasis on what inflation factors can be expected given a wider range of tasks and R_{max} values.

Timing-analysis conclusions. Before moving on, we recap our conclusions from this subsection. In the experiments discussed here, we found that $q_{c(1000)} \geq q_{b(1000)}$ held perfectly, even without Premise 1, when SMT was not in play. When SMT was used, $q_{c(1000)} \geq q_{b(1000)}$ held most of the time, or all of the time if do not allow SMT to be used when C_i and C_j differ by more than a factor of 10.

The seemingly obvious solution is to use SMT only for cases where $q_{c(1000)} \geq q_{b(1000)}$ holds. The problem is that to know whether the inequality holds, we need either an assurance that Premise 1 holds or the ability to compare multiple traces to the population. In real applications, we would not have the population available. For this reason, the implementation of our timing-analysis method should be done in consultation with domain experts who could give assurances that a trace is large enough to capture relevant execution-time variations.

This approach may seem unsatisfactory, but it is similar to formal system verification under a fault hypothesis. In a safety-critical context, such as avionics, software properties are demonstrated to hold subject to a given fault hypothesis – informally, a list of possibilities that should not happen. For example, an aviation software fault hypothesis might include "foreign objects will not enter the engines." If this condition fails – perhaps a flock of geese flies into the engines – then correctness is no longer guaranteed. A fault hypothesis will be based on expert input, but it cannot be mathematically guaranteed. Selecting a trace size that will allow at least the first point of Premise 1 to hold is essentially another element of a fault hypothesis. In this case we would be claiming that $|R|$ is large enough to capture the task's overall behavior. This concern is not unique to us, but applies to any MBPTA method. Even for static analysis, the results hinge on the assumption that all relevant factors were adequately accounted for.

Table 1 Task pairs with min., median, and max. $q_{c(1000)}$ values.

benchmark 1	benchmark 2	$q_{c(1000)}$	inf.	$q_{c(inf.\|R\|)}$
cjpeg_wrbmp	susan	0.987	1.5	0.992
adpcm_dec	rijndael_enc	0.999	NA	NA
dijksta	gsm_enc	0.999	NA	NA

5.2 Execution Times with and without SMT

We now consider how execution times with SMT compare to those without. We need this information to make a realistic assessment of the potential benefits of using SMT. In Sec. 5.1, we saw that given our populations, trace sizes of 1,000 are safe for tasks without SMT and trace sizes of 1,500 are safe for tasks with SMT. With that in mind, our results in this subsection are based on the first 1,000 execution times for solo tasks and the first 1,500 execution times for pairs. This is a conservative approach; while we found only five task pairs for which inflating $|R|$ was necessary, we do so here for all pairs.

For each pair of benchmark tasks, we record their *multithreading score.*

▶ **Definition 24.** *If $\tau_{i:j}$ is a task pair for which $C_i \geq C_j$ holds, then the* multithreading score *$M_{i:j}$ satisfies the following:*

$$C_{i:j} = C_i + M_{i:j} \cdot C_j.$$

When τ_i and τ_j are co-scheduled, we are essentially hiding the execution requirement for τ_j at the cost of increasing the execution time needed for τ_i. $M_{i:j}$ thus gives how much greater $C_{i:j}$ is than C_i for every unit of C_j, i.e., the cost to τ_i, per unit of τ_j, of "hiding" τ_j.

If $M_{i:j} \geq 1$ holds, then there is no benefit to pairing τ_i and τ_j together, since SMT would not succeed in "hiding" τ_j at all. If $M_{i:j} < 1$ holds, then pairing jobs of the two tasks is potentially beneficial, with lower values indicating greater benefit. If $M_{i:j} < 0$ holds, then $\tau_{i:j}$ actually requires less measured time to execute than τ_i alone.

We found that $M_{i:j} > 1$ was frequently the case for pairs where $\frac{C_i}{C_j} > 10$ holds. For this reason, along with the difficulty in using R_{max} for timing analysis in those cases described in Sec. 5.1, we decided that using SMT in those cases is not advisable. For the remaining pairs, where $\frac{C_i}{C_j} \leq 10$ holds, we summarize our results via a histogram in Fig. 4. The remainder of our discussion considers only these remaining pairs. Again, more detailed results are available in [57]. We did see five negative values, ranging from -0.003 to -0.532. For all of these values, comparing the traces for the component solo tasks to the larger population showed unexpectedly high R_{max} values. Essentially, we overestimated both τ_i and τ_j individually, thereby distorting the value of $M_{i:j}$; our values were not unsafe.

Even if we ignore the negative values, our findings indicate that SMT can have some benefit in almost all pairs that have similar solo costs for τ_i and τ_j. We make further use of these findings, and explore their implications for schedulability, in Sec. 5.3.

5.3 Schedulability Study

In this subsection, we turn our attention to assessing the benefits of allowing SMT from a system-wide perspective. To provide such an assessment, we conducted a schedulability study involving CERT-MT.

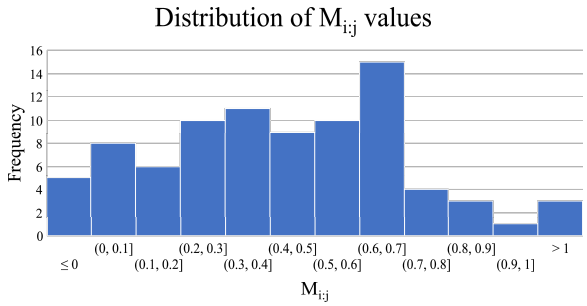


Figure 4 Distribution of $M_{i:j}$ values. The distribution has a median of 0.42 and values less than one have a mean of 0.40. Three values are greater than 1.0.

Generating task sets. We examined 104 scheduling scenarios, with each scenario defined by a core count, per-task utilization range, and a model for SMT interaction. The last factor is described in detail below; the others require only a brief explanation. We considered core counts of four and eight and per-task utilizations drawn from four uniform ranges: $(0, 0.4)$ (*low*), $(0.3, 0.7)$ (*medium*), $(0.6, 1)$ (*high*), and $(0, 1)$ (*wide*). Each task was created by selecting a utilization from the appropriate distribution and a period from the set $\{10, 20, 40, 80\}$, with all periods having equal probability. A solo task execution cost was then assigned as a function of utilization and period. For each scenario, we determined schedulability ratios (i.e., the percentage of schedulable task sets) for task systems ranging in total utilization from $\frac{3m}{4}$ to $2m$. Recall that m gives the core count, and consequently the maximum schedulable utilization when SMT is not used.

For each scenario, the corresponding schedulability ratios are summarized in one graph. Each data point in one of these graphs represents the fraction of approximately 50 task systems that could be deemed as schedulable. The total set of graphs took over 2 CPU years to compute on a 6,500-CPU research cluster.

Modeling pair costs. A key aspect of our study is that of modeling the cost of job pairs executing with SMT. Our goal in modeling such costs was not to reproduce any system's behavior exactly, but to be able to draw broad conclusions about the value of allowing SMT without relying on one specific model for task interactions.

In Fig. 4, we see that most $M_{i:j}$ values fall between 0.1 and 0.8. To capture the possibility of $M_{i:j} \geq 1$, we first gave each job pair either a 0.0, 0.1, or 0.2 probability of having $M_{i:j} \geq 1$. We refer to this value as the *split*, i.e., a split of 0.1 indicates that each task pair has a 10% chance of being declared unsuitable for SMT. This particular split value closely reflects our observed results discussed previously. If a task pair was not selected to have $M_{i:j} \geq 1$, then we determined its $M_{i:j}$ value based on one of three normal distributions – (0.45, 0.12), (0.6, 0.07), or (0.45, .06) – or one of three uniform distributions – (0.1, 0.8), (0.4, 0.8), or (0.27, 0.63). All of the normal distributions were truncated, with any negative value produced replaced by 0.01. Of these distributions, the first uniform and first normal distributions most closely match our observed results, which had a median of 0.43 and a mean of 0.40 excluding cases where $M_{i:j} > 1$. We exclude those cases from our mean because any pair where $M_{i:j} > 1$ holds is unsuitable for threading; the exact value of $M_{i:j}$ will not impact

the decision. The second distribution of each type gives a higher (i.e., more pessimistic) mean, and the third of each type has the same mean as the first, but a smaller standard deviation (resp., range) in the normal (resp., uniform) case. All of these distributions add some pessimism to Fig. 4 by eliminating or reducing the probability of $M_{i:j} < 0.1$.

In total, we utilized 18 combinations of distributions in determining task-pair costs: three probabilities for $M_{i:j} \geq 1$, each combined with each of three uniform and three normal distributions. After determining $M_{i:j}$, we calculated $C_{i:j}$ per Def. 24. For task pairs where solo costs differ by a factor exceeding 10, we did not allow SMT. For our four-core scenarios, we tested every possible combination of per-task utilizations and $M_{i:j}$ distributions, yielding 72 scenarios. For our eight-core scenarios, we were more selective, considering 32 scenarios.

Creating and evaluating schedules. To determine whether a task system was schedulable, we attempted to create a schedule that complied with the conditions given in Sec. 3 for each system.[7] The graphs show the proportion of systems that are schedulable at each total utilization. We summarize CERT-MT's overall performance for each graph by recording its *relative schedulable area (RSA)*, defined as the area under the schedulability curve divided by the core count m. In calculating RSAs, we assumed that the schedulability ratio is constant between total utilization 0.0 and $\frac{3m}{4}$, which is the smallest utilization we tested in each scenario. This assumption results in RSAs being understated in some scenarios.

An ideal (e.g., fluid) scheduler, not using SMT, that can preempt and migrate jobs arbitrarily would have an RSA of 1.0; it could schedule all task systems with total utilization at most m and no task systems with greater utilization. RSAs for practical hard real-time schedulers would typically be less than 1.0, sometimes dramatically so [15].

Schedulability graphs. Our full set of graphs is included in an online appendix [57]. Here we present a small selection of our results, along with some observations on general trends we observed. In Figs. 5 – 8 we show, respectively, our best, median, and worst four-core results, along with our best eight-core results. Generally, CERT-MT's performance in the eight-core scenarios was not as good as in the four-core scenarios, for reasons discussed below.

▶ **Observation 1.** *At its best, CERT-MT proved capable of increasing RSAs by a factor of 1.4 or more. This ability is demonstrated by Fig. 5. Of our 72 four-core graphs, 26 have RSAs of at least 1.3, and 45 have RSAs are greater than 1.0.*

▶ **Observation 2.** *The performance of CERT-MT tended to decrease as the job count increased. For example, Fig. 7 uses light per-task utilization, meaning there are more tasks for a given total utilization. The poorer performance of CERT-MT for high job counts was not due to any ill effects from SMT but rather to the time required to produce a schedule. Given the magnitude of our study, it was necessary to enforce a time limit on our optimization program. If this limit was reached for a particular task set, then it was deemed as unschedulable. We used a 60-second limit. This observation also explains the relatively poor performance we saw in the eight-core scenarios. We counter this observation, however, by noting that when testing a specific task system, a much higher time limit could be used. Moreover, it is possible that a more efficient schedule-construction method could be devised.*

[7] We used Gurobi Optimizer, a commercial optimization programming solver with free academic licensing.

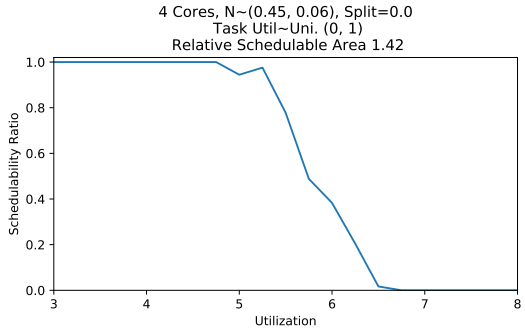


Figure 5 The best four-core scenario.

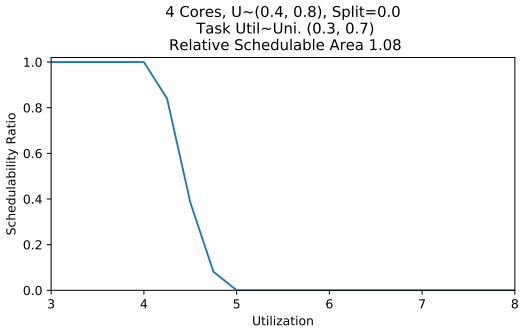


Figure 6 The median four-core scenario.

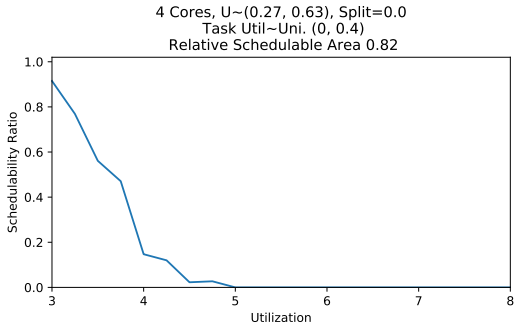


Figure 7 The worst four-core scenario.

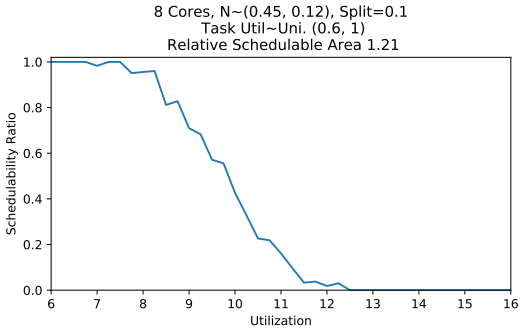


Figure 8 The best eight-core scenario.

▶ **Observation 3.** *CERT-MT performed best when the majority of $M_{i:j}$ values fell within a narrow range. For example, the standard deviation of such values is the smallest in the most successful scenario (depicted in Fig. 5). This observation is also connected to the time limits we imposed on our optimization program; a wider range of $M_{i:j}$ values gives this program more "choices" to consider, which takes more time.*

▶ **Observation 4.** *The mean of $M_{i:j}$ had surprisingly little effect on overall performance. The graphs in Figs. 5 and 7 have distributions that produce identical means for $M_{i:j}$.*

Improving schedulability. The schedulability study we conducted suggests that employing SMT can potentially enable substantial schedulability gains in hard real-time systems. Additionally, the observations above hint at several ways of further increasing schedulability beyond what we have seen. First, a larger schedule-construction timeout value could be used, as noted already. Second, for larger systems, it may be more effective to divide tasks into clusters scheduled separately. Third, given that the standard deviation of $M_{i:j}$ seems to be more important for schedulability than its mean, it might be possible, counter-intuitively, to improve schedulability by artificially increasing the lower $M_{i:j}$ values to decrease their range.

6 Conclusion

In industry today there is interest in using SMT to increase the capacity of safety-critical systems, as evidenced by the FAA inquiries noted earlier [55]. In this paper, we have shown that when SMT-enabled jobs are scheduled with care, producing a safe, measurement-based

timing analysis is comparable in difficulty to doing the same for jobs not using SMT. Within the context of our schedulability study, we demonstrated that allowing SMT can increase system capacity by up to 40%. We have also highlighted a subtlety of measurement-based timing analysis – any estimate based on random measured data is itself a random variable – that is not always emphasized, and that could compromise safety, if not accounted for.

Due to space constraints, there are many topics we have not been able to address that we plan to address in the future. These include: the impacts of (well-studied) sources of cross-core interference in multicore systems when SMT is allowed; the selection of appropriate task inputs in measurement-based timing analysis; the impacts of occasional deadline misses on overall system safety; and refined methods for constructing schedules under CERT-MT. As to the question posed in this paper's title, the evidence provided in this paper suggests that SMT can indeed be both safe and beneficial for hard real-time systems.

References

1 J. Abella, E. Quiñones, F. Wartel, T. Vardanega, and F. J. Cazorla. Heart of gold: Making the improbable happen to increase confidence in MBPTA. In *ECRTS 2014*, pages 255–265, 2014.

2 S. Altmeyer, R. Douma, W. Lunniss, and R. I. Davis. Outstanding paper: Evaluation of cache partitioning for hard real-time systems. In *ECRTS 2014*, pages 15–26. IEEE, 2014.

3 J. H. Anderson, S. Baruah, and B. Brandenburg. Multicore operating-system support for mixed criticality. In *Proceedings of the Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, volume 4, page 7. Euromicro, 2009.

4 B. Andersson, H. Kim, D. De Niz, M. Klein, R. Rajkumar, and J. Lehoczky. Schedulability analysis of tasks with corunner-dependent execution times. *ACM Trans. Embed. Comput. Syst.*, 17(3):71:1–71:29, May 2018.

5 T. P. Baker and A. Shaw. The cyclic executive model and ada. *Real-Time Systems*, 1(1):7–25, June 1989.

6 A. A. Balkema and L. De Haan. Residual life time at great age. *The Annals of Probability*, pages 792–804, 1974.

7 P. Benedicte, L. Kosmidis, E. Quinones, J. Abella, and F. J. Cazorla. A confidence assessment of wcet estimates for software time randomized caches. In *INDIN*, pages 90–97, 2016.

8 P. Benedicte, L. Kosmidis, E. Quinones, J. Abella, and F. J. Cazorla. Modelling the confidence of timing analysis for time randomised caches. In *SIES*, 2016.

9 B. D. Bui, M. Caccamo, L. Sha, and J. Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. In *RTCSA 2008*, pages 101–110. IEEE, 2008.

10 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/`.

11 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.

12 A. Burns and S. Baruah. Migrating mixed criticality tasks within a cyclic executive framework. In J. Blieberger and M. Bader, editors, *Reliable Software Technologies – Ada-Europe 2017*, pages 203–216, Cham, 2017. Springer International Publishing.

13 A. Burns and S. Edgar. Predicting computation time for advanced processor architectures. In *ECRTS 2000*, pages 89–96, February 2000.

14 A. Burns, T. Fleming, and S. Baruah. Cyclic executives, multi-core platforms and mixed criticality applications. In *ECRTS 2015*, pages 3–12, July 2015.

15 J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. H. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.

16 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.

17 F. J. Cazorla, L. Kosmidis, E. Mezzetti, C. Hernandez, J. Abella, and T. Vardanega. Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey. *ACM Comput. Surv.*, 52(1):14:1–14:35, February 2019.

18 M. Chisholm, N. Kim, S. Tang, N. Otterness, J. H. Anderson, F. D. Smith, and D. E. Porter. Supporting mode changes while providing hardware isolation in mixed-criticality multicore systems. In *RTNS 2017*, pages 58–67. ACM, 2017.

19 M. Chisholm, B. C. Ward, N. Kim, and J. H. Anderson. Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *RTSS 2015*, pages 305–316. IEEE, 2015.

20 R. Davis and L. Cucu-Grosjean. A survey of probabilistic timing analysis techniques for real-time systems. *Leibniz Transactions on Embedded Systems*, 6(1):03–1–03:60, 2019.

21 C. Deutschbein, T. Fleming, A. Burns, and S. Baruah. Multi-core cyclic executives for safety-critical systems. *Science of Computer Programming*, 172:102–116, 2019.

22 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.

23 H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *WCET 2016*, volume 55, pages 2:1–2:10, 2016.

24 R. A. Fisher and L. H. C. Tippett. Limiting forms of the frequency distribution of the largest or smallest member of a sample. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 24, pages 180–190. Cambridge University Press, 1928.

25 A. Fog. The microarchitecture of Intel, AMD, and VIA CPUs: an optimization guide for assembly programmers and compiler makers, 2018. Available at `https://www.agner.org/optimize/microarchitecture.pdf`.

26 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.

27 T. Gomes, S. Pinto, P. Garcia, and A. Tavares. RT-SHADOWS: Real-time system hardware for agnostic and deterministic OSes within softcore. In *ETFA 2015*, pages 1–4, September 2015.

28 F. Guet, L. Santinelli, and J. Morio. On the Reliability of the Probabilistic Worst-Case Execution Time Estimates. In *ERTS 2016*, Toulouse, France, January 2016.

29 D. Guo and R. Pellizzoni. A requests bundling DRAM controller for mixed-criticality systems. In *RTAS 2017*, pages 247–258. IEEE, 2017.

30 M. Hassan, H. Patel, and R. Pellizzoni. A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In *RTAS 2015*, pages 307–316. IEEE, 2015.

31 J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson, and D. M. Johnson. RTOS support for multicore mixed-criticality systems. In *RTAS 2012*, pages 197–208, April 2012.

32 C. Hernandez, J. Abella, A. Gianarro, J. Andersson, and F. J. Cazorla. Random modulo: a new processor cache design for real-time critical systems. In *Proceedings of the 53rd Annual Design Automation Conference*, page 29. ACM, 2016.

33 T. Hsing. On tail index estimation using dependent data. *The Annals of Statistics*, pages 1547–1569, 1991.

34 W. Huang, J. Lin, Z. Zhang, and J.M. Chang. Performance characterization of Java applications on SMT processors. In *ISPASS 2005.*, pages 102–111, March 2005.

35 R. Jain, C. J. Hughes, and S. V. Adve. Soft real-time scheduling on simultaneous multithreaded processors. In *RTSS 2002*, pages 134–145, 2002.

36 S. Jiménez Gil, I. Bate, G. Lima, L. Santinelli, A. Gogonel, and L. Cucu-Grosjean. Open challenges for probabilistic measurement-based worst-case execution time. *IEEE Embedded Systems Letters*, 9(3):69–72, September 2017.

37 S. Kato, H. Kobayashi, and N. Yamasaki. U-link scheduling: bounding execution time of real-time tasks with multi-case execution time on SMT processors. In *RTCSA 2005*, pages 193–197, August 2005.

38 S. Kato and N. Yamasaki. Extended u-link scheduling to increase the execution efficiency for SMT real-time systems. In *RTCSA 2006*, pages 373–377, August 2006.

39 J. Kim, M. Yoon, R. Bradford, and L. Sha. Integrated modular avionics (IMA) partition scheduling with conflict-free I/O for multicore avionics systems. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 321–331, July 2014.

40 N. Kim. *Combining Hardware Management with Mixed-Criticality Provisioning in Multicore Real-Time Systems*. PhD thesis, UNC Chapel Hill, 2019. URL: `https://www.cs.unc.edu/~anderson/diss/namhoondiss.pdf`.

41 D. B. Kirk. SMART (strategic memory allocation for real-time) cache design. In *RTSS 1989*, pages 229–237. IEEE, 1989.

42 L. Kosmidis, J. Abella, E. Quiñones, and F. 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.

43 L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla. Efficient cache designs for probabilistically analysable real-time systems. *IEEE Transactions on Computers*, 63(12):2998–3011, 2013.

44 M. R. Leadbetter, G. Lindgren, and H. Rootzén. Conditions for the convergence in distribution of maxima of stationary normal processes. *Stochastic Processes and their Applications*, 8(2):131–139, 1978.

45 G. Lima and I. Bate. Valid application of EVT in timing analysis by randomising execution time measurements. In *RTAS 2017*, pages 187–198. IEEE, 2017.

46 G. Lima, D. Dias, and E. Barros. Extreme value theory for estimating task execution time bounds: A careful look. In *ECRTS 2016*, pages 200–211. IEEE, 2016.

47 J. W. S. Liu. *Real-Time Systems*. Prentice Hall, New York, NY, USA, 2000.

48 S. Lo, K. Lam, and T. Kuo. Real-time task scheduling for SMT systems. In *RTCSA 2005*, pages 5–10, August 2005.

49 R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *RTAS 2013*, pages 45–54, April 2013.

50 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.

51 S. Milutinovic, I. Abella, J.and Agirre, M. Azkarate-Askasua, E. Mezzetti, T. Vardanega, and F. J. Cazorla. Software time reliability in the presence of cache memories. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 233–249. Springer, 2017.

52 S. Milutinovic, J. Abella, and F. J. Cazorla. Modelling probabilistic cache representativeness in the presence of arbitrary access patterns. In *2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 142–149, May 2016.

53 J. Mische, S. Uhrig, F. Kluge, and T. Ungerer. Using SMT to hide context switch times of large real-time tasksets. In *RTAS 2010*, pages 255–264, August 2010.

54 D. Muench, M. Paulitsch, and A. Herkersdorf. Temporal separation for hardware-based I/O virtualization for mixed-criticality embedded real-time systems using PCIe SR-IOV. In *ARCS 2014; 2014 Workshop Proceedings on Architecture of Computing Systems*, pages 1–7, February 2014.

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

56 S. Osborne and J. H. Anderson. Work in progress: Combining real time and multithreading. In *RTSS 2019*, pages 139–142, December 2018.

57 S. Osborne and J. H. Anderson. Simultaneous multithreading and hard real time: Can it be safe? (longer version with additonal material), 2020. Available at `http://jamesanderson.web.unc.edu/papers/`.

58 S. Osborne, J. Bakita, and J. H. Anderson. Simultaneous multithreading applied to real time. In *ECRTS 2019*, July 2019.

59 R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha. Coscheduling of CPU and I/O transactions in COTS-based embedded systems. In *RTSS*, 2008.

60 J. Pickands III. Statistical inference using extreme order statistics. *The Annals of Statistics*, 3(1):119–131, 1975.

61 L. Santinelli, F. Guet, and J. Morio. Revising measurement-based probabilistic timing analysis. In *RTAS 2017*, pages 199–208, 2017.

62 L. Santinelli, J. Morio, G. Dufour, and D. Jacquemart. On the sustainability of the extreme value theory for WCET estimation. In *14th International Workshop on Worst-Case Execution Time Analysis*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

63 G. N. Seetanadi, J. Camara, L. Almeida, K. Arzen, and M. Maggio. Event-driven bandwidth allocation with formal guarantees for camera networks. In *RTSS 2017*, pages 243–254, December 2017.

64 A. Snavely and D. M. Tullsen. Symbiotic job scheduling for a simultaneous multithreaded processor. In *ASPLOS 2000*, ASPLOS IX, pages 234–244, New York, NY, USA, 2000. ACM.

65 K. Suito, K. Fujii, H. Matsutani, and N. Yamasaki. Dependable responsive multithreaded processor for distributed real-time systems. In *2012 IEEE COOL Chips XV*, pages 1–3, April 2012.

66 N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *PACT*, pages 26–35, Washington, DC, USA, 2003. IEEE Computer Society.

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

68 P. K. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *RTAS 2016*, pages 1–12, April 2016.

69 B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson. Outstanding paper award: Making shared caches more predictable on multicore platforms. In *ECRTS 2013*, pages 157–167, July 2013.

70 M. Xu, L. T. X. Phan, H. Choi, and I. Lee. Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation. In *RTAS 2016*, pages 1–12, April 2016.

71 H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *RTAS 2014*, pages 155–166, April 2014.

72 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 *RTAS*, pages 55–64, April 2013.

73 M. Zimmer, D. Broman, C. Shaver, and E. A. Lee. FlexPRET: A processor platform for mixed-criticality systems. In *RTAS 2014*, pages 101–110, April 2014.

Tracing Hardware Monitors in the GR712RC Multicore Platform: Challenges and Lessons Learnt from a Space Case Study

Xavier Palomo
Barcelona Supercomputing Center, Spain
xavier.palomo@bsc.es

Mikel Fernandez
Barcelona Supercomputing Center, Spain
mikel.fernandez@bsc.es

Sylvain Girbal
Thales Research, Palaiseau, France
sylvain.girbal@thalesgroup.com

Enrico Mezzetti
Barcelona Supercomputing Center, Spain
enrico.mezzetti@bsc.es

Jaume Abella
Barcelona Supercomputing Center, Spain
jaume.abella@bsc.es

Francisco J. Cazorla
Barcelona Supercomputing Center, Spain
francisco.cazorla@bsc.es

Laurent Rioux
Thales Research, Palaiseau, France
laurent.rioux@thalesgroup.com

Abstract

The demand for increased computing performance is driving industry in critical-embedded systems (CES) domains, e.g. space, towards the use of multicores processors. Multicores, however, pose several challenges that must be addressed before their safe adoption in critical embedded domains. One of the prominent challenges is software timing analysis, a fundamental step in the verification and validation process. Monitoring and profiling solutions, traditionally used for debugging and optimization, are increasingly exploited for software timing in multicores. In particular, hardware event monitors related to requests to shared hardware resources are building block to assess and restraining multicore interference. Modern timing analysis techniques build on event monitors to track and control the contention tasks can generate each other in a multicore platform. In this paper we look into the hardware profiling problem from an industrial perspective and address both methodological and practical problems when monitoring a multicore application. We assess pros and cons of several profiling and tracing solutions, showing that several aspects need to be taken into account while considering the appropriate mechanism to collect and extract the profiling information from a multicore COTS platform. We address the profiling problem on a representative COTS platform for the aerospace domain to find that the availability of directly-accessible hardware counters is not a given, and it may be necessary to the develop specific tools that capture the needs of both the user's and the timing analysis technique requirements. We report challenges in developing an event monitor tracing tool that works for bare-metal and RTEMS configurations and show the accuracy of the developed tool-set in profiling a real aerospace application. We also show how the profiling tools can be exploited, together with handcrafted benchmarks, to characterize the application behavior in terms of multicore timing interference.

2012 ACM Subject Classification Computer systems organization → Multicore architectures

Keywords and phrases Multicore Contention, Timing interference, Hardware Event Counters, PMC

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.15

Funding This work has been partially supported by a collaboration agreement between Thales Research and the Barcelona Supercomputing Center, and the European Research Council (ERC) under the EU's Horizon 2020 research and innovation programme (grant agreement No. 772773). MINECO partially supported Jaume Abella under Ramon y Cajal postdoctoral fellowship (RYC-2013-14717).

32nd Euromicro Conference on Real-Time Systems (ECRTS 2020).
Editor: Marcus Völp; Article No. 15; pp. 15:1–15:25

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

In the critical embedded systems (CES) industry domain, applications are usually characterized by stringent real-time constraints, making time predictability a major concern with regards to the regulation / certification standards [28, 29, 40, 16]. In this view, the ability to monitor the behavior of the software functionalities is fundamental to promptly intercept both functional and timing misbehaviors, and provide controlled degraded service in case of failures. The shift from single-core COTS (component off-the-shelf) to multicore COTS is appealing for the industry, as it fits with the exponential growth in terms of performance requirements while providing an excellent compromise in terms of size, weight and power (SWaP) [4]. Indeed, this transition is already happening and multicore-based platforms are slowly but inescapably becoming the de-facto computing solution for supporting such functionalities, and aerospace is not an exception [15, 49]. On the downside, however, multicore COTS hardware shared resources complicate software timing analysis and validation, a mandatory step in critical embedded systems development. Industry is facing a trade-off between performance and predictability [31, 35]. Classical analysis and modeling tools [50, 39, 26] relying only on timing analysis are not currently able to provide an efficient solution for the computation of the Worst Case Execution Time (WCET) for multi-core systems [51, 42]. The main challenges posed by multicore COTS analysis arise from the side effects of parallel execution. Multicore architectures provide more performance by allowing the concurrent execution of several threads. However, these threads are competing to use the shared hardware resources of the processor architecture, causing potential conflicts between concurrent accesses to the same hardware component. At hardware level, these conflicting accesses are arbitrated, introducing inter-thread jitters defined as *multicore timing interference* [24]. The maximum impact of timing interference on the execution time of real-time applications has been quantified to be quite large in several studies [6, 36], with an order of magnitude of 20x compared to a single-core execution for several 8-core architectures. The problem of multicore interference has been explicitly addressed by safety-critical industry standards, which already defined ad-hoc verification requirements for the adoption of multi-core processors, forcing us to clearly identify all *interference channels* [17], to either ensure robust partitioning (guaranteeing both space and time isolation between applications), or to upper bound the timing interference.

Modern contention-aware solutions for multicore timing analysis, exemplified by [37, 23], track and control hardware event counters. The overall timing analysis framework usually builds on the (contention-free) time in isolation of the task under analysis, τ_a, referred to as C_a^{isol} and a bound to the delay that τ_a can suffer Δ_a^{cont}, to derive the worst-case execution time in multicore C_a^{muc}. The term Δ_a^{cont} is typically computed by exploiting the number of requests each task τ_x performs to each shared hardware resource and the worst-case contention delay each request can suffer (L_{max}). Both pieces of information are empirically derived via event monitors. Each technique proposes a different trade-off in terms of performance and time predictability by enforcing, for example, usage quotas. Most approaches require the ability to monitor accesses to the shared hardware resources. It follows that *monitoring the hardware behavior such as accesses to the hardware components has therefore become instrumental to multicore timing analysis*, far beyond their initial intended usage for debugging and regular timing monitoring purposes [34].

The identification of the most adequate tracing solution depends on several factors including the multicore timing analysis approach used, verification requirements from safety standards, induced costs, as well as the features provided by existing event counters and tracing solution. We provide three illustrative examples. First, mainstream processors

from the consumer electronic market provide powerful statistic units, also referred to as performance monitoring units or PMUs. However, this is not always the case for processors commonly used in the embedded domain due to cost-reduction reasons. To make things more complex for the end user, powerful (and expensive) tracing solutions [1] carry specialized (debug) cables and hardware modules, which are not trivial to use by software developers. Second, for some particular hardware events, PMUs may not provide a specific counter. For instance, authors in [11] highlight the lack of dedicated counters for loads and stores misses in the last level (L2) cache in a LEON4-based multicore processor. As these counters are considered necessary for timing characterization, authors resort to derive upper bounds to their values by conservatively combining information from other event counters. Arguably, tracing analysis can help identifying loads and stores causing dirty misses, helping to tight contention bounds. As a third example, PMUs are rarely able to capture on-chip controller (e.g. DMA) access counts. This is reported in [21] where access counts to the different on-chip memories (pflash, dflash, and lmu) in an Infineon AURIX TC27x are indirectly derived by parsing the address accessed by each load/store operation. Overall, how to obtain the necessary event counter information in an embedded processor, is a real industrial concern, and a fundamental enabler for the adoption of multicore processors for industrial products.

As part of a collaborative effort between a technology provider research center and an aerospace industry, we have been exploring the problem of extracting event monitoring information for the analysis of an aerospace application touching both methodological and practical aspects. In this paper we report on our experience along several aspects:

First, we analyze the trade-offs of different tracing solutions with emphasis on adapting to the specific requirements of the end user. We present this trade-off as a taxonomy in this work with the goal that it helps researchers and practitioners in the future in the selection of a tracing solution that better fits their needs.

Second, we enter into more practical aspects and address the challenge of profiling access counts to the different shared resources in the GR712RC multicore processor [46], deployed in space systems, whose implementation provides no event monitoring. We show how we successfully exploited the debug support unit to obtain the necessary information on which to build a multicore timing analysis solution. We discuss the main limitations of our approach and show their impact on timing analysis accuracy. We show that our approach successfully identifies the number of instruction and data cache misses, their type (load or store, line fills), and their target (on-chip SRAM, off-chip SRAM, SDRAM, I/Os), providing information akin to performance monitoring counters. We also provide evidence that our approach can be deployed equally to bare-metal (BM) (i.e. no operating system) and RTOS-based systems by showing that we obtain consistent results on BM and RTEMS, as a reference RTOS in the space domain.

Finally, we assess the applicability our profiling solution on a real space application for characterization and timing analysis of representative software functions of the space domain.

The rest of this paper is organized as follows. Section 2 presents our analysis and taxonomy of tracing solutions. Section 3 describes our tracing solution for the GR712RC which is subsequently evaluated in Section 4. The last two sections of this work present the related works and the main take away messages in Section 5 and Section 6, respectively.

[1] These solutions are usually referred to as development and debug solutions.

2 Profiling support for timing analysis

Multicore embedded platforms currently assessed for CES increasingly inherit a score of performance-improving features from the high-performance domain that usually exhibit a low degree of time predictability for being tightly modeled by static timing analysis approaches [51, 42]. More pragmatic approaches based on (or augmented with) on-target measurements are increasingly considered by embedded stakeholders for characterizing the timing behavior of critical applications. While static timing analysis derives mathematically provable bounds from an abstract model of the hardware and structural representation of the target application, measurement-based approaches and their hybrid variants aim at collecting empirical evidence from the execution on the real target, under the claim that there is no more accurate model than the hardware itself [52]. From an industrial perspective, measurement-based approaches are appealing given their similarities with the consolidated practice of functional verification. It is in fact at the verification and validation phases that the timing dimension is typically addressed and early figures from the design phases are assessed against actual observations.

In practice, however, the effectiveness of each approach depends on the characteristics of the system under analysis, with static analyses more equipped to deal with simpler, more predictable scenarios. As the increase in complexity of multicore hardware and advanced software functionalities is jeopardizing the applicability and effectiveness of conventional timing analysis approaches [51, 42, 1], it is becoming evident that novel forms of timing analysis are required that capture the peculiarities of multicore execution [34]. In particular, relevant aspects in multicore execution, such as utilization of shared resources and entailed contention delay need to be explicitly captured to meet emerging certification and qualification requirements (e.g., interference channels characterization [17] and freedom from interference [29]). Measurements appear to be the most practical way to meet this requirements and being able to gather timing evidence from actual execution is a fundamental prerequisite for any approach based on measurements [14, 13, 34, 30].

Monitoring and profiling solutions are becoming fundamental aspects in the timing verification. While several profiling and monitoring solutions exist, they have been designed and deployed for software/hardware debugging and (average) performance optimization purposes, and are not particularly tailored to timing analysis. In the following we cover some of the key trade-offs when considering different tracing solutions, with particular focus on the specific end user requirements.

2.1 Selecting the Profiling Solution for Timing Analysis

For timing analysis, profiling solutions usually build on the extraction of relevant information whilst the analyzed program executes. Here, the relevant information consists in all hardware events with bearing on the timing behavior. Modern COTS hardware platforms provide a more or less complex Performance Monitoring Unit (PMU) that allows accessing a set of hardware event counters via a set of Performance Monitoring Counters (PMC), which can be configured to track specific hardware events. Typical hardware events will be incremented either every time they occur (e.g., a cache miss) or track the number of cycles they affect (e.g., stall cycles). It is worth noting that monitoring support has been typically designed and implemented to support the low-level debugging of hardware components or high-level performance optimization, with completely different objectives than timing analysis.

While several profiling solutions exist, there is no consolidated solution for supporting multicore timing analysis as there is no unique set of timing analysis requirements. An industrial user should select the profiling solution (and look for the profiling support) that

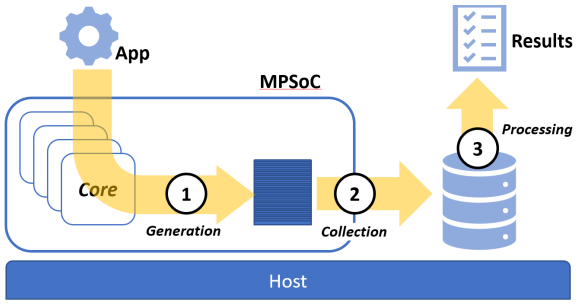


Figure 1 Measurement process breakdown.

better matches the particular qualification and certification requirements. In the following, we discuss relevant aspects that should guide the selection of the profiling approach.

2.1.1 Steps in the measurement process

The smaller unit of timing measurements is an observation item (`oItem`) that usually consists in the value of (a set of) event counters at a specific point in time. Based on our experience, we provide in Figure 1 a high-level view of a reference measurement framework. It comprises three main steps:

(1) **Generation of observation items.** This steps generates a snapshot of a given set of event counters (usually consisting in a time stamp and a set of events) and stores it in a dedicated location in the memory space. Approaches can be classified according to the mechanism used for triggering the generation of `oItems`, which is normally implemented by marking specific points in the program, also termed *instrumentation points*: we distinguish between *software* and *hardware* instrumentation. Further, the location used to store `oItems` can vary from a *shared memory* area to a *dedicated on-chip device*. The specific `oItem` generation approach is greatly affecting the measurement framework in terms of intrusiveness and required hardware support [20, 2].

(2) **Collection of observation items.** The next module or step is responsible for gathering the stream of collected `oItems` in order to enable the successive processing and analysis step. Exporting the observations out of the on-chip storing location is a delicate aspect in the process from the intrusiveness standpoint. It can be done either through *in-band* or *out-of-band* solutions [34]. The former category identifies those approaches where `oItems` are collected using the same hardware interconnect used by the application and standard debugger support. The latter solutions, instead, exploit dedicated trace collection mechanisms. One critical aspect in the collection step is that the amount of `oItems` to be collected can become pretty large, depending on the *scope* of the analysis: in this case, in-band solutions may become too resource-consuming to the point of resulting unusable. On the other hand, however, out-of-band approaches are only possible when the necessary hardware support is available [20, 33].

(3) **Processing of observation items.** When it comes to processing the set of `oItems`, we need to distinguish between *on-line* and *off-line* approaches. On-line approaches [13] are capable of processing the profiling information as soon as it is collected out from the target hardware, while the off-line alternatives can only process the collected data in a single block. The relevant difference between the two approaches is again relative to the scope and size of the analysis: off-line approaches require storing larger amounts of data

but limit the interference in collecting the `oItem` (as it only happens at the end of the experiment). On-line approaches limit the amount of data to be stored but may increase the interference if a dedicated out-of-band tracing solution is not available.

2.1.2 Characterization of a Profiling Approach

In the following we identify and discuss some of the relevant aspects to be considered in the selection of a profiling solution.

Intrusiveness of instrumentation. From the analysis process standpoint, it is fundamental that observations are collected over a system without introducing considerable distortion in the system behavior itself. The *probe effect* is a well-known potential pitfall of any experimental process. While in principle less intrusive approaches are to be preferred, the actual probing configuration may depend on the hardware and tool support for a specific target. Intrusiveness is largely affected by the instrumentation approach, which can be either software or hardware. Software instrumentation [43] guarantees easy integration with any target and development tool-chain with almost effortless application to any analysis scenario. The main requirement with software instrumentation is that the instrumentation code needs to be extremely lightweight in order not to overly affect the system behavior [10]. Software instrumentation frameworks offer the easiest approach for obtaining low-level, lightweight access to memory-mapped hardware monitoring counters. The interface to the memory mapped registers should guarantee minimum overhead and reuse across different configurations [22]. While standardized profiling API support is available for mainstream and high-performance targets [32, 7], no mature standard exists for embedded targets. Hardware instrumentation, while it requires specific debug support [20, 2] through advanced Debug Support Units (DSUs), it enable *zero-intrusiveness* `oItem` collection [13]. DSUs are increasingly present in target platforms in the embedded domain. On the downside, despite some standardization efforts [20, 2], the instrumentation framework needs to be adapted to the specific target and debug device.

Intrusiveness of data collection. Also the approach for storing the collected `oItem`s, and the data collection mechanism itself, are relevant when determining the intrusiveness of the profiling approach. In-band profiling, which is typically exploited by software instrumentation frameworks and relies on local memories to store `oItem`s, offers a straightforward solution guaranteeing easy integration. However, they are prone to generating non-negligible interference on the platform behavior, especially with fine-grained instrumentation. Out-of-band solutions, instead, are the mandatory approach when performance counters are not memory mapped or are made inaccessible to the user. These solutions are typically exploited by advanced debugging devices and tools to provide non-intrusive profiling solutions. Further, out-of-band solutions allow for larger amount of data to be collected without incurring the risk of affecting the system under analysis. Hardware solutions, however, are typically specific to a (family of) targets and, thus, support only limited reuse.

Scope of the Timinig Analysis. The effectiveness of a profiling solution also depends on the scope of the analysis itself. Different dimensions need to be considered: instrumentation granularity, type of collected information, execution model.

With respect to the instrumentation profile, conventional measurement-based analysis typically build on end to end measurements taken at task/partition level. Finer-grained analyses are considered for example by hybrid analysis approaches [30, 43] to be able to

correlate measurements over small code blocks with the structural information of the program under analysis. The scope of the analysis affects the frequency at which `oItem`s are produced, which in turn can increase the intrusiveness of the profiling mechanism. For multicores, the profiling solution may need to profile the execution of multiple cores at the same time, which can exacerbate intrusiveness issues and complicate `oItem` generation and collection. The alternative solution consists in building on `oItem`s extracted from execution in isolation and analytically defining conservative assumptions (e.g., on overlapping of contender requests) to enforce worst-case interference scenarios [8, 27, 12].

Regarding the type and amount of information to collect at each instrumentation point, conventional measurement-based timing analysis approaches typically require to associate a unique point in the program to a timestamp. More advanced approaches, as those advocated for multicore systems [34], may require to collect extensive information from the performance monitoring unit. Again, this may affect both the intrusiveness of the approach and the measurement process as not all events can be tracked at the same time, owing to the limited number of performance counters available.

Profiling is typically done at the level of single event counts, which means that `oItem`s store information on how many events (or cycles related to an event) happened in the observation period. In fact, and especially in multicore, some events are not necessarily independent and simple event counts may not carry sufficient information. For example, the latency of a bus access can be affected by recent events, e.g. previous accesses to the bus performed in a short interval. In these cases, event counts are not sufficient for a precise profiling because counters provide a measure of the event in the observation period, but fail to provide any information about how the event distributes over time. In these cases, full traces are required, with notable effects on the amount of data to be collected and processed. Large amounts of data are not easily handled with in-band solutions and can be also challenging the processing capabilities of in-line processing approaches [13]. In fact, in-band approaches that cannot guarantee low intrusiveness simultaneously collecting events and timing information can penalize the accuracy of the latter as they factor in the cost of profiling.

Hardware support. The hardware support for run-time monitoring available in a specific target is instrumental for the selection of the profiling approach. Since debug support is not the primary driver of the platform selection process, the selection of the profiling solution is sometimes not a choice. Available solutions can range from fully integrated on-chip solutions to more complex off-chip solutions based on external devices.

Fully integrated on-chip solutions represent the baseline approach to support hardware profiling. A specialized debugger, normally developed by the same hardware manufacturer, is executed in the host platform which is in turn connected to the target through a generic communication port (e.g., JTAG, GPIO, Eth). This probably represents the cheapest and less demanding approach for profiling. This scenario can support hardware instrumentation but can only implement in-band tracing: the `oItem`s are stored in the on-chip memory, typically in a circular buffer, while a daemon running in the host is responsible for moving the `oItem`s to the host memory space for off-line processing. On-line processing, although in principle supported, is discouraged by the reduced bandwidth guaranteed by the in-band solutions. Specialized external hardware can also be deployed to extract the `oItem`s [33].

On the other side of the spectrum, specialized external hardware support is available that delivers tracing capabilities with the combination of an external debugging device, which is connected to the target via a generic or target-specific probe and high-bandwidth

protocols (e.g., AURORA). The host machine is then connected to the debugging device to configure the profiling process and select the `oItem`s. While relatively more expensive, specialized hardware debug support allows to take advantage of out-of-band tracing support. Commercially available debugging devices (e.g., Lauterbach) normally allow some type of reuse across a family of targets by changing only the terminal probe. Specialized out-of-band capabilities and debug hardware support represent the best configuration for zero-intrusiveness profiling solution [13]. From the timing perspective, it is also worth mentioning that specific hardware support is also determining the *resolution* for timing measurements as available implementations may support different frequencies in the clock used to get the timestamp, which is not always the same as the one exhibited by cycle clock frequency. Throughout all the experiments performed, the GR712RC board was operating with its default clock frequency of 80MHz.

Safety-Related Requirements. Profiling solutions may also hit safety certification aspects. Within software instrumentation approaches, the instrumentation code (usually a light-weight macro) needs to be carefully considered from the software certification standpoint. The instrumentation code may need to remain in the final software configuration as the analysis results are valid for the instrumented program, but some certification requirements may be against leaving deactivated or not strictly functional code in the target program. The approach to follow depends on the specific industrial domain and certification standard [10].

Other industrial Requirements. The selection of the hardware profiling approach can be steered by more practical concerns. These includes purely technical aspects such as integration with a specific host OS or the software development tools and processes. The principal criterion from the industrial standpoint is the cost/benefit ratio: the profiling framework is assessed with respect to the cost of the solution itself and the induced cost (training, use, etc.). It stands to reason that the optimal solution depends on the actual profiling requirements and on the available support on the target platform. Further, in a longer perspective, portability of the profiling solution is also a major industrial concern: having to rethink and redeploy consolidated, profiling tools and practices because of a shift in the family of processors or RTOS is simply unaffordable.

3 Profiling a Space Application on the GR712RC

In this section, we present the target hardware platform and the requirements coming from the multicore timing analysis approach we use, and then introduce our profiling solution to cover them building on the support in the underlying board.

3.1 Target platform

The GR712RC board by Cobham Gaisler is a common choice for space missions [9] and it is the target of our space case study. The GR712RC is a radiation-hard-by-design board intended for aerospace applications. It integrates a dual-core LEON3 processor (see Figure 2), which is connected to the rest of the on-chip devices via a high-bandwidth AMBA AHB Bus. Other components such as the UART are first connected to a low-bandwidth AMBA APB Bus that is a master to the AMB AHB Bus.

Each core has its private instruction and data caches, with the rest of the processor resources, such as memories and peripherals, shared between both cores. For our study, we are particularly interested in capturing accesses to the on-chip SRAM, the off-chip SRAM

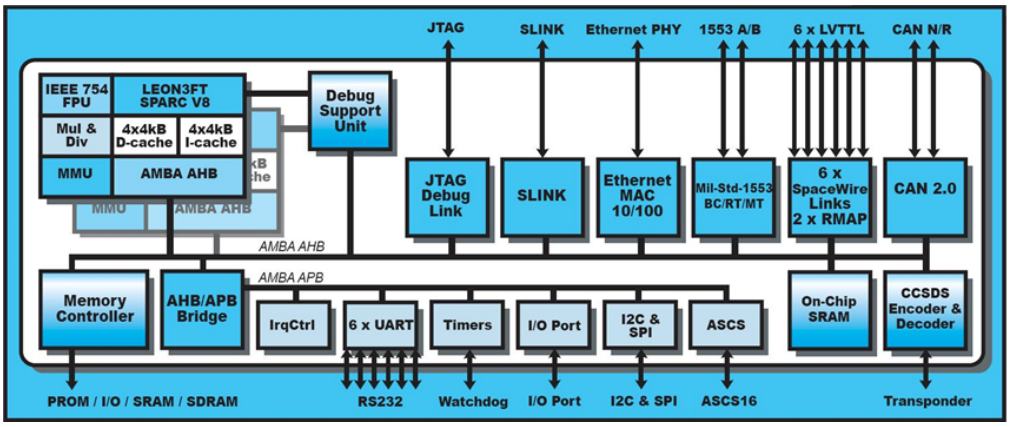


Figure 2 Block Diagram of the GR712RC [46].

and SDRAM, and the UART, as they are the resources used by the case study application. We target at deriving contention bounds by tracking read/write operations to these resources.

Unlike other similar systems-on-chips (SoC), the GR712RC lacks a PMU, which is in charge of collecting processor usage statistics, e.g. related to the memory accesses performed and instruction types executed by the core. The decision of not adding a PMU is a design choice of the hardware manufacturer and can be related to area or cost constraints. The GR712RC comes equipped with a DSU that can be accessed connecting the GR712RC board to a host using a JTag cable. We used GRMON [47], a debug tool provided by Cobham Gaisler, to connect a host computer to the DSU and issue debug commands to it, such as to extract data from the ITB and the BTB. The processor must be stopped before the traces can be extracted. We use breakpoints and step-by-step execution to stop the processor and issue debug commands, including trace extraction.

3.2 Profiling Requirements

We aim at using our profiling solution in the context of multi-core processors, with the goal of characterizing application sensitiveness to inter-core contention. We also target deriving bounds to the worst contention each request type can suffer accessing each shared resource.

As presented in the introduction, the contention a task τ_a suffers accessing shared resources Δ_a^{cont}, can be computed by exploiting the maximum number of requests each task τ_a performs to each shared resource r, N_a^r, hardware resource and the worst-case contention delay each request type can suffer, $L_{r,y}^{max}$. That is, $\Delta_a^{cont} = f(N_a^r, L_{r,y}^{max})$.

Regarding the latter, $L_{r,y}^{max}$, in the GR712RC it covers the contention accesses suffer accessing the AHB bus. In the worst scenario, a request is sent from one core at the same time another request is sent from the other core, with the latter getting priority on the bus. In this scenario, $L_{r,y}^{max}$ matches the duration of the latter request. Hence, the piece of information we need from the tracing solution is the number of accesses to the shared resources, breaking them down between reads and writes, while the contention requests generate each other are derived empirically.

The following features of the target board and the contention modelling approach are relevant for the proposed tracing solution.

1. For the GR712RC, the number of requests a task performs to the different memories in isolation matches the number it does with any co-runner task, assuming tasks are

independent. While the latency it takes each request, of course, is affected by the integration with other tasks, the total number of requests is not.
2. The multicore timing model factors in the interleaving among requests. This is so because, as explained above, the model assumes the worst interleaving between the task under analysis and the contender task.

The first feature allows us to perform the profiling of access counts when each application runs in isolation, removing the need of performing multicore executions. The latter makes that when we collect the number of accesses, the time when they happen is irrelevant, so when our solution captures end-to-end access counts it can affect the timing between requests.

3.3 Proposed profiling solution

The profiling approach we propose in this work builds on end-to-end observations and relies on a limited set of monitoring counters. More importantly, we do not need to associate events and timing, as the contention impact on timing is analytically modeled exploiting event information. This allows us to collect timing information and event counts separately. Our profiling solution provides on-line support for the trace collection and off-line support for the processing of the trace, see Figure 3. The on-line support consists in a GRMON script that loads an image and collects the BTB and the ITB (`Dumper`). The off-line part provides a (`Merger`) function to filter the output file produced by the Dumper, containing combined data from both the ITB and the BTB. The `Merger` removes repeated and redundant entries, processes the raw data, and isolates two separated data structures, one for the ITB and another for the BTB. In order to identify some of the events, it also performs a merging process of both traces, to identify the instructions generating each request to the bus. The second step of the post-processing script derives the access counts of each kind (`Collector`).

3.3.1 Implementation

We work-around the lack of a PMU by using some of the debug features included by the Debug Support Unit (DSU). The DSU is connected to the main AHB Bus which, in turn, connects cores, debug I/O, and on-chip/off-chip memories. It is also connected to the cores directly to issue commands and receive information from the cores by-passing the shared AHB. It provides regular debugging capabilities such as breakpointing, step-by-step execution, and memory inspection. The DSU provides two key features for our study.

- The DSU snoops the AHB and captures the activity on the bus (btrace) initiated by the cores when accessing the different resources. This is provided as a trace with events recorded when they occur.
- The DSU also captures a stream of executed instructions in each core. The stream is built from the chronological sequence of instructions architecturally executed. For each instruction in the stream, the DSU records information like the instruction itself and its PC (more details below).

In the bus trace we can identify the source (core) of all the activity by checking the AHB MASTER field, which uniquely identifies the origin of the request. Also, each of the different memories are mapped to different ranges of the address space. By tracking the addresses generated by each event we know the memory it targets. Using the same approach we identify accesses to peripherals such as the UART. With these two pieces of information we are able to identify unambiguously the source and the target of each entry in the trace.

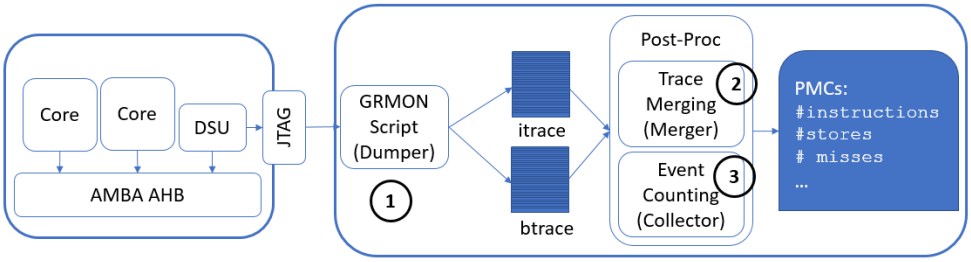


Figure 3 Trace collection process with created modules shown in grey.

This information is stored into two separated circular buffers called (AHB) Bus Trace Buffer (BTB) and Instruction Trace Buffer (ITB). The BTB is filled by snooping the AHB, while the ITB is filled by directly receiving the instruction execution stream from the cores. The buffers are filled fast, but they cannot be dumped as fast because the DSU is connected to the debug port via the shared AHB bus. The buffers are 256 entries long each, and their content must be dumped before becoming full or data may be overwritten and lost.

- Each BTB entry includes the following fields: timestamp with the cycle the request was observed in the bus, program counter (PC) of the instruction causing the bus transaction, target memory address, data, access being a read or a write, and AMBA protocol options, including `AHB HTRANS`, `AHB HBURST`, `AHB HSIZE`, `AHB HMASTER`, `AHB MASTLOCK` and `AHB HRESP` among others [46]. More information about these fields can be found in the AMBA AHB specification [3].
- Each ITB entry includes the following fields: timestamp, PC, instruction word (`iword`), result (or data for memory instructions), as well as bits signaling trap (indicating that the processor is in debug mode, resulting in tracing the instruction twice and as a result these entries are filtered out), processor error mode, and single or multicycle instructions [46].

3.3.2 Dumper

The data fields in the trace buffers cannot be collected in real time, and the buffers need to be dumped regularly to prevent them from overrunning. That is achieved by the Dumper by breakpointing the region of interest and using a step-by-step execution. The **Dumper** is a TCL script that issues GRMON commands. It is executed when the GRMON debugger is launched, connecting the host to the board. It first loads the executable binary to the board, sets the entry point and the stack pointer to fixed addresses when running in BM, and sets a breakpoint at the beginning of the region of interest, which is the region to be traced. This region can be either a whole task or just a part of it. The processor is booted and the binary executed normally until the breakpoint is reached. The control is then returned to the Dumper, which resumes the program using a step-by-step execution.

Every 16 steps, the DSU breaks its execution and the contents of the BTB and the ITB are dumped to the host through the UART. The selected step (16) is deemed to be conservative (small) enough due to the size of the buffers (256 entries each). That causes dumping several entries from the trace buffers to be sent more than once, but they are filtered at a later stage by the Merger. An even smaller step would imply slowing more the process needlessly.

The tracing dumping process is executed until the program counter that determines the end of the region of interest is encountered. At this point, the Dumper stops executing the

Table 1 Collected Events.

Event ID	Event
Icount	Total number of instructions executed
LDc	Total number of load instructions executed
STc	Total number of store instructions executed
LDm	Total number of data cache load misses
LDh	Total number of data cache load hits
Ihits	Total number of instructions cache hits
Imiss	Total number of instructions cache misses
Ifill	Total number of instructions snooped as a result of an instruction miss
Bus	Total number of bus accesses
LD_SDRAM	Total number of loads from the SDRAM memory
LD_offSRAM	Total number of loads from the off-chip SRAM memory
LD_onSRAM	Total number of loads from the on-chip SRAM memory
LD_IO	Total number of loads/reads from the I/O peripherals
ST_SDRAM	Total number of stores to the SDRAM memory
ST_offSRAM	Total number of stores to the off-chip SRAM memory
ST_onSRAM	Total number of stores to the on-chip SRAM memory
ST_IO	Total number of stores/writes to the I/O peripherals

program and the dumped file is saved. The outcome of this dumping process results in a plain text file, comprised of a series of pairs of 256 ITB elements followed by 256 elements from the BTB.

3.3.3 Merger

Once the region of interest is completely traced and the output file has been saved, it is **Post-processed** in the host.

As a first step of the **Merger**, the btrace and the itrace are filtered into two separate data structures using Pandas, which is an open source data analysis tool built on top of Python. At this point, the Merger also removes redundant entries introduced as a result of the conservative tracing dumping rate. Then, we post-process the opcode from the iword field in order to determine the instruction type.

As a second and last step, the Merger links up the entries of the data structure built up from the btrace, which correspond to data load misses in cache to the load instruction that causes that bus activity. To do so, we filter those events in the bus data structure whose opcode is that of a load, their `AHB HTRANS` field value is *non-sequential* and their `AHB HBURST` field value is *single*. Then, we match them with the entries of the instruction data structure which correspond to a load instruction traced (executed) one cycle after being traced (snooped) in the bus. The "tracing relationship" for the difference in cycles for a load miss was empirically observed across several experiments, always matching this pattern. We relate both data structures to identify instruction misses in a similar manner.

3.3.4 Collector

Finally, once the data structures are filtered and properly processed, the **Collector** derives the counts of the events for each kind of access. As a result of this step, we derive the events listed in Table 1 as follows.

- The number of instructions executed by checking the number of entries in the instruction data structure. Instruction cache misses are identified by matching the *address* field in both traces, and checking that in the bus data structure the access is in read mode (instruction cache miss) and the value of the `AHB HTRANS` field is *non-sequential*. Conversely, the instruction fetches resulting in a cache line fill are identified in the bus data structure as

those with a *sequential* value for the `AHB HTRANS` field, and an *incremental* value for `AHB HBURST`. These usually occur as a set of 7 instructions fetched after an instruction miss, which is consistent with the size of an instruction cache line. The reasoning, is that an executed instruction which is also traced in the btrace implies that the instruction was not found in the instruction cache and therefore had to be fetched.
- We derive the instruction cache hit count by counting the amount of instructions executed that do not cause bus activity. To do so, we subtract from the instruction count the sum of instruction misses and instruction line fills.
- The total number of load and store instructions executed is derived from the data structured by the Merger, which has previously decoded the instruction type. Data cache load misses are identified as detailed in 3.3.3.
- Data cache load hits are derived by subtracting these misses to the total amount of loads.
- The total number of bus accesses from the bus data structure, as well as their target memory, is derived by checking their *address* field in the case of load instructions, since each target is mapped to a different address range. The target memory of the store instructions can be extracted from the instruction data structure, concretely from its *result* field, which contains the target address.

Overall, the proposed tracing solution enables deriving instruction and access counts accurately as needed by the contention model.

4 Experimental Evaluation of the Profiling Solution

The experimental evaluation has a three-fold objective. We aim at providing evidence on the accuracy of the implemented profile library (Section 4.1). We also deploy the same library to derive a timing characterization of the impact of contention on accessing the shared memory devices in the GR712RC (Section 4.2). Finally, we use that timing characterization for deriving a preliminary model potentially incurred by a space case study (Section 4.3).

4.1 Validation of the profiling solution

We build on the concept of specific code snippets to assess the accuracy of the proposed profiling solution, which covers two dimensions.

- Comparing the expected access count values with the ones obtained with our profiling solution in a bare metal setup. This allows assessing the accuracy of the solution in a pristine scenario.
- Comparing the results obtained with the profiling solution when we run the exact same code snippet under bare metal and RTEMS. This allows assessing any portability issues of the solution for different RTOS on the GR712RC.

To satisfy these goals, code snippets are designed so that a hardware expert with understanding of the GR712RC architecture can provide high-accurate estimates of the expected access counts. Also, they are small enough for the expert to be able to reasonably handle them. A preliminary exploration on the use of specialized code snippets for characterizing multicore timing interference has been reported in [41]. The basic structure of each code snippet is a main loop with a large body comprising one or two types of instructions only, usually load and/or store. On the one hand, this reduces the overhead of the loop control instructions; on the other hand, by playing with the range of addresses accessed by the load/store operations, we force accesses to be sent to the on-chip SRAM, the off-chip SRAM/SDRAM or the UART.

```
for(i=0; i<ITER; i++) {
    OPERATION;
    OPERATION;
    OPERATION;
    OPERATION;
    OPERATION;
    OPERATION;
    OPERATION;
    OPERATION;
    OPERATION;
    // ...
    OPERATION;
    OPERATION;
    OPERATION;
    OPERATION;
    OPERATION;
    OPERATION;
}
```

Figure 4 pseudo-code of the code snippets.

Table 2 List of code snippets.

Name	Description
cs_ic_hit	Causes instruction cache hits
cs_ic_miss	Causes inst. cache misses and line refills
cs_dc_hit	Causes data cache load hits
cs_on_SRAM_rd	Causes on-chip SRAM read accesses
cs_on_SRAM_wr	Causes on-chip SRAM write accesses
cs_off_SRAM_rd	Causes off-chip SRAM read accesses
cs_off_SRAM_wr	Causes off-chip SRAM write accesses
cs_off_SDRAM_rd	Causes off-chip SDRAM read accesses
cs_off_SDRAM_wr	Causes off-chip SDRAM write accesses
cs_UART_rd	Causes UART read accesses
cs_UART_wr	Causes UART write accesses

The range of accessed addresses can also be changed to force those accesses to cause capacity misses in cache. The high density of desired operations in these code snippets, in combination with visual inspection of the object code, enables the hardware expert to predict with high accuracy the expected behavior with regard to the most relevant events.

We assess the accuracy of the proposed solution by applying it to the code snippets listed in Table 2. Experiments are executed in a single-core scenario and as specified in Section 3. We compare the values we collect to those expected. In the following subsections we present the validation for some code snippets, due to space limitations we do not describe the validation of cs_ic_miss and cs_ic_hit. Also note that in each validation experiment we show only relevant events.

4.1.1 Bare Metal Snippets

cs_dc_hit. This code snippet triggers loads that systematically hit the L1 data cache. Prior to profiling the code snippet, the loop function is executed in order to warm up the caches and avoid as many cold misses as possible. These cold misses go to the off-chip SRAM memory. A total of 128 load instructions are executed in each iteration of the loop that iterates 1000 times. Table 3 shows that the reported values by our library have high accuracy in terms of instruction count (ICount), load and store counts (LDc and STc), load hits and misses (LDh and LDm), instruction cache hits (Ihits), and accesses to the off-chip SDRAM (SDRAM) and the on-chip SRAM (oSRAM). In the worst case the deviation is 0.03% for instruction counts. Such tiny deviations are regularly observed in many COTS, even in single core and non-speculative ones. Another possible reason could also be tied to the rotating trace buffer. Some requests slightly before and after the region of interest may be included in the trace dump. The 23 accesses to the off-chip SRAM reported in Table 3 are due to cache line re-fills, which are also considered and captured by the post-processing script. For the instruction hits (Ihits) count, we expect all instructions (Icount) to hit in the L1 instruction cache. Further experiments showed that, as we increase the loop iteration count, the relative deviation decreases, hinting at a constant instruction count overhead. Also, the obtained counts are deterministic across several runs.

cs_X_rd. Table 4 shows the derived and expected access counts with the snippets designed to systematically miss with load operations from the L1 cache and target only one of the different memories: cs_on_SRAM_rd, cs_off_SRAM_rd and cs_off_SDRAM_rd.

Table 3 Validation of the profile solution with cs_dc_hit.

	Load Hit		
Event	Exp.	Obs.	Dev (%)
Icount	131000	131040	0.03
LDc	128000	128004	0.00
STc	0	1	-
LDm	0	1	-
LDh	128000	128003	0.00
Ihits	131040	131036	0.00
LD SDRAM	0	0	0.00
LD offSRAM	0	23	-
LD onSRAM	0	0	0.00
ST offSRAM	0	1	-

Table 4 Expected & Observed event counts, and relative Deviation (%) for cs_X_rd.

	OnSRAM			OffSRAM			SDRAM		
Event	Exp.	Obs.	Dev	Exp.	Obs.	Dev	Exp.	Obs.	Dev (%)
Icount	131000	131073	0.06	131000	131024	0.02	131000	131024	0.02
LDc	128000	128006	0.00	128000	128001	0.00	128000	128001	0.00
STc	0	1	-	0	0	-	0	0	-
LDm	128000	128003	0.00	128000	128000	0.00	128000	128000	0.00
LDh	0	3	-	0	1	-	0	1	-
Ihits	131073	131061	-0.01	131024	131021	0.00	131024	131021	0.00
LD SDRAM	0	0	0.00	0	0	0.00	128000	128000	0.00
LD offSRAM	0	43	-	128000	128007	0.01	0	0	0.00
LD onSRAM	128000	128000	0.00	0	0	0.00	0	0	0.00
ST offSRAM	0	1	-	0	0	0.00	0	0	0.00

In order to avoid any potential residual data in the cache from previous executions, the loop functions are also run before starting the profiling phase. Similarly to cs_dc_hit, the loop function consists of 1000 iterations over a loop which performs 128 load operations to every given memory, with a stride between them so that every load instruction causes a miss in the L1 data cache.

Results for all three memories confirm a very high accuracy for the relevant events, with the worst case deviation (0.06%) being again associated to instruction counts. The 43 loads from the off-chip SRAM in cs_loadmiss_onsram, include instruction misses and instruction cache line refills that are triggered as a consequence of an instruction cache miss.

cs_X_wr: We proceed likewise with the code snippets that perform write operations to the different memory devices in the board (cs_on_SRAM_wr, cs_off_SRAM_wr, and cs_off_SDRAM_wr). Once again, these code snippets consist of 1000 loop iterations triggering 128 store operations to addresses of every particular memory. Given the write-through no-allocate policy, we do not need to pre-heat the caches, as each store results in a bus access and no content is loaded into the L1 cache. Accuracy results, not shown for space constraints, are consistent with those observed for read operation.

Different cs_UART_X: For the UART snippets, which read/write from/to an I/O device, we first configure the registers of the UART, and then perform reads or writes in a loop composed of 128 accesses to the memory-mapped address for data in the UART registers. Results confirm the accuracy of our tool to trace and collect events, with a maximum deviation of 0.11% in the case of the instruction count.

4.1.2 RTEMS Real-Time Operating System

One of the requirements for our profiling solution is the ability to support different real-time operating systems, with minimum effort, as some case studies run bare metal while others run on consolidated RTOS. To show adherence to this requirement, we evaluate our profiling method with the Real-Time Executive for Multiprocessor Systems (RTEMS) v5.

Changes in the profiler. The adaptation of our solution to RTEMS required no change to its scripts, with the steps of tracing and post-processing matching those for bare metal. At the RTEMS level, the only configuration required is to either use a uniprocessor scheduler or disabling the other potentially contending cores when profiling an application and using a multicore scheduler. Both allow tracing in isolation as required by our profiler.

Changes in the Code Snippets. Our goal when developing the snippets for RTEMS was ensuring that we had the same binary running in BM and in RTEMS. This objective is achieved by compiling the snippet into a .o object file with the sparc-elf-gcc cross-compiler, then linking it either into a BM image or an RTEMS image by using the sparc-elf or the sparc-rtems tool-chain linker. In order to enable the use of the same object file for BM and RTEMS the code snippet is built without any library or system call dependencies.

Evaluation. For the evaluation we proceed analogously as for bare metal, comparing the expected and observed counts. The region of interest is the loop function of the code snippets, i.e. after RTEMS has already been initialized. As a result, we do not expect variance in the results due to the interference by the RTOS. Table 5 reports expected and observed results for the snippets reading from the different memories under RTEMS. Results are very similar with a slight difference in the number of instructions executed by the processor (0.1%) and the load count (0.02%). This differences, which can be caused by the RTOS, are deemed as negligible. We also conducted the same experiments for the other code snippets, whose results are not shown for space constraints, resulting in the same conclusions: 0.12% Icount at most and no deviation in STc for all the cs_*_wr snippets. The results show a very small deviation between expected and observed values, even in the presence of an RTOS.

Also, as we have the same code running in BM and in RTEMS and we trace the same region of interest, we can perform a direct comparison between the results under BM and RTEMS, e.g. Table 4 and Table 5. The fact that the tool-chain can be applied to RTEMS without any modification neither in the collector code nor in the dumper and merger scripts, shows that this process is easily extensible to embrace RTOS.

Table 5 Validation with memory read snippets in RTEMS.

	OnSRAM			OffSRAM			SDRAM		
Event	Exp.	Obs.	Dev	Exp.	Obs.	Dev	Exp.	Obs.	Dev (%)
Icount	131000	131136	0.10	131000	131136	0.10	131000	131136	0.10
LDc	128000	128022	0.02	128000	128022	0.02	128000	128022	0.02
STc	0	0	-	0	0	-	0	0	-
LDm	128000	128002	0.00	128000	128002	0.00	128000	128022	0.02
LDh	0	20	-	0	20	-	0	1	-
Ihits	131136	131129	-0.01	131136	131129	-0.01	131136	131129	-0.01
LD SDRAM	0	0	0.00	0	0	0.00	128000	128000	0.00
LD offSRAM	0	44	-	128000	128045	0.04	0	44	-
LD onSRAM	128000	128000	0.00	0	0	0.00	0	0	0.00

4.2 Evaluation Results: Contention Slowdown Matrix

As we introduced in Section 3, contention models typically build on access counts, which we can derive with the support of our profiling tool as shown in Section 4.1, and worst-case contention latencies to each target shared resource. In fact, we need for every pair of requests type/target resource the contention they generate each other. This information is stored in the *slowdown matrix*. Each cell in the slowdown matrix is generated by running stressing benchmarks [12] that put maximum load on the resource. It follows that the reliability of the slowdown matrix builds on that of the stressing benchmark used in the experiments. To cover the latter, evidence is required that stressing benchmarks intensively stress their target resource.

Table 6 Stressing benchmark validation.

		Expected Relation	Acc.	Validated
On-c. SRAM	RD	LDc ≈ Icount;	97%	yes
		LDc = LDm = LD_onSRAM = Bus	100%	yes
		Ihits = Icount	100%	yes
		LD_offSRAM = LD_offSDRAM = LD_IO = 0	100%	yes
	WR	STc ≈ Icount;	95%	yes
		STc = ST_onSRAM = Bus	100%	yes
		Ihits = Icount	100%	yes
		ST_offSRAM = ST_offSDRAM = ST_IO = 0	100%	yes
Off-c. SRAM	RD	LDc ≈ Icount;	97%	yes
		LDc = LDm = LD_offSRAM = Bus	100%	yes
		Ihits = Icount	100%	yes
		LD_onSRAM = LD_offSDRAM = LD_IO = 0	100%	yes
	WR	STc ≈ Icount;	95%	yes
		STc = ST_offSRAM = Bus	100%	yes
		Ihits = Icount	100%	yes
		ST_onSRAM = ST_offSDRAM = ST_IO = 0	100%	yes
Off-c. SDRAM	RD	LDc ≈ Icount;	97%	yes
		LDc = LDm = LD_offSDRAM = Bus	100%	yes
		Ihits = Icount	100%	yes
		LD_offSRAM = LD_onSRAM = LD_IO = 0	100%	yes
	WR	STc ≈ Icount;	95%	yes
		STc = ST_offSDRAM = Bus	100%	yes
		Ihits = Icount	100%	yes
		ST_offSRAM = ST_onSRAM = ST_IO = 0	100%	yes
UART	RD	LDc ≈ Icount;	97%	yes
		LDc = LDm = LD_IO = Bus	100%	yes
		Ihits = Icount	100%	yes
		LD_offSRAM = LD_offSDRAM = LD_onSRAM = 0	100%	yes
	WR	STc ≈ Icount;	97%	yes
		STc = ST_IO = Bus	100%	yes
		Ihits = Icount	100%	yes
		ST_offSRAM = ST_offSDRAM = ST_onSRAM = 0	100%	yes

In our setup, we leverage our profiling solution to effectively achieve this goal. While the goal for snippets was to check that the expected absolute event count values matched the observed ones, for stressing benchmarks the goal is to ensure a given relation between event counts. For instance, a read stressing benchmark on the on-chip SRAM should have (1) as many instructions as load operations; (2) as many dcache hits, dcache misses, and on-chip SRAM accesses as load operations. The former condition captures the fact that, except for few control instructions in the main loop of the stressing benchmark, the rest of the instructions should match the target type. Few additional instructions can also be added to avoid systematic behaviors [19]. The latter condition, instead, states that read operations must miss in the data cache and access the on-chip SRAM.

In Table 6 we present the microbenchmarks (uBs) developed as well as the expected relation among event counters, and the assessment done from the observed event counts obtained with our tracing solution. We observe that the stressing benchmarks can be deemed as satisfactorily achieving their goal of stressing its target resource. This provides evidence on the results obtained in the Slowdown Matrix, see Table 7, with stressing benchmarks. The first column of the table reports the number of clock cycles taken to execute each type of instruction running in isolation. The following columns indicate the number of cycles taken by the same instruction when the other core is executing another particular instruction intensively.

4.3 Case Study

We evaluate our profiling approach on a real space application realizing a subset of the telemetry and telecommand functionalities. The telemetry and telecommand (TM/TC) subsystem is a spacecraft component that (i) allows a remote collection of measurements

Table 7 Derived Slowdown Matrix for the GR712RC [number of cycles].

				Contender							
				On-chip SRAM		Off-chip SRAM		SDRAM		UART	
			Isol.	RD	WR	RD	WR	RD	WR	RD	WR
Analysis	On-c. SRAM	RD	7	9.0	8.5	11.0	10.0	12.0	8.1	9.0	8.0
		WR	2	4.3	3.0	4.5	7.0	5.0	6.1	3.5	5.0
	Off-c. SRAM	RD	8	11.0	9.0	12.0	11.0	13.0	11.1	10.0	9.0
		WR	6	10.0	7.0	11.0	11.0	13.0	11.8	9.0	9.0
	SDRAM	RD	9	12.0	10.1	13.0	13.0	14.1	13.2	11.1	10.1
		WR	6	8.0	6.1	11.0	12.0	13.1	12.1	7.1	7.1
	UART	RD	6	9.0	7.0	10.0	9.0	11.1	7.1	8.0	7.0
		WR	4	8.0	5.0	9.0	9.0	10.0	7.1	7.0	7.0

(telemetry) and their transmission to ground-based facilities, and (ii) receives commands (telecommand) from the ground allowing a direct control of the spacecraft during spacecraft development, assembly, integration, test, launch phases and operation. The TM/TC is typically connected to every other spacecraft subsystems using a variety of interfaces such as CAN buses, Spacewire, Spacefiber or RS-232 connections, while the communication to the ground segment relies on RF baseband interfaces.

For the purpose of this paper, we evaluated three applications composing the TM/TC subsystem, restricting ourselves to the UART communication for taking care of the I/Os. The first application is a **watchdog** application whose purpose is to ensure continuity of service for an unmanned system, which in fact could not be humanly operated in case of failure. The watchdog performs health monitoring on the other co-running applications composing the TM/TC subsystem, rebooting the hardware board if the service of any other application is discontinued as a result of any type of misbehavior: infinite loop, software crash, unhandled exception, etc. For this purpose, the watchdog relies on hardware timers available in the GR712RC board, setting up a counter that is progressively decremented, and would reboot the board whenever it reaches zero. In the meantime, the other applications are regularly sending a *keep-alive* signal to the watchdog application, that reset the watchdog timer when all signals have been received.

The **scrubber** application, which is in charge of assuring the reliability of the memory subsystem in a potentially radiated environment. It aims at exercising and refreshing the ECC protection of the GR712 off-chip SRAM memory. The scrubber progressively reads the whole 8MB SRAM memory per 64B blocks, and when detecting an ECC error correction, writes the word back to memory, so that further bit shift will not make the error non-correctable. The scrubber performs uncachable word-per-word load accesses to the memory and checks, for each load, the AHB status register to check if an ECC correction occurred.

Finally the **crypter** application implements the AES encryption / decryption of telemetry data and ground commands. The crypter is relying on a symmetric block cipher based on a substitution–permutation network that involves several computation rounds.

4.4 Profiling and Contention Results

Following the methodology we have presented in the previous section we derive the event counts for the three applications that resulted in the profiles shown in Table 8.

We derive fully time-composable [11] execution time estimates (*FTCestimate*) results.

Table 8 Resulting profiles for the considered TM/TC sub-system elements.

Event	Crypter	Scrubber	Watchdog
Icount	2057	539	26
LDc	513	132	5
STc	121	2	2
LDm	312	132	5
LDh	201	0	0
Ihits	1882	531	19
Imiss	176	8	7
I line fill	39	12	16
LD SDRAM	0	0	0
LD offSRAM	531	87	27
LD onSRAM	0	0	0
LD IO	1	65	2
ST SDRAM	0	0	0
ST offSRAM	121	2	1
ST onSRAM	0	0	0
ST IO	0	0	1

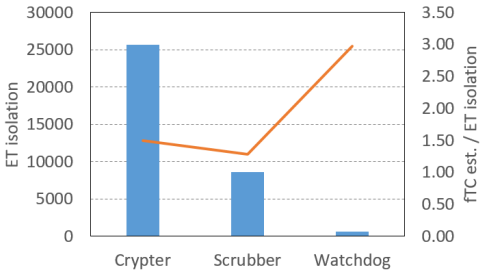
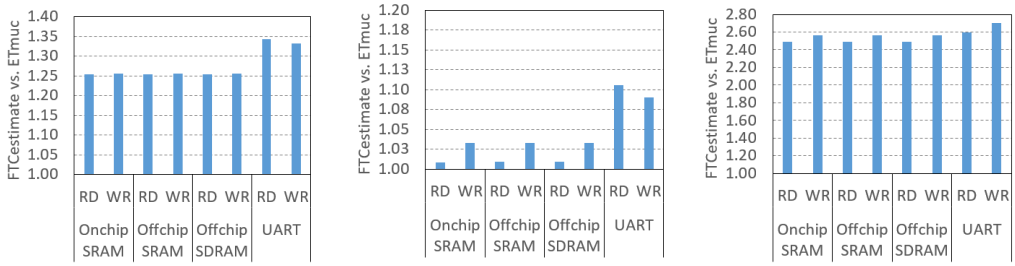


Figure 5 Comparison to ETisol.

FTC estimates assume that each request generated by the application contends with one request, so that both request arrive at the same time but the application request always looses the arbitration, suffering maximum contention. For instance if the application generates a read to the on-chip SRAM, the contender generates a read to the Off-chip SDRAM on the same cycle and wins the arbitration. This results in the request from the application suffering a contention of 12.0 cycles.

Figure 6 compares the FTCestimate with the observed execution time when we run each application against each stressing contender, i.e. the execution time of the application under a multicore scenario (ETmuc). As expected, the estimate is always higher than ETmuc. For Crypter and Scrubber we observe reduced over-estimation that ranges from 1.25 to 1.35 for the former and from 1.01 to 1.11 for the latter. This over-estimation relates to the alignment between the request of the application and the one of the contenders. In general, it is hard if at all possible to achieve that the request from an arbitrary application aligns with that of its contender. The FTC model instead assumes this case, as it can in theory happen. This results in some over-estimation, that is limited though. For the Watchdog, given its small execution time (201 cycles in isolation) the impact of the worst-case assumption on the alignment becomes more significant in relevant terms. However, it is irrelevant in global terms.



(a) Crypter (b) Scrubber (c) Watchdog

Figure 6 Observed multicore execution times vs. fully time-composable estimate (FTCest).

In Figure 5 we report in the bars the absolute execution time of each application when it runs in isolation (ETisol). It is measured on the left vertical axis. The line shows the ratio between FTCest/ETisol, that is the ratio between the estimated and the execution time in

isolation. For Crypter and Scrubber it remains below 1.5x, while for the watchdog due to its very small execution time, it goes above 3.0. This is so, as any small over-estimation of the FTC estimate is significant in relative terms.

4.5 Discussion

While at the beginning of a project many frameworks seem to provide a similar cost/benefit ratio, a detailed analysis is required to understand what is needed (user requirements) by the case study and the timing analysis solution, and what is the support provided in terms of hardware and software support by the chip vendor (DSU and GRMON respectively in our case). As the existing debugging support is not intended for timing analysis, it is necessary to cover the gap between user requirements and debug support. This involves gathering a deep understanding of hardware tracing and developing scripts to interact with the software tracing support. A fair amount of time must be devoted to validating the solution before it can be used with a high degree of trustworthiness. In this line, our next step is to deploy the developed solution with other applications used in specific business units of Thales.

In terms of reusability and portability, the particular scripts for trace collection are depending on the debug solution. They can be used for processors of the same family with similar debug solutions. For instance, we are using our solution for LEON4-based multicore processors. For a wider applicability, the solution can also be deployed on top of external debugging solutions and protocols (e.g. Lauterbach and AURORA respectively), that support different families of processors and allow to capture the baseline information we exploit in our method. Last but not least, the main steps of our overall methodology, and in particular its validation part, will be a necessary part of any profiling solution to provide evidence of correct behavior.

5 Related Work

Software and hardware profiling is an umbrella term that encompasses many forms of dynamic analysis techniques, all of them exploiting information gathered from the execution of a program on a specific hardware target. Relevant information to be profiled depends on the specific application domain and is not only limited to the timing performance of a piece of software but can include other hardware-level concerns (e.g., power consumption) or more functional aspects (e.g., frequency of invocation of a given function or heap and stack usage). The focus of this work is on the profiling requirements from the timing analysis perspective, with special focus on contention analysis in multicore systems. A generic overview of existing approaches for functional and (average) performance profiling of embedded systems is provided in [45, 38].

Hardware profiling solutions are being increasingly considered as a fundamental enabler for timing characterization of critical applications running on multicore embedded platforms. The complexity of those platforms, especially emanating from multicore execution, in combination with increasingly richer functionalities, poses a challenge to conventional timing analysis approaches [51, 42]. However, despite the increasing standardization effort [20, 2], advanced non-intrusive hardware monitoring support is not always available in embedded targets.

Approaches based on (some form of) software and hardware profiling are particularly appealing from the industrial standpoint as they closely resemble the consolidated functional verification process. A number of industrial-quality measurement-based analysis tools are commercially available and have achieved a good level of penetration in the respective industrial domains [43, 25, 48]. RapiTime, part of the RVS verification suite [43], is mainly

targeting the avionics domain. It builds on software instrumentation at different granularity levels to collect timing information of instrumented programs and provides basic timing statistics (such as high water-marks, averages, and minima) and an hybrid WCET figure. The T1 timing tool [25] from Gliwa, and TA tool suite [48] from Vector, are instead specifically addressing the automotive domain. Both tools, though at different granularities, allow to capture timing information and reconstruct a timing model of the system under analysis. Currently these tools, however, are essentially exploiting relatively simple timing information (i.e., timestamps) combined with software-level information such as executed branches, components interaction, or operating system events. When it comes to analyzing multicore systems, such basic information can be used to explore specific (multicore) execution scenarios but do not allow to perform a generic contention analysis. While we recognize the value of these approaches, we contend that simple timing information is not enough to capture the complexity of advanced multicore systems [34] and, in our approach, we focus on capturing other relevant events beyond the clock cycles.

More specifically on multicore timing analysis, several interesting approaches have been proposed that rely on hardware profiling information and software means to remove or limit the amount of multicore interference [53, 37, 27, 18, 12]. While [53] exploits application profiling and run-time monitoring to preserve memory bandwidth quotas at the memory controller level, the work in [37] builds on RTOS support to enforce memory usage quotas. Other approaches, instead, use profiling-based methods to derive contention bound [27, 18, 12]. None of these works, however, focus on the fundamental practical aspect of the profiling framework. In this work, instead, we address the relevant aspects of an industrially amenable profiling solution, and report our practical experience in the definition of a profiling solution for the GR712RC.

Advanced hardware tracing support, which has been long advocated by industry [44], is becoming more and more fundamental for the timing verification process. Interesting approaches for non-intrusive hardware tracing support supporting the hardware instrumentation paradigm have been also proposed [5, 13]. Both works are proposing hybrid measurement-based approaches building on advanced DSU interfaces. They both implement an in-band profiling solution, whose limits are overcome in [13] by supporting high-bandwidth, on-line processing of the collected traces by exploiting a computationally powerful FPGA. Still, the main focus of these approaches is to collect basic timing information, and they do not consider the inherent challenges, in terms of intrusiveness, arsing from the collection of comparatively richer information.

6 Conclusions

Hardware profiling and monitoring support are increasingly becoming fundamental to support industrially-viable approaches for the timing characterization of critical functions in embedded multicore systems. In this work, we address the relevant aspects of an industrially amenable profiling solution and we report our practical experience in the definition of a profiling solution for the GR712RC, a reference embedded target for the space domain, where no performance counter support is actually available. We presented the main design challenges of a custom profiling approach and show how it fulfills industrial requirements by providing an efficient profiling framework. We validate and extensively assess our solution on synthetic benchmarks and on a case study from the space domain. Finally, we show how the profiled information can be exploited to derive tight early bounds on the impact of multicore interference on applications' timing behavior.

References

1 Jaume Abella, Carles Hernández, Eduardo Quiñones, Francisco J. Cazorla, Philippa Ryan Conmy, Mikel Azkarate-askasua, Jon Pérez, Enrico Mezzetti, and Tullio Vardanega. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems, SIES 2015, Siegen, Germany, June 8-10, 2015*, pages 39–48. IEEE, 2015. doi:10.1109/SIES.2015.7185039.

2 ARM. ARM® CoreSight©ip. URL: https://www.arm.com/products/system-ip/coresight-debug-trace.

3 ARM. *ARM Advanced Microcontroller Bus Architecture (AMBA) 5 AMBA High-performance Bus (AHB) Protocol Specification*, 2015.

4 Thomas G. Baker. Lessons learned integrating COTS into systems. In John C. Dean and Andrée Gravel, editors, *COTS-Based Software Systems, First International Conference, ICCBSS 2002, Orlando, FL, USA, February 4-6, 2002, Proceedings*, volume 2255 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2002. doi:10.1007/3-540-45588-4_3.

5 Adam Betts, Nicholas Merriam, and Guillem Bernat. Hybrid measurement-based WCET analysis at the source level using object-level traces. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, volume 15 of *OASICS*, pages 54–63. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010. doi:10.4230/OASIcs.WCET.2010.54.

6 Jingyi Bin, Sylvain Girbal, Daniel Gracia Pérez, Arnaud Grasset, and Alain Mérigot. Studying co-running avionic real-time applications on multi-core COTS architectures. In *Embedded Real Time Software and Systems (ERTS2014)*, Toulouse, France, February 2014. URL: https://hal.archives-ouvertes.fr/hal-02271379.

7 Shirley Browne, Jack J. Dongarra, Nathan Garner, George Ho, and Philip Mucci. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, 2000. doi:10.1177/109434200001400303.

8 Dakshina Dasari and Vincent Nélis. An analysis of the impact of bus contention on the WCET in multicores. In Geyong Min, Jia Hu, Lei (Chris) Liu, Laurence Tianruo Yang, Seetharami Seelam, and Laurent Lefèvre, editors, *14th IEEE International Conference on High Performance Computing and Communication & 9th IEEE International Conference on Embedded Software and Systems, HPCC-ICESS 2012, Liverpool, United Kingdom, June 25-27, 2012*, pages 1450–1457. IEEE Computer Society, 2012. doi:10.1109/HPCC.2012.212.

9 Olivier Notebaert (Airbus Defence and Space). On-board software technology trends in space applications (keynote). In *Euromicro Conference on Real-Time Systems*, ECRTS'2018, 2018.

10 Enrique Díaz, Jaume Abella, Enrico Mezzetti, Irune Agirre, Mikel Azkarate-Askasua, Tullio Vardanega, and Francisco J. Cazorla. Mitigating Software-Instrumentation Cache Effects in Measurement-Based Timing Analysis. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 1:1–1:11, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2016.1.

11 Enrique Díaz, Mikel Fernández, Leonidas Kosmidis, Enrico Mezzetti, Carles Hernández, Jaume Abella, and Francisco J. Cazorla. MC2: multicore and cache analysis via deterministic and probabilistic jitter bounding. In *Reliable Software Technologies - Ada-Europe 2017 - 22nd Ada-Europe International Conference on Reliable Software Technologies, Vienna, Austria, June 12-16, 2017, Proceedings*, pages 102–118, 2017. doi:10.1007/978-3-319-60588-3_7.

12 Enrique Díaz, Enrico Mezzetti, Leonidas Kosmidis, Jaume Abella, and Francisco J. Cazorla. Modelling multicore contention on the aurix^{tm} tc27x. In *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*, pages 97:1–97:6, 2018. doi:10.1145/3195970.3196077.

13 Boris Dreyer, Christian Hochberger, Alexander Lange, Simon Wegener, and Alexander Weiss. Continuous non-intrusive hybrid WCET estimation using waypoint graphs. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016,*

July 5, 2016, Toulouse, France, volume 55 of *OASICS*, pages 4:1–4:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/OASIcs.WCET.2016.4.

14 Boris Dreyer, Christian Hochberger, Simon Wegener, and Alexander Weiss. Precise continuous non-intrusive measurement-based execution time estimation. In Francisco J. Cazorla, editor, *15th International Workshop on Worst-Case Execution Time Analysis, WCET 2015, July 7, 2015, Lund, Sweden*, volume 47 of *OASICS*, pages 45–54. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. doi:10.4230/OASIcs.WCET.2015.45.

15 European Aviation Safety Agency. The Use of Multicore Processors in Airborne Systems (MULCORS). Technical Report EASA/2011/6, European Aviation Safety Agency, 2011.

16 European Cooperation for Space Standardization. Standard ECSS-E-40: Space Software: Engineering. Technical report, European Cooperation for Space Standardization, 2013.

17 Federal Aviation Administration, Certification Authorities Software Team (CAST). *CAST-32A Multi-core Processors*, 2016.

18 Gabriel Fernandez, Javier Jalle, Jaume Abella, Eduardo Quiñones, Tullio Vardanega, and Francisco J. Cazorla. Resource usage templates and signatures for COTS multicore processors. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pages 155:1–155:6. ACM, 2015. doi:10.1145/2744769.2744901.

19 Gabriel Fernandez, Javier Jalle, Jaume Abella, Eduardo Quiñones, Tullio Vardanega, and Francisco J. Cazorla. Computing safe contention bounds for multicore resources with round-robin and FIFO arbitration. *IEEE Trans. Computers*, 66(4):586–600, 2017. doi:10.1109/TC.2016.2616307.

20 Nexus 5001 Forum. Nexus 5001 forum. URL: http://www.nexus5001.org.

21 Jeremy Giesen, Pedro Benedicte, Enrico Mezzetti, Jaume Abella, and Francisco J. Cazorla. Modeling contention interference in crossbar-based systems via sequence-aware pairing (SeAP). In *26th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2020, Sydney, Australia, April 21-24, 2020*, 2020.

22 Jeremy Giesen, Enrico Mezzetti, Jaume Abella, Enrique Fernández, and Francisco J. Cazorla. ePAPI: Performance Application Programming Interface for Embedded Platforms. In Sebastian Altmeyer, editor, *19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019)*, volume 72 of *OpenAccess Series in Informatics (OASIcs)*, pages 3:1–3:13, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2019.3.

23 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 *Proceedings of the 34th Digital Avionics Systems Conference*, DASC'2015, 2015. doi:10.1109/DASC.2015.7311646.

24 Sylvain Girbal, Daniel Gracia Pérez, Jimmy Le Rhun, Madeleine Faugère, Claire Pagetti, and Guy Durrieu. A complete toolchain for an interference-free deployment of avionic applications on multi-core systems. In *Proceedings of the 34th Digital Avionics Systems Conference*, DASC'2015, 2015. doi:10.1109/DASC.2015.7311625.

25 GLIWA GmbH. T1.timing. URL: https://www.gliwa.com/.

26 Reinhold Heckmann and Christian Ferdinand. Verifying safety-critical timing and memory-usage properties of embedded software by abstract interpretation. In *Proceedings of the conference on Design, Automation and Test in Europe*, DATE'05, pages 618–619, 2005. doi:10.1109/DATE.2005.326.

27 Rafia Inam, Mikael Sjödin, and Marcus Jägemar. Bandwidth measurement using performance counters for predictable multicore software. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation, ETFA 2012, Krakow, Poland, September 17-21, 2012*, pages 1–4, 2012. doi:10.1109/ETFA.2012.6489714.

28 International Electrotechnical Commission. IEC 61508: Functional safety of electrical, electronic, or programmable electronic safety-related systems. Technical report, International Electrotechnical Commission, 2011.

29 International Organization for Standardization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.

30 Daniel Kästner, Markus Pister, Simon Wegener, and Christian Ferdinand. TimeWeaver: A Tool for Hybrid Worst-Case Execution Time Analysis. In Sebastian Altmeyer, editor, *19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019)*, volume 72 of *OpenAccess Series in Informatics (OASIcs)*, pages 1:1–1:11, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2019.1.

31 Raimund Kirner and Peter Puschner. Obstacles in worst-case execution time analysis. In *Proceedings of the 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 333–339, 2008. doi:10.1109/ISORC.2008.65.

32 Linux. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.

33 Rapita Trace Box (RTBx) Data Logger). https://www.rapitasystems.com/products/rtbx, 2020.

34 Enrico Mezzetti, Leonidas Kosmidis, Jaume Abella, and Francisco J. Cazorla. High-integrity performance monitoring units in automotive chips for reliable timing V&V. *IEEE Micro*, 38(1):56–65, 2018. doi:10.23919/DATE.2019.8715177.

35 Enrico Mezzetti and Tullio Vardanega. On the industrial fitness of wcet analysis. *11th International Workshop on Worst-Case Execution-Time Analysis*, 2011. doi:10.1109/sies.2015.7185039.

36 Jan Nowotsch and Michael Paulitsch. Leveraging multi-core computing architectures in avionics. In Cristian Constantinescu and Miguel P. Correia, editors, *2012 Ninth European Dependable Computing Conference, Sibiu, Romania, May 8-11, 2012*, pages 132–143. IEEE Computer Society, 2012. doi:10.1109/EDCC.2012.27.

37 Jan Nowotsch, Michael Paulitsch, Daniel Buhler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *26th Euromicro Conference on Real-Time Systems, ECRTS 2014, Madrid, Spain, July 8-11, 2014*, pages 109–118. IEEE Computer Society, 2014. doi:10.1109/ECRTS.2014.20.

38 Rajendra Patel and Arvind Rajawat. A survey of embedded software profiling methodologies. *CoRR*, abs/1312.2949, 2013. arXiv:1312.2949.

39 Peter P. Puschner and Alan Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2/3):115–128, 2000. doi:10.1023/A:1008119029962.

40 Radio Technical Commission for Aeronautics (RTCA) and EURopean Organisation for Civil Aviation Equipment (EUROCAE). DO-297: Software, electronic, integrated modular avionics (IMA) development guidance and certification considerations. Technical report, EUROCAE, 2005.

41 Petar Radojković et al. On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments. *ACM TACO*, 2012.

42 Jan Reineke. Challenges for timing analysis of multi-core architectures. Workshop on Foundational and Practical Aspects of Resource Analysis, 2017. Invited Talk.

43 Rapita Verification Suite (RVS). https://www.rapitasystems.com/, 2020.

44 Karsten Schmidt, Denny Marx, Jens Harnisch, Albrecht Mayer, Udo Dannebaum, and Herbert Christlbauer. Non-intrusive tracing at first instruction. In *SAE Technical Paper*. SAE International, April 2015. doi:10.4271/2015-01-0176.

45 Jason G. Tong and Mohammed A. S. Khalid. Profiling tools for fpga-based embedded systems: Survey and quantitative comparison. *JCP*, 3(6):1–14, 2008. doi:10.4304/jcp.3.6.1-14.

46 https://www.gaisler.com/doc/gr712rc-usermanual.pdf. *GR712RC User Manual*. Cobham Gaisler.

47 https://www.gaisler.com/doc/grmon2.pdf. *GRMON2 User's Manual*. Cobham Gaisler.

48 Vector Informatik GmbH. TA Tool Suite. URL: https://www.timing-architects.com/.

49 Reinhard Wilhelm. Mixed feelings about mixed criticality (invited paper). In Florian Brandner, editor, *18th International Workshop on Worst-Case Execution Time Analysis, WCET 2018, July 3, 2018, Barcelona, Spain*, volume 63 of *OASICS*, pages 1:1–1:9. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/OASIcs.WCET.2018.1.

50 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3):36:1–36:53, 2008. doi:10.1145/1347375.1347389.

51 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, June 2012. doi:10.1109/SIES.2012.6356583.

52 Wilhelm R. et al. The worst-case execution-time problem overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7:1–53, May 2008.

53 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 2013, Philadelphia, PA, USA, April 9-11, 2013*, pages 55–64, 2013. doi:10.1109/RTAS.2013.6531079.

Discriminative Coherence: Balancing Performance and Latency Bounds in Data-Sharing Multi-Core Real-Time Systems

Mohamed Hassan
McMaster University, Hamilton, Canada
mohamed.hassan@mcmaster.ca

Abstract

Tasks in modern multi-core real-time systems share data and communicate among each other. Nonetheless, the majority of published research in real-time systems either assumes that tasks do not share data or prohibits data sharing by design. Only recently, some works investigated solutions to address this limitation and enable data sharing; however, we find these works to suffer from severe limitations. In particular, approaches that bypass private caches to avoid coherence interference altogether suffer from significant average-case performance degradation. On the other hand, proposed predictable cache coherence protocols increase the worst-case memory latency (WCL) quadratically due to coherence interference. In this paper, by carefully analyzing the scenarios that lead to high coherence interference, we make the following observation. A protocol that distinguishes between non-modifying (read) and modifying (write) memory accesses is key towards reducing the effects of coherence interference on WCL. Accordingly, we propose DISCO, a discriminative coherence solution that capitalizes on this observation to balance average-case performance and WCL. This is achieved by disallowing modified data in private caches, and hence, the significant coherence delays resulting from them are avoided. In addition, DISCO achieves high average performance by allowing tasks to simultaneously read shared data in the private caches. Moreover, if the system supports the distinction between private and shared data, DISCO further improves average performance by allowing for the caching of private data in cores' private caches regardless of whether it is modified or not. Our evaluation shows that DISCO achieves 7.2× lower latency bounds compared to the state-of-the-art predictable coherence protocol. DISCO also achieves up to 11.4× (5.3× on average) better performance than private cache bypassing for the SPLASH-3 benchmarks.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Real-time system architecture; Computer systems organization → Multicore architectures

Keywords and phrases Coherence, Shared Data, Caches, Multi-Core, Real-Time, Memory

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.16

1 Introduction

Demands from modern applications of embedded systems such as those in automotive, avionics, industrial automation and healthcare are shaping the research directions in real-time embedded systems. The high performance demands from these applications ignited the transition from single-core to multi-core real-time systems [37]. In addition, meeting the high data demand was a strong motive behind exploring the adoption of complex memory hierarchies composed of multiple levels of caches [44] and include shared caches [10, 13, 25, 36, 29], shared interconnects [7, 15, 19] as well as off-chip memories [9, 12, 18, 22] instead of the small-sized static on-chip memories found in traditional low-end embedded systems. Despite this large volume of research, one demand from the aforementioned applications is yet to be efficiently addressed: allowing a seamless, predictable, and high performance data sharing among different running tasks. Unfortunately, most prior works in real-time systems do not meet this demand. They either assume tasks are not sharing data or prohibit data sharing by design [6]. This is mainly because data sharing is problematic and can lead to

significant interference delays [3, 11] or even unpredictable behaviors [14] if it is not carefully addressed.

On the other hand, researchers have already realized the importance of enabling data sharing in the context of real-time systems [4, 6, 11, 14, 16]. The common approach followed by these works is to allow data to be shared among tasks but prevent tasks from simultaneously accessing this shared data in an attempt to accommodate for the data sharing demand, while ensuring system predictability. As a result, large interference delays due to this simultaneous access are avoided altogether. This is achieved by either modifying the task-to-core mapping [6], data-aware scheduling [4, 11], or bypassing caches [6, 3, 26]. The main drawback of such approach is that by disallowing simultaneous access to shared data, it can severely deteriorate the system performance. To improve system performance and enable simultaneous access to shared data, [14, 39, 40, 23] propose predictable cache coherence protocols. The problem with these protocols is that they require complex changes to cache controllers and more importantly, they result in a significant increase in the worst-case latency (WCL) upon accessing memory due to coherence interference. In this paper, we propose DISCO: a discriminative coherence solution that addresses the aforementioned drawbacks. Towards this target, we make the following contributions.

1. We exhaustively study all possible access scenarios under coherence protocols to distill the main sources of their large coherence delays. As a result of our study we make the following important observation. Significant coherence interference delays that arise from the worst-case scenarios are exclusively due to cache lines being modified in the private caches without an immediate update to the shared cache. The other key observation that DISCO is based on is that the number of memory writes usually represents a small percentage of the total memory requests of applications. We discuss these two observations in details in Section 5.
2. Based on these two observations, we propose DISCO to prohibit the caching of modified data in the cores' private caches. All data modifications are carried out at the shared cache. DISCO intentionally discriminates against write memory requests since they are forced to access the shared cache even if data already exists in the requesting core's private cache, while read requests are allowed to hit in private caches if their data exists; therefore, reads are managed exactly as in traditional coherence approaches deployed in commodity systems. Since all writes are treated equally, we call this version of the proposed solution, DISCO-AllW.
3. To further improve the system performance, we also propose another version of our solution that we call DISCO-SharedW. DISCO-SharedW leverages information about tasks' data (namely, whether data is shared or private). DISCO-SharedW relaxes the constraint of DISCO-AllW by allowing private data to be modified in the private caches. It allows both read and write hits to the private data that is not shared among tasks since it causes no coherence interference. Both DISCO-AllW and DISCO-SharedW can be either implemented as a hardware cache coherence protocol (Section 6) or realized in commodity platforms using already available support on these platforms as we discuss in Section 7.3.3.
4. We conduct a detailed analysis to calculate the WCL incurred by any memory request as well as the total WCL for both DISCO-AllW and DISCO-SharedW (Section 7).
5. We compare both versions of DISCO with the two state-of-the art competitive solutions: PMSI coherence protocol [14] and cache bypassing [26, 3]. Our evaluation uses both the SPLASH-3 [35] parallel data-sharing benchmarks as well as synthetic experiments that are based on the EEMBC-Auto benchmarks [33]. Results in Section 8 show the notorious improvements that DISCO-AllW and DISCO-SharedW achieve compared to both PMSI and cache bypassing. We summarize these results in Table 1.

Table 1 Summary of DISCO improvements over state-of-the-art competitive approaches.

	Per-request WCL		Total WCL				Avg. Performance			
	PMSI	Bypass	PMSI		Bypass		PMSI		Bypass	
	analytical		up to	avg.	up to	avg.	up to	avg.	up to	avg.
DISCO-AllW	7.2×	same	3.3×	2×	65%	42%	100%	12%	2.8×	1.5×
DISCO-SharedW	7.2×	same	6×	3.5×	3.8×	1.5×	3.2×	1.6×	11.4×	5.3×

2 Related Work

With the adoption of multi-core architectures in real-time embedded systems being on the rise, several new challenges face the researchers in these systems. Predictably managing the shared hardware components among different cores (such as interconnects, on-chip caches, and off-chip memories) is one of the biggest challenges. This is because processing elements in multi-core architectures compete to access these resources which results in significant interference in the system. To address this challenge, several recent research efforts aim at providing predictable access to shared interconnect [44, 7, 15, 31, 19], shared caches [42, 36, 43, 10], and shared DRAM [32, 34, 1, 9, 22, 17, 12]. While these efforts successfully address the timing interference problem, the data interference problem is usually overlooked.

Most of the aforementioned solutions adopt the independent-task model, where tasks do not share data. Recently, researchers recognized the importance of data sharing and proposed solutions to handle it [6, 11, 14, 3, 4, 26, 39, 23, 40]. We classify these works into three groups according to their research direction: 1) data-aware scheduling, 2) cache bypassing, and 3) cache coherence protocols.

1) **Data-aware scheduling**. The first direction incorporates data-awareness in the task scheduling to avoid data interference. This is achieved by one of the following means: 1.1) scheduling tasks with shared data such that they never run in parallel [4]; hence, they do not compete for shared data; 1.2) assigning tasks with shared data to the same core [6]; hence, they share the same private cache(s) and do not suffer coherence interference from each other; or 1.3) incorporating run-time performance metrics collected through hardware counters to make data-wise scheduling decisions that mitigate the data sharing effects [11]. This direction enforces new constraints on the system scheduler interference, which deteriorates system schedulability [40]. Unlike these solutions, DISCO does not require any modifications to the system scheduler and coherently handles data sharing in hardware.
2) **Cache bypassing**. A second alternative is cache bypassing, which was first utilized in the context of reducing the shared cache conflict interference [13, 26] but is then used to avoid coherence interference of shared data [6, 3]. If private caches are bypassed, coherence interference is eliminated, but at the expense of degrading average-case performance.
3) **Cache coherence**. The third direction is to make data sharing transparent to the application and the scheduler by handling it completely in hardware using cache coherence protocols [14, 39, 23, 40]. Cache coherence is notoriously the main solution adopted by commercial-of-the-shelf (COTS) multi-core architectures [30, 41]. It has the advantage of enabling data sharing without imposing any restrictions on the real-time scheduler compared to data-aware scheduling solutions. It is also shown to provide high average-case performance compared to both data-aware scheduling and cache bypassing [14, 40]. On the other hand, it suffers a notably high worst-case memory latency due to the introduced coherence interference. For instance, PMSI [14] has a worst-case latency that is quadratic in the number of cores in the system.

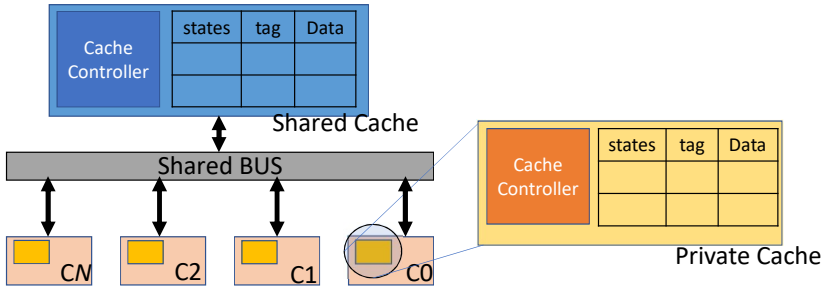


Figure 1 System model.

We discuss both cache bypassing and cache coherence in more details in Section 5 since they are the most related to this work.

3 System Model

We assume the multi-core architecture depicted in Figure 1, where tasks running on this architecture can share data. The proposed solution does not depend on the core architecture and can be seamlessly deployed for in-order or out-of-order cores.

Memory Hierarchy. Each core has its own private cache(s) and all cores share a last-level cache (LLC). LLC is accessed through a shared bus. Cores can also share an off-chip memory. We also assume that timing interference is resolved in the shared cache using partitioning or coloring [10], and in shared main memories using existing solutions orthogonal to this work [12, 8].

Bus Arbitration. Without loss of generalization, we assume that accesses to the shared bus are managed according to a Time Division Multiplexing (TDM) scheme. Solutions proposed in this paper are independent of the deployed arbiter and can be applied to other arbiters as well. However, the timing analysis we perform in Section 6 assumes a TDM bus arbitration. Similar to existing work [14, 39], we set the slot width to accommodate for the one data transfer between private and shared caches in addition to coherence messages. This slot width is denoted as L_{acc}^{miss} since it is incurred if a request misses in the corresponding core's private cache.

Task Scheduling. We do not make any assumption on how the executing tasks are scheduled on cores. The proposed approach is orthogonal to task scheduling and should operate in tandem with any schedule.

4 Cache Coherence Background

When multiple cores accessing the same data, the system has to maintain data correctness. Data correctness is achieved when all cores have access to the most up-to-date data. On the other hand, data incorrectness occurs when a core accesses a stale data that has been already changed in another location in the system (e.g. another core's cache). Modern multi-core systems deploy cache coherence protocols to prevent such situation and preserve data correctness. The Modified-Shared-Invalid (MSI) is considered the baseline coherence

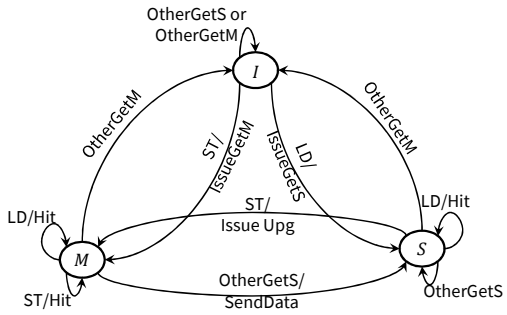


Figure 2 MSI coherence states, messages, and transitions. Messages observed by the core on the bus from other cores are indicated as *Other* (e.g. *OtherGetS*).

protocol [38], where many of the commercial-off-the-shelf architectures adopt protocols that inherit its three fundamental states: Modified (M), Shared (S), and Invalid (I) such as the MESIF protocl deployed in Intel's i7 and the MOESI protocol deployed in AMD's Opteron [20]. Therefore, we use it as a mean to explain the basics of a coherence protocol.

Figure 2 depicts the three states of MSI as well as all possible transitions between them. If a cache line does not exist in the private cache or its data is stale, its state will be I. The S state indicates that the data of this cache line is valid and is not modified, while the M state indicates that the data of this cache line is valid and modified. Therefore, multiple cores can share a cache line in their private cache in the S state, while only one core can have a cache line in the M state. All other cores in this case will have this line in the I state. If a core has a load/read request to a cache line in the I state, the private cache controller of this core (or for simplicity we refer to this throughout the paper as just the core) issues a $GetS$ coherence message on the bus to inform all other cores and the shared cache about this request. Once the core receives the requested data, it moves to the S state. Similarly, if a core has a write request to a cache line in the I state, it issues a $GetM$ message on the bus and moves to the M state once data is received. The core is not required to take any action upon observing messages of other cores to a cache line that it has in the I state. Read requests to a cache line in the S state are hits and no message is broadcasted on the bus. In contrast, write requests to a cache line in the S state has to broadcast an Upg message on the bus to ask other cores to invalidate their local copies in their private caches since it is going to modify it. A core takes no action upon receiving an $OtherGetS$ from another core to a line that it has in the S state since multiple cores can simultaneously read the same cache line. Read and write requests to a cache line in the M state are hits and no message is broadcasted on the bus. If the core observes an $OtherGetS$ on the bus from another core requesting to read a cache line that it has in the M state, it sends the modified data to the requesting core and/or the shared memory and moves to the S state.

5 Motivation

5.1 Performance Gains of Cache Coherence

PMSI [14] provides high performance gains compared to other approaches such as shared-data aware scheduling and private cache bypassing by deploying cache coherence to orchestrate accesses to share data. In Figure 3, we show the execution time of both PMSI and bypassing

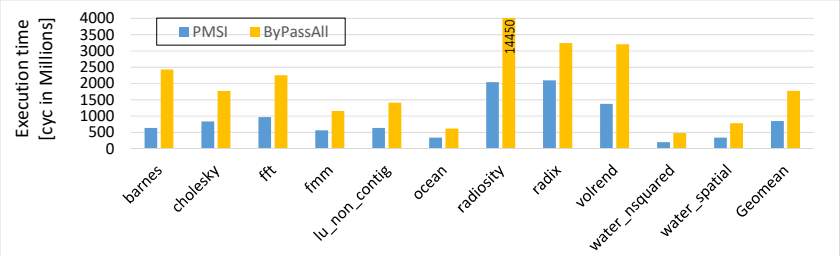


Figure 3 Execution time.

private caches entirely (or simply bypassing[1]). The applications used in this experiment are from the SPLASH3 benchmark suite [35], and the experimental setup is discussed in details in Section 8. As Figure 3 illustrates, PMSI outperforms bypassing for all benchmarks. Performance improvements reach up to 3.7× (barnes) and 7× (radiosity) with a geometric mean performance improvement across all benchmarks of 2×. This clearly represents promising results that motivate us to investigate cache coherence in the context of real-time systems.

5.2 Per-Request WCL

Despite its average-case performance gains, PMSI suffers from large worst-case delays due to the introduced coherence interference. For instance, with bypassing, all cores pay the price of a shared cache access delay once granted access to the bus by the arbiter, regardless of the access pattern of other cores. This results in an access latency of one TDM slot, which we denoted as L_{acc}^{miss} in Section 3 with no coherence latency at all. In addition to this access latency, for a system with N cores and a fair TDM arbiter, a request can suffer an arbitration latency up to one full TDM period or $N \cdot L_{acc}^{miss}$. Table 2 summarizes these worst-case values. This is notably lower than the worst-case scenario under PMSI, where all cores compete to simultaneously access the same shared cache line. As Table 2 illustrates, in addition to the access latency and arbitration latency that is the same as those of bypassing, a memory request under PMSI suffers from a significant worst-case coherence latency. The value of this latency directly follows from [14]. From Table 2, total WCL of both bypassing and PMSI can be calculated as follows.

$$WCL_{perReq}^{PMSI} = 2 \cdot N^2 \cdot L_{acc}^{miss} + 2 \cdot N \cdot L_{acc}^{miss} + L_{acc}^{miss} \tag{1}$$

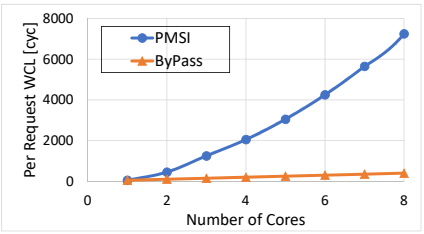
$$WCL_{perReq}^{ByPass} = N \cdot L_{acc}^{miss} + L_{acc}^{miss} \tag{2}$$

Figure 4a delineates this per-request WCL across different number of cores, which shows the significant gap between WCLs of cache coherent solution (PMSI) and bypassing solution due to coherence interference.

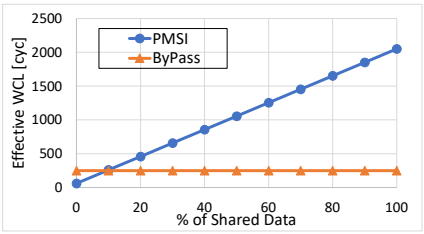
[1] Bypassing throughout this paper refers to skipping the access to the private cache and access directly the shared cache.

Table 2 Worst-case latency components of private cache bypassing and PMSI techniques.

Latency Component	Bypassing	PMSI
Arbitration Latency	$N \cdot L_{acc}^{miss}$	$N \cdot L_{acc}^{miss}$
Coherence Latency	0	$N \cdot (2 \cdot N + 1) \cdot L_{acc}^{miss}$
Access Latency	L_{acc}^{miss}	L_{acc}^{miss}



(a) Per-request WCL across different number of cores.



(b) Effective WCL for various percentage of shared data.

Figure 4 Per-request WCLs (Equations 1 and 2) and effective WCLs (Equation 5.

5.3 Total task's WCL

To bound the task's total Worst-Case Execution Time (WCET), the cumulative WCL over all requests generated by the task under analysis has to be computed. Towards this target, we are interested in calculating the total memory WCL suffered by the total number of memory requests generated by a core during a period of time t, $M(t)$ or simply M [2].

For bypassing, it is straightforward since all requests are serviced from the shared memory, every request can suffer the same WCL that is indicated in Equation 2. Therefore, the total WCL for by passing is computed as:

$$WCL_{tot}^{Bypass} = M \cdot WCL_{perReq}^{Bypass} = M \cdot L_{acc}^{miss} \cdot (N + 1) \tag{3}$$

For PMSI, it is more involved since requests to private (non-shared) cache lines need to be differently handled compared to requests to shared cache lines as the former will not suffer from coherence interference. Considering a partitioned cache hierarchy, where private and shared data are located in separate set such that shared data will not cause any conflict interference to private data, it is safe to assume that the access pattern (private hits and misses) to private cache lines (those not shared with other cores) can be analyzed in isolation and remains the same when the core suffers interference from other $N - 1$ cores. Additionally, with this partitioning, from the task's analysis in isolation (either statically or experimentally), one can compute the number of requests to private cache lines (let it be $M^{private}$), and the number of requests to shared cache lines (M^{shared}) by examining the addresses of memory requests. Moreover, accesses to private cache lines can be further classified into hits and misses to the private cache, which we denote as $M_{hits}^{private}$ and $M_{misses}^{private}$, respectively. Unlike $M^{private}$, it is not possible to statically determine the hits or misses to the shared cache lines since this depends on the access behavior of other cores during run time, which can also access these shared lines. Therefore, the WCL has to be assumed for all accesses to shared lines. Assume the access latency to the core's private cache is

[2] For readability, we drop the usage of t from the remainder of the paper. For instance, we use W instead of $W(t)$ to refer to the number of total writes generated by a core during time t.

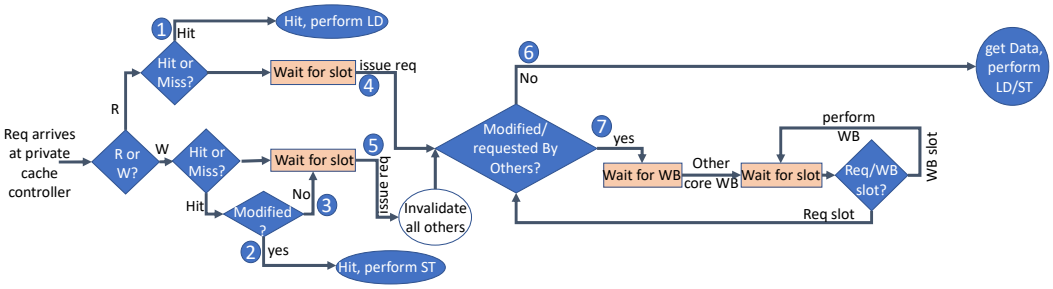


Figure 5 PMSI flow diagram.

$L_{acc}^{private}$ and recall that the WCL to access a shared cache line (which includes coherence interference if exists) is WCL_{perReq}^{PMSI} as calculated by Equation 1. Accordingly, the cumulative total worst-case memory latency suffered by the task, WCL_{tot}, can be computed as:

$$WCL_{tot}^{PMSI} = M_{hits}^{private} \cdot L_{acc}^{private} + M_{misses}^{private} \cdot (N+1) \cdot L_{acc}^{miss} + M^{shared} \cdot WCL_{perReq}^{PMSI} \tag{4}$$

Dividing Equation 4 by the total number of task requests, we get the effective WCL of a single request (WCL_{eff}) as in Equation 5, which can be considered as the average WCL suffered by a single request to the cache.

$$WCL_{eff} = \%M_{hits}^{private} \cdot L_{acc}^{private} + \%M_{misses}^{private} \cdot (N+1) \cdot L_{acc}^{miss} + \%M^{shared} \cdot WCL_{perReq}^{PMSI} \tag{5}$$

To visualize this effect, Figure 4b plots the effective WCL for both bypass and PMSI for different percentage of accesses to shared data. Figure 4b shows that with increased percentage of shared data accesses, the gap between PMSI and bypass significantly increases. The reason for this behavior is that since the WCL_{perReq}^{shared} of PMSI (Equation 1) is much larger than that of bypass (Equation 2), increasing shared data accesses, this latency component will dominate the total WCL.

5.4 Distilling Coherence Effects on WCL

Now, our target is to reduce this high WCL resulting from cache interference. In doing so, we study carefully the effect of coherence across all access scenarios. We find that the high WCL is resulting from a pathological scenario and does not apply to all cases even for accesses to shared lines. This is a key observation and one of the main contributions of this paper; therefore, this subsection will discuss it in detail. We study all possible access scenarios in the existence of coherence, and plot these scenarios in Figure 5. Figure 5 follows the design guidelines of PMSI [14].

5.4.1 Hit Scenario

In case of a read hit (shown as event 1 in Figure 5) or a write hit to an already modified cache line in the private cache (at 2), the core proceeds with the load/store instruction without any arbitration or coherence delays. On the other hand, if the request is a write hit to a non-modified cache line (at 3), the core has to wait for a slot to access the shared bus as writes require to exchange coherence messages to invalidate copies of this line in all other private caches.

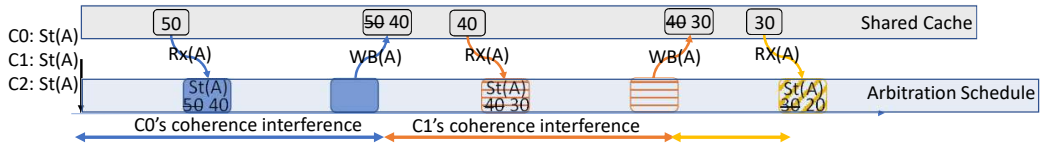


Figure 6 Coherence interference in case of writes for PMSI. C2 is the core under analysis and it has to wait for both C0 and C1 before it gains access to block A.

5.4.2 Miss Scenario

If the request is not available in the core's private cache, the core has to wait for its slot to issue this request on the shared bus. Once the core is granted a slot, it issues the request (④ in Figure 5 in case of a read, and ⑤ for a write). If the requested cache line is not modified in another core's private cache and no write requests are pending to this line, the core receives the data from the shared memory in the same slot and proceed to finish the load/store instruction. This is the scenario highlighted as ⑥ in Figure 5. On the other hand, if one of these two conditions is not satisfied, the core has to wait for all pending writes (if exist) to same line to finish first and the data to be updated in the shared memory (through a write back by another core) before it can obtain the requested line in its private cache. This is the scenario at ⑦ in Figure 5. As Figure 5 illustrates, the scenario at ⑦ is the one that triggers coherence interference and causes the largest delays.

5.4.3 Worst-Case Scenario

Based on this discussion, the pathological worst-case scenario is to assume that all cores in the system simultaneously ask to modify the same cache line. Accordingly, the request under analysis in the worst case has to wait for all other cores, where each core performs its access to obtain the cache line in its private cache, modifies it (performs the store instruction), and writes the new data back to the shared memory. This requires two memory transfers per each core of the other $N-1$ cores before the core under analysis is able to proceed with its access. Figure 6 depicts this scenario for a system with three cores, where the core under analysis is $C2$ and it has to wait for store requests from the other two cores before it can issue its own request. Under TDM scheduling, each transfer can wait for a complete TDM period (arbitration effects) before it can gain access to bus, where a TDM period is a function of N. This explains the quadratic effect of coherence interference on WCL.

Bypassing avoids this scenario by directly accessing the shared memory for every memory request, which eliminates the need for write backs, and hence, the coherence interference. However, this comes at the expense of not utilizing the private caches at all making every request suffering the large shared cache access time. This explains the performance degradation of bypassing compared to PMSI as discussed in 5.1. In contrast, we observe that the explained scenario can be avoided if only writes are made visible instantaneously to the shared memory, while reads do not cause any additional coherence interference. In Figure 6, if cores write directly to the shared memory, the resulting effect will be completely equivalent to bypassing independent of how reads are handled. The other important observation is that writes represent usually a small percentage of applications. Our analysis shows that across the SPLASH3 suite, writes represent on average 30% of the memory requests of the application as Figure 7 illustrates. The same observation holds for other benchmarks as well. For instance, we find that the PARSEC benchmark [5] suite and the EEMBC-auto [33] suite have on average 21% and 32% writes per application, respectively.

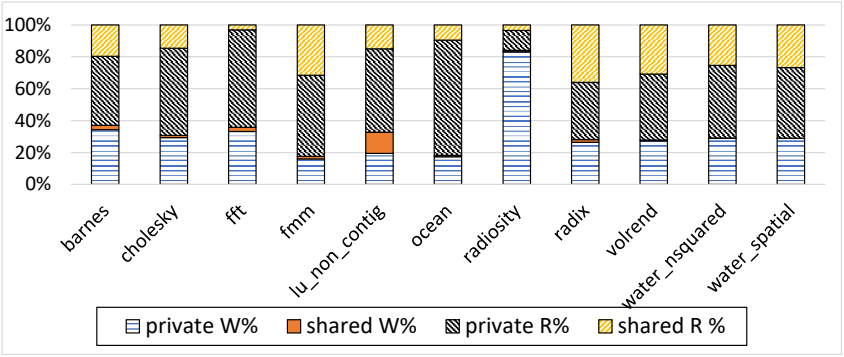


Figure 7 Breakdown of Splash benchmark memory requests.

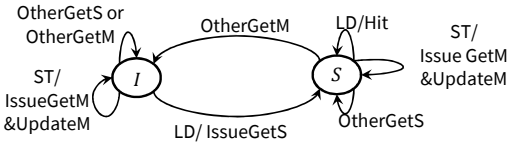
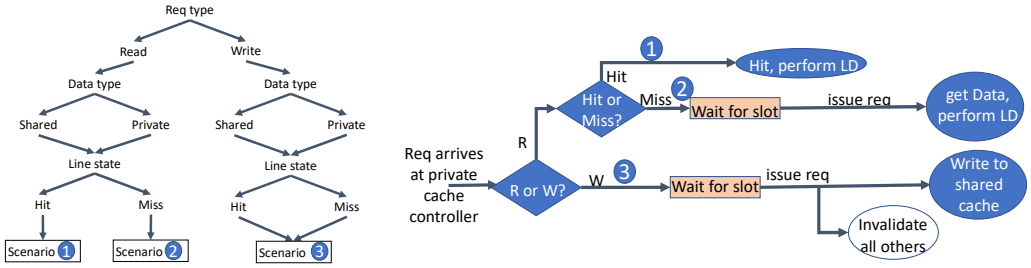


Figure 8 Coherence states of the simple SI protocol adopted by DISCO.

6 Proposed Solution

Motivated by the observations we made in Section 5, we propose DISCO: a discriminative coherence approach. The key idea behind DISCO is to eliminate by design the worst-case scenarios covered in the previous section, and therefore, avoid its significant coherence delays. On the other hand, DISCO still maintains a high average-case performance by employing coherence and enabling simultaneous access to shared data. These two objectives are achieved by intentionally discriminating write memory requests by forcing them to update the shared memory immediately with any write (new) data from any core. This significantly simplifies the coherence protocol as it eliminates all the transient states needed because of data being updated privately by other cores such as in PMSI [14], and reduces the coherence protocol to the simple SI (or sometimes referred to as VI) protocol [38]. Figure 8 shows the coherence states of this SI protocol. It is worth noting that there are two different ways to realize DISCO, either by 1) implementing this SI protocol in hardware, or 2) achieving the write bypass in already existing platforms through available support in these platforms. For now, we will detail the operation of DISCO, while we explain the required support in existing architectures that can allow for the realization of DISCO without redesigning the coherence protocol in Section 7.3.3. As Figure 8 illustrates, the M state is completely removed since no core will have a cache line modified in its private cache without updating the shared cache.

If a core has a read request to a cache line that is in the I state in its private cache, it issues a GetS message once granted access to the bus and moves to the S state once it receives data from the shared cache. In contrast, if the request is a write to a line in the I state, it modifies this line directly into the shared cache and it does not allocate it to the private cache. Hence, it remains in the I state. Although allocating the cache line in the private cache might improve average performance by potentially allowing future read hits to this line, it requires an additional data transfer from the shared cache to the core, which increases the WCL. In particular, the slot width of the shared bus arbiter has to accommodate for two memory transfers instead of one. As a result, we choose not to allocate the line in this case



(a) Handling of different access scenarios.

(b) Flowchart of operations.

Figure 9 Proposed DISCO-AllW coherence approach.

to improve WCL. As explained in Section 4, a core with a cache line in the I state makes no change to its state as a response to events on this line by other cores. If a core has a read request to a cache line that is in S state in its private cache, it is a hit and no change in the state is required. However, if the core has a write request to a line that is in the S state, it has to issue a $GetM$ message on the bus to invalidate copies of this line in all other private caches and perform the write to its private cache as well as to the shared cache to keep it updated. If while in the S state, a core observes an $OtherGetS$ message on the bus, it remains in the S state since the other core is requesting this line for a read and is not going to modify it. Contrarily, if the core observes an $OtherGetM$ message to a line it has in the S state, it has to invalidate its copy since the data is going to be updated by the other core.

Leveraging this simple protocol, DISCO eliminates the large coherence delays due to write requests that modify data in private caches of cores while not being reflected on the shared memory. In other words, the long path in Figure 5 due to the modified/requested by others condition (the scenario at (7)) is eliminated since this condition will be always false (no cache line will be modified in a core's private cache). Figure 9 illustrates the operation of DISCO. Since all writes are handled equally, we denote this approach as DISCO-AllW.

6.1 DISCO-AllW: Discriminative Coherence for All Cache Lines

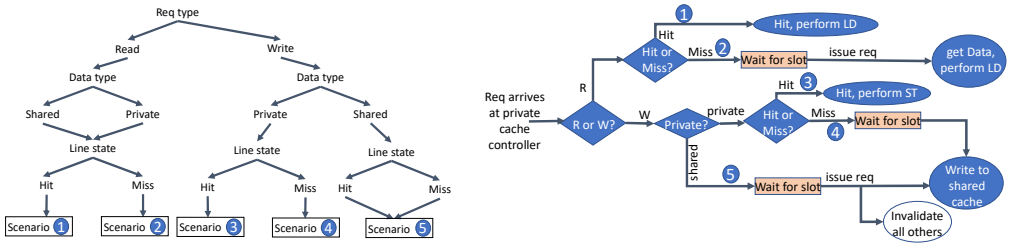
A request to a cache line can be classified according to three factors.

1. **Request type.** With regard to the the instruction type, a request can be either a *read* (e.g. from a load instruction), or a *write* (e.g. from a store instruction).
2. **Data type.** This is related to the nature of the data stored in this cache line, it can be one of two possibilities: a *private* cache line (only accessed by the current task), or a *shared* cache line across tasks.
3. **Line state.** Finally, a request can be either a *hit* if the requested data exists (and is valid) in the private cache of the requesting core or a *miss* otherwise.

This results in a total of eight possibilities for any such request. For example, one request can be a read hit to a private cache line, while another request could be a write hit to a shared cache line, etc. DISCO-AllW operates on these cases based on the following four rules:

▶ **Rule 1.** ***Operating Rules of*** DISCO-AllW**.**

(A) *It does not distinguish between shared and private lines, both are treated equally.*
(B) *It treats all writes equally by sending them to the shared cache.*
(C) *Read hits are allowed and can proceed without requesting an access to the shared bus.*
(D) *Read misses have to wait for an access to the shared bus to obtain data from the shared cache.*



(a) Handling of different access scenarios. **(b)** Flowchart of operations.

Figure 10 Proposed DISCO-SharedW coherence approach.

Based on these rules, the aforementioned eight cases are reduced to only three scenarios under DISCO-AllW as illustrated in Figure 9a. Figure 9b depicts a flow chart for the operation of DISCO-AllW in all these three scenarios.

1. **Scenario 1: A Read Hit in the Private Cache.** Read hits are allowed immediately in the private caches and operate similar to traditional coherence protocols. This is because they do not require an access to the shared bus and do not result in any modification in the coherence state of the requested cache line.
2. **Scenario 2: A Read Miss in the Private Cache.** If the requested line is a read miss in the private cache, it has to be requested from the shared memory. Thus, the core has to wait for its slot and then issue its request on the shared bus. Since all writes are reflected immediately in the shared cache, the shared cache will always have the up-to-date data. Accordingly, the core will receive its requested line in the same slot and perform its load operation.
3. **Scenario 3: A Write Request.** As Figure 9b shows, any write request has to wait for an access slot to the shared bus to update the shared cache with the new data. In addition, all copies of the requested cache line in other cores' private caches have to be invalidated (since it is now outdated).

6.2 DISCO-SharedW: Discriminative Coherence for Shared Lines Only

DISCO-AllW operation does not make any assumption about the cache lines; in particular, it does not rely on the knowledge of which lines are shared, which facilitates its adoption if such knowledge is not made available during execution. On the other hand, if such knowledge is available, we can improve the performance of the solution. This can be done by leveraging the fact that if a line is private for a task (and hence not shared among tasks), DISCO can safely allow write hits to this line without worrying about coherence interference. In doing so, we introduce another alternative to DISCO-AllW that we call DISCO-SharedW. Figure 10 illustrates the details of DISCO-SharedW, which operates according to the following rules.

▶ **Rule 2.** ***Operating Rules of*** DISCO-SharedW**.**

(A) *Read hits are allowed to both private and shared lines and can proceed without requesting an access to the shared bus.*

(B) *Read misses have to wait for an access to the shared bus to obtain data from the shared cache.*

(C) *Write hits are allowed only to private lines that are not shared with other tasks. Those hits can proceed without requesting an access to the shared bus.*

(D) *Write misses to private lines has to wait for an access slot to the bus since it has to be requested from shared memory.*

(E) *Writes to shared lines have to go the shared cache.*

According to these rules, DISCO-SharedW handles reads exactly as in DISCO-AllW. On the other hand, writes are handled differently based on whether they are targeting a private or a shared cache line. This results in the following five scenarios of any memory request as depicted in Figure 10a.

1. **Scenario ❶: A Read Hit in the Private Cache.**
2. **Scenario ❷: A Read Miss in the Private Cache.**
   As illustrated in Figure 10 (compared to Figure 9), DISCO-SharedW handles Scenarios ❶ and ❷ exactly the same as DISCO-AllW.
3. **Scenario ❸: A write hit in the private cache for a private cache line.** Write hits to private lines are allowed based on Rule 2(C) and they execute immediately without the need to exchange any coherence messages.
4. **Scenario ❹: A write miss in the private cache for a private cache line.** Write misses to private cache lines are managed according to Rule 2(D) and they have to wait for an access slot to be sent to the shared memory.
5. **Scenario ❺: A write to a shared cache line.** Write hits to shared cache lines are still not allowed. This is necessary to avoid the high coherence delays resulting from it. Therefore, a write request to a shared cache line has to wait for an access slot to the shared bus since it has to update the shared cache (Rule 2(E)).

7 Worst-Case Latency

In this section, we derive both the per-request as well as the total WCL for a system that deploys DISCO to manage shared data in its cache hierarchy.

7.1 Per-Request Worst-Case Latency

From the previous discussion, any memory request in either DISCO-AllW or DISCO-SharedW requires in the worst case only one memory transfer between the shared cache and the requesting core. For read requests, the shared cache sends data to the core, while for writes, the core sends updated data to the shared cache. Consequently, any memory request suffers only access and arbitration latencies. Excessive coherence delays because of multiple data transfers as discussed in Section 5 are completely eliminated.

▶ **Lemma 1.** *A request to a cache hierarchy deploying either versions of* DISCO *encounters a latency that is at most:*

$$WCL_{perReq}^{DISCO} = WCL_{perReq}^{DISCO\text{-}AllW} = WCL_{perReq}^{DISCO\text{-}SharedW} = N \cdot L_{acc}^{miss} + L_{acc}^{miss} \tag{6}$$

Proof. The proof directly follows from the fact that under the deployed TDM arbitration, a core in the worst case has to wait for all other cores before it can send an access to the shared bus. Recall that we have N cores and that the slot width is L_{acc}^{miss}. Thus, the worst-case arbitration latency a memory request can suffer is $N \cdot L_{acc}^{miss}$. In addition, as per definition, a request to the shared memory consumes an access latency of L_{acc}^{miss}. ◀

7.2 Total Worst-Case Latency

Although both DISCO-AllW and DISCO-SharedW have the same per-request WCL, DISCO-SharedW improves the total WCL compared to DISCO-AllW. This is because leveraging the distinction between private and shared lines, a core's hit rate for private writes under interference from competing tasks is maintained the same as it is calculated in isolation. This is true since private lines by definition are not shared among tasks, and hence, do not experience interference from requests of tasks running on other cores. It is important to notice that although DISCO-AllW assumes that the knowledge of shared vs private lines is not made available to the hardware online upon execution, we can still use this information offline to derive the total WCL of the task.

▶ **Lemma 2.** *Total worst case memory latency incurred by any task under* DISCO-AllW *can be calculated as:*

$$WCL_{tot}^{DISCO\text{-}AllW} = R_{hits}^{private} \cdot L_{acc}^{private} + (R_{misses}^{private} + R^{shared} + W) \cdot WCL_{perReq}^{DISCO} \tag{7}$$

Proof. Based on the discussion of DISCO-AllW in Section 6.1, we prove Lemma 2 as follows.

Since all writes are treated equally, we denote write requests as simply W. By design, each one of these W requests has to wait for the corresponding core's slot to update the shared cache. Thus, they suffer the worst case scenario in Lemma 1 and each of them can have a WCL of WCL_{perReq}^{DISCO}.

For the read requests to shared lines, denoted as R^{shared}: from Lemma 1, each one of those requests under DISCO-AllW suffers a WCL of WCL_{perReq}^{DISCO}. Finally, since tasks do not interfere on private cache lines as aforementioned, tasks maintain the same hit rate calculated in isolation for read requests to these private lines. Accordingly, the number of read hits and misses to the private lines remain the same. Each one of the $R_{hits}^{private}$ encounters a hit latency of the private cache, $L_{acc}^{private}$, while every read miss has to access the shared cache encountering the scenario of Lemma 1 with a WCL of WCL_{perReq}^{DISCO}. This constructs $WCL_{tot}^{DISCO\text{-}AllW}$ in Equation 7. ◀

▶ **Lemma 3.** *Total worst case memory latency incurred by any task under* DISCO-SharedW *can be calculated as:*

$$WCL_{tot}^{DISCO\text{-}SharedW} = M_{hits}^{private} \cdot L_{acc}^{private} + (M_{misses}^{private} + M^{shared}) \cdot WCL_{perReq}^{DISCO} \tag{8}$$

Proof. In DISCO-SharedW, requests (whether reads or writes) to private lines maintain their hit rate calculated in isolation. This entails any memory request to suffer one of three possible worst case scenarios as follows. Hits to private lines, $M_{hits}^{private} = R_{hits}^{private} + W_{hits}^{private}$, encounter the favorable private cache hit latency $L_{acc}^{private}$. Misses to private lines, $M_{misses}^{private}$, still has to wait for a slot to access the shared cache, and thus, suffers the WCL of WCL_{perReq}^{DISCO} as per Lemma 1. Finally, Requests to shared lines, M^{shared}, also suffer the WCL of WCL_{perReq}^{DISCO} since we cannot decide whether they are misses or hits as they are susceptible to interference from other tasks accessing same lines. Adding the WCL of these three scenarios lead to the $WCL_{tot}^{DISCO\text{-}SharedW}$ in Equation 8. ◀

7.3 Other Considerations: A discussion

In this section, we discuss factors that we believe are important to consider for the generalization of the proposed approaches.

7.3.1 On the Derivation of the Total WCL

Private and Shared Data. Equations 4, and 7 – 8, which derive the total WCL for PMSI, DISCO-AllW, and DISCO-SharedW, respectively, make an implicit assumption. They assume no conflict interference between shared and private data in the core's private cache (e.g. L1). As aforementioned, this can be achieved by partitioning the cache such that private and shared data are mapped to different memory spaces (e.g. different sets). Splitting memory address space to private and shared locations is an already existing approach to mitigate interference in the cache hierarchy [27]. However, if such partitioning is not possible, the analysis conducted in isolation to derive the miss and hit rates of a task's private data cannot be used. When running in a contending environment, private cache lines suffer additional conflict interference from shared data as they can be evicted because of the access pattern of shared cache lines that are mapped to the same cache line (under a direct mapped cache) or same set (under a set-associative cache). Therefore, to derive a safe bound, all private lines have to be declared misses. In this case, the total WCL will change to:

$$WCL_{tot}^{PMSI} = M^{private} \cdot (N+1) \cdot L_{acc}^{miss} + M^{shared} \cdot WCL_{perReq}^{PMSI} \tag{9}$$

$$WCL_{tot}^{DISCO} = M \cdot WCL_{perReq}^{DISCO} \tag{10}$$

These two equations also can be used when no information is available about the requests classification (i.e., misses or hits), and therefore, all requests have to be assumed misses. Finally, it is important to highlight that this only affects the calculated analytical total WCL, and it has no effect on the actual operation of different solutions during run time. In other words, it does not affect the average-case performance of PMSI nor DISCO. Per-request WCL also remains as previously calculated in Equations 1 and 6.

Reads and Writes. Another assumption that is made by Equation 7 is that it assumes the knowledge of the number of read and write requests made by the task. This information can be obtained from the task analysis (statically or dynamically) to obtain the number of load and store instructions [24]. Nonetheless, if such information is not available, Equation 10 can be used instead. Again, this does not affect the run-time behavior (and hence, average performance) of DISCO-AllW. It only affects the tightness of its derived bounds.

7.3.2 Effect of Write-backs Due to Replacement in DISCO-SharedW

Since DISCO-SharedW allows write hits to private cache lines, those lines become dirty: they are modified in the core's private cache and are not updated in the shared cache. Hence, those lines need to be written to the shared memory at some point before they are evicted from the private cache due to replacement. The analysis in Section 7 for DISCO-SharedW does not take into account the effect of the write-back of these lines. Here, we discuss possible alternatives to account for this additional delay.

1) At the Request Level. In worst case, a miss request to the private cache initiates a replacement to a dirty cache line. This dirty line has to be written back to the shared memory before fetching the newly requested data. Moreover, this write back has to wait until the requesting core is granted access to the shared bus. As a result, a miss request encounters an additional TDM period due to this write back of the evicted line. This adds $N \cdot L_{acc}^{miss}$ cycles to the WCL_{perReq} in Equation 6 in case of DISCO-SharedW. However, this delay is unnecessarily pessimistic since not every request is going to cause a replacement.

2) At the Task Level. Recall that the number of write-backs are because of writes in the private cache that are not updated at the shared memory. This number is obtainable for the task in isolation by using static analysis or experimental means since private cache is not shared among tasks running on other cores. We refer to the total number of write-backs initiated by a core during a period of time t as WB. For instance, a safe, but rather pessimistic, bound for WB is the total number of issued write requests to private cache lines during the same period t. This is true because write backs are initiated only because of dirty cache lines that are evicted, which in turn is bounded by the total number of writes to private lines. Shared lines cannot be dirty under DISCO-SharedW since similar to DISCO-AllW, they have to be sent directly to the shared memory. As a consequence, $WB = W^{private}$ is a safe bound. We say that this bound can be pessimistic, and hence, can be further tightened since a line can be written multiple times before it is evicted. However, obtaining an accurate value of the maximum number of WB is the concern of the analysis of the task in isolation, and is outside the scope of this paper. Lemma 4 calculates the new total WCL under DISCO-SharedW, while accounting for the delay effect of the write-backs due to cache replacement.

▶ **Lemma 4.** *Total worst case memory latency incurred by any task under* DISCO-SharedW *can be calculated as:*

$$\begin{aligned} WCL_{tot}^{DISCO\text{-}SharedW} &= M_{hits}^{private} \cdot L_{acc}^{private} + (M_{misses}^{private} + M^{shared}) \cdot WCL_{perReq}^{DISCO} \\ &\quad + N \cdot L_{acc}^{miss} \cdot WB \end{aligned} \tag{11}$$

Proof. The proof directly follows from the proof of Lemma 3, while adding the last term to account for the write-backs effect. Since each one of those write-backs requests an access to the shared memory, it can take a maximum of one TDM period to finish. This gives a total delay of $N \cdot L_{acc}^{miss} \cdot WB$, which is the last term in Equation 11. ◀

It is worth noting that even with adding the write-back delays to the total WCL, DISCO-SharedW still provides a lower total WCL compared to DISCO-AllW. Comparing Equation 7 with 11, this is true for two reasons. 1) $WB \leq W^{private}$ as previously explained, and 2) $WCL_{PerReq}^{DISCO} > N \cdot L_{acc}^{miss}$.

7.3.3 Realization in Existing Architectures

One of the main advantages of DISCO is that it significantly simplifies the coherence protocol, while maintaining the average-case benefit of allowing tasks to simultaneously access coherent data. In this section we discuss how to realize DISCO in existing architectures. The first version of DISCO (DISCO-AllW) requires only the ability to bypass private caches for write requests. Contrarily, the second version (DISCO-SharedW) requires, in addition to write bypassing, the ability to distinguish between shared and private lines. We discuss these two requirements below.

1) Selective Bypassing of Writes. Bypassing writes in the private caches can be realized in existing hardware by multiple means. First, a write-through cache achieves exactly the necessary behavior. Many existing architectures enable the user to set caches to operate as write-through caches. For instance, ARM allows the user to switch to write-through caches using a special register named Cache Behavior Override Register [2]. The same register also allows for setting caches as non-write-allocate, which means upon a write request, the cache line is written in the shared cache but is not fetched to the private cache. This is the same behavior we adopted to reduce the WCL. However, it is worth noting that as

explained at the beginning of this section, DISCO can operate correctly even if this capability of non-write-allocate is not provided, albeit with two memory transfers per slot in the worst case. It is important to notice that this register controls the core's private cache only and is independent of the shared cache as implemented in the ARM1176JZ-S processor [2], which is again exactly the same behavior needed for DISCO. Intel processor also provides various control registers to support setting caches to different cache types including write-through [21]; nonetheless, it seems to apply the setting for all cache levels, which forces writes to be sent to the main memory.

2) Isolation Between Shared and Private Data. Isolation between shared and private lines is needed only by the DISCO-SharedW. This can be achieved in existing hardware by placing the shared memory in specific memory regions and then handle requests to every cache line differently in DISCO-SharedW based on the address of this line (whether it is within the boundaries of the shared region or not). This is possible since the aforementioned support about write bypassing can be applied to only specific regions in the memory both in ARM's [2] and Intel's platforms [21]. For instance, DISCO-SharedW can set those shared regions to write-through, while private regions operate normally in a write-back fashion.

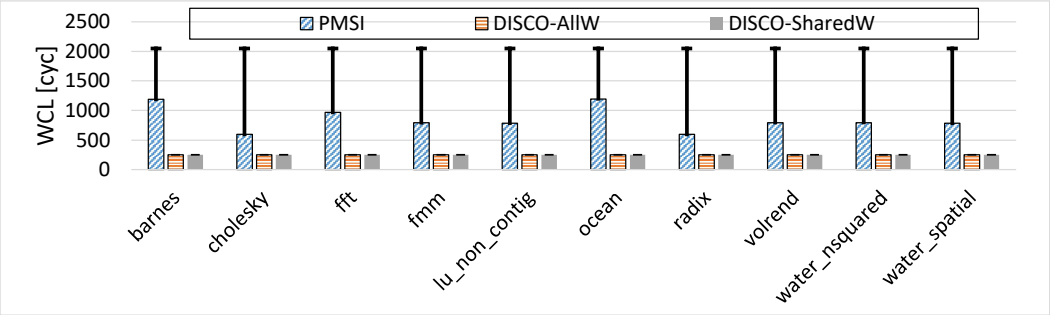
8 Evaluation

To quantitatively evaluate the behavior of DISCO and compare it with state-of-the-art solutions, we simulate the behavior of a multi-core system with in-order pipelines, 8KB direct-mapped L1 per-core private cache, and a $1MB$ L2 shared cache across all cores. Cores are connected to the shared cache using a shared bus. Accesses to this shared bus are managed using a TDM arbiter. The access latency of the L1 cache is 2 cycles, while access latency of the $L2$ is 50 cycles. To eliminate the large delays of off-chip memory access, simialr to existing solutions [14, 23], we set $L2$ to be a perfect cache, i.e. all requests to $L2$ are hits. It is worth noting that this setup has no effect on the evaluated approaches, while it allows to avoid the effect of off-chip memory interference on the total execution time. The DRAM overheads are considered additive to the latencies derived in this work and can be computed using existing work [12].

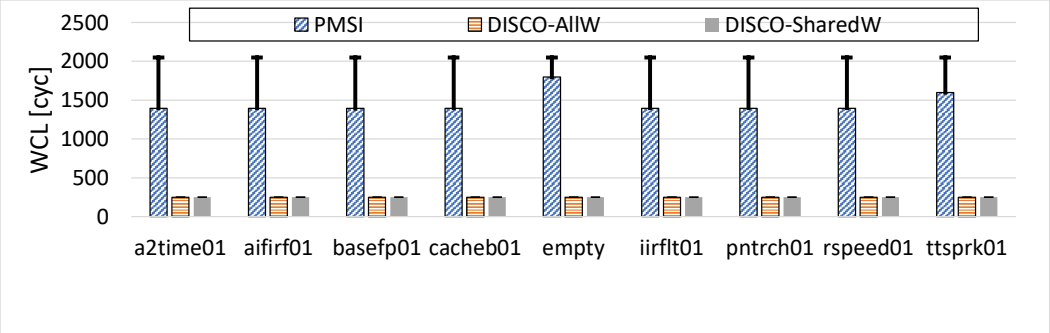
We deploy both versions of DISCO: DISCO-AllW and DISCO-SharedW. In addition, we also implement PMSI [14] and ByPassAll solutions discussed in Section 5. We use benchmarks from the SPLASH3 multi-threaded benchmark suite [33] as well as the EEMBC-Auto suite [33]. The simulation environment integrates with the Intel PIN tools [28] as follows. We run each benchmark through the PIN tool and collect execution traces that we run through the environment. For the SPLASH3 benchmarks, we run them using four threads in four cores (a thread for each core). For the EEMBC benchmarks, we use them to emulate a synthetic scenario that stresses the coherence effect. This is done by executing each of the EEMBC-auto benchmarks through the PINtool and feed the collected trace to each of the four cores in the environment. Doing so, all data is shared across all cores, which signifies the coherence interference.

8.1 Per-Request Worst-Case Latency

Figure 11 delineates the WCL for any request to the cache hierarchy in a four-core system for both SPLASH3 (Figure 11a) and EEMBC (Figure 11b). The figure shows both the analytical WCL bounds (T bars) and the the observed (experimental) WCL (colored solid bars) for both PMSI and the two versions of DISCO. From this experiment, we make the following observations.



(a) Splash3.



(b) EEMBC.

Figure 11 Both analytical (T bars) and experimental (solid bars) per-request WCL.

1) DISCO is able to reduce the analytical WCL by 7.2× compared to PMSI. The analytical WCL of PMSI is 2050 cycles compared to 250 cycles in DISCO. 2) PMSI incurs a large gap between experimental and analytical WCLs. In the SPLASH3 benchmarks (Figure 11a), this gap ranges from 70% (barnes and ocean) and reaches up to 3.4× (*cholesky* and *radix*). This is because PMSI's analytical WCL assumes a pathological worst-case scenario that is hard to construct in real applications as explained in Section 5. Even with the synthetic experiments of EEMBC (Figure 11a), the gap is more than 45% for most benchmarks. On the other hand, DISCO's analytical and experimental WCLs are identical, which indicates the tightness of the derived bounds. DISCO achieves this tightness by deliberately avoiding the large-latency scenarios created by write requests in private caches without updating the shared memory. 3) It is worth noting that DISCO achieves the same WCL as BypassAll solution (not shown in Figure 11), while still allowing read hits to the private caches, which improves both total WCL and average performance as we discuss in the next subsections.

8.2 Total WCL

Figure 12 delineates the total WCL of all the evaluated approaches for the SPLASH-3 benchmarks. To facilitate readability, all results in Figure 12 are normalized to the total WCL of the ByPassAll approach. Recall that the total WCL is the worst-case memory latency that is suffered by a core during a time period t and is calculated in Equations 3, 4, 7, and 11 for ByPassAll, PMSI, DISCO-AllW, and DISCO-SharedW, respectively. Figure 12 introduces several interesting observations.

1) PMSI encounters the largest total WCL. This is due to the quadratic effect of coherence interference as we discussed in details in Section 5. The normalized PMSI's total WCL varies per benchmark based on the percentage of shared data. For instance, the *radix* benchmark

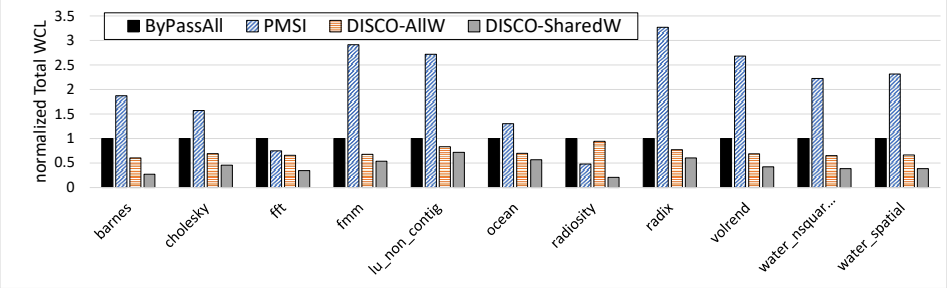


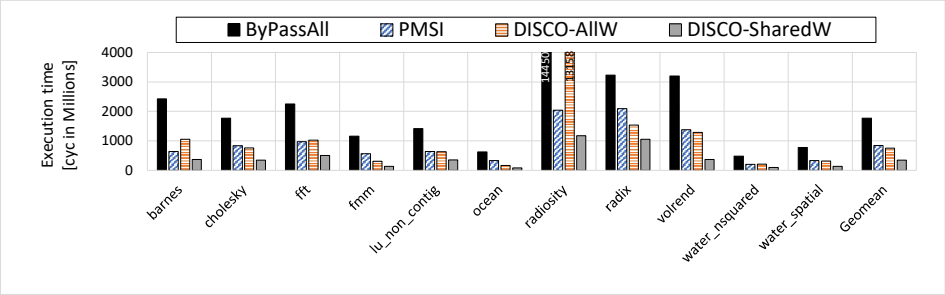
Figure 12 Total worst-case latency of Splash3.

suffers the maximum value of PMSI's total WCL (3.3× ByPassAll's). Investigating the reason for this, we found that *radix* has the maximum percentage of shared data (around 38%). Accordingly, from Equation 4, the term that suffers the maximum latency of WCL^{PMSI}_{perReq} dominates the total WCL. Interestingly, there are cases where PMSI has a lower total WCL than ByPassAll. Namely, this is the case for the *fft* and *radiosity* benchmarks in Figure 12. Analyzing both benchmarks, we found that both benchmarks in contrast to the *radix* benchmark have the maximum percentage of private (non-shared) data: 94% and 96% for *fft* and *radiosity*, respectively. This enables PMSI to leverage hits to this non-shared data, which gives it an advantage over ByPassAll, which forces all requests to go to the shared memory. 2) Compared to PMSI, DISCO-SharedW achieves up to 6x tighter total WCL (*barnes*) and 3.5x on average. DISCO-AllW, on the other hand, has up to 3.3x tighter total WCL (*fmm*) and 1.95x on average. PMSI has a lower total WCL than DISCO-AllW in case of *fft* and *radiosity* benchmarks for the same reasons as in observation 1 because DISCO-AllW does not allow write hits. An extended discussion about the behavior of these two benchmarks is provided in Section 8.4. 3) Although DISCO-AllW and DISCO-SharedW offer the same per-request WCL of ByPassAll as we highlighted in Section 8.1, both proposed approaches provide a tighter total WCL than ByPassAll. The reason for that is that both solutions allow read hits in cores' L1 caches, while DISCO-SharedW also allows write hits to core's private (non-shared) cache lines. This improves the total WCL since as proved in Lemmas 2-4, those hits will not suffer the arbitration latency due to contention on the shared bus. This enables DISCO-AllW to provide up to 65% (*barnes* benchmark) and 42% on average tighter total WCL than ByPassAll. Furthermore, DISCO-SharedW provides up to 3.8× (*radiosity*) and 1.5× on average tighter WCL compared to ByPassAll.

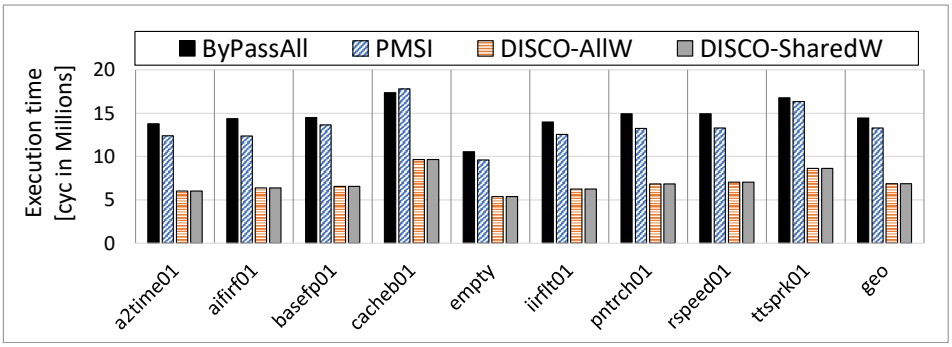
8.3 Average-Case Performance (Execution Time)

Figure 13 depicts the overall execution time for both SPLASH3 (Figure 13a) and EEMBC (Figure 13b) under four different approaches: PMSI, ByPassAll (all requests access L2), and both versions of DISCO. From this experiment, we make the following observations.

1) Compared to ByPassAll, DISCO-AllW improves performance (reduced execution time) by up to 2.8× and 1.5× on average for the SPLASH3 benchmarks. Recall from Section 8.1 that DISCO achieves same WCL as ByPassAll, this verifies the ability of DISCO to balance WCL and performance. 2) DISCO-AllW also has a better overall performance compared to PMSI for SPLASH3 benchmarks (up to 100% and 12% better performance on average). Nonetheless, PMSI has better performance than DISCO-AllW for four benchmarks: *barnes, fft, radiosity*, and *water_nsquared*. We discuss the reasons behind these results in more details later in Section 8.4. 3) Even with the synthetic maximum-sharing scenario of EEMBC experiments



(a) Splash3.



(b) EEMBC.

Figure 13 Execution time.

(Figure 13b), PMSI outperforms ByPassAll, though slightly. In contrast, DISCO shows its maximum performance benefit with increased sharing for two main reasons. a) On the one hand, it does not suffer from the large coherence interference delays incurred by PMSI due to writes. b) On the other hand, it does not suffer from large delays due to forcing all requests to access L2 incurred by ByPassAll as DISCO allows read hits in the private caches. Please note that both versions of DISCO incur exactly same behavior under the EEMBC experiment since all requests (including writes) are shared among all cores. 4) Figure 13a clearly illustrates the benefits of DISCO-SharedW. DISCO-SharedW outperforms all other approaches for all benchmarks: it achieves up to 3.2× and 1.6× on average better performance than PMSI, more than 11× and 5× on average better performance than ByPassAll, and 2× on average better performance than DISCO-AllW. Again, using either version of DISCO depends on the system capabilities. If the system has the capabilities (either in software or hardware) that isolates between shared and unshared data, DISCO-SharedW represents a promising design choice. Contrarily, if the system is not able to distinguish shared data, DISCO-AllW is the best available design choice.

8.4 Average-Case Performance (Average-Case Memory Latency)

To further study the average-case performance behavior of DISCO compared to PMSI, we show the average-case memory latency for SPLASH3 benchmarks in Figure 14. Figure 14 confirms the same behavior observed in the execution time in Figure 13a. 1) DISCO-AllW outperforms PMSI on average by 18%, while PMSI achieves better performance for some benchmarks; *namely, barnes*, and *fft*. 2) DISCO-SharedW, on the other hand, considerably outperforms PMSI and achieves up to 12× and 5.8× on average less average latency. The intuition

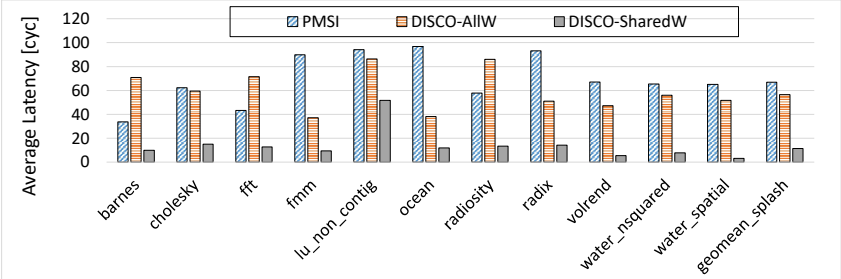


Figure 14 Average latency of Splash3.

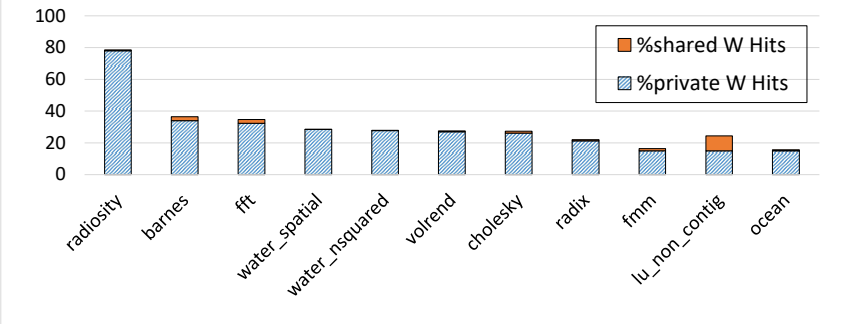


Figure 15 Measured PMSI write hits for Splash3.

behind such behavior of benchmarks where PMSI outperforms DISCO-AllW is that they exhibit larger number of write hits to private cache lines compared to other benchmarks. Because DISCO-AllW forces all writes to access the shared cache, it does not leverage this temporal locality characteristic of such benchmarks; hence, it incurs worse overall performance. In contrast, DISCO-SharedW does leverage this locality by allowing write hits to private cache lines and hence, achieves better performance. To investigate this theory, we deploy performance counters in the simulation environment to count the number of write hits both to private and shared cache lines under PMSI. Figure 15 plots write hit both to shared and private lines as a percentage from the total number of issued requests. Benchmarks are shown in a decreasing order in number of write hits to private lines. Figure 15 confirms our explanation that PMSI achieves better performance for those benchmarks that exhibit high number of write hits to private lines. Nonetheless, for those benchmarks, DISCO-SharedW still achieves better performance than PMSI.

9 Conclusion

Modern real-time systems applications mandate data sharing. In this paper, we propose DISCO: a discriminative coherence protocol that significantly reduces the coherence delays, and hence, provides tighter bounds than existing predictable coherence protocols. DISCO also achieves a high average-case performance by allowing tasks to simultaneously cache and access data in the cores' private caches. DISCO provides the tight latency bounds by eliminating the scenarios that cause high coherence interference under traditional coherence protocols. DISCO can be realized in systems that support write through caches or cache bypassing without any modifications. If such support is not available, it can be realized by modifying the cache controller to adopt the coherence protocol. DISCO achieves up to

7.2× tighter latency bounds than existing predictable coherence protocols, while improving performance by up to 11.4× (5.3× on average) compared to competitive cache bypassing techniques.

References

1 Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: a predictable SDRAM memory controller. In *IEEE/ACM international conference on Hardware/software codesign and system synthesis (CODES+ ISSS)*, 2007.

2 ARM. ARM arm1176jz-s technical reference manual, 2013.

3 Ayoosh Bansal, Jayati Singh, Yifan Hao, Jen-Yang Wen, Renato Mancuso, and Marco Caccamo. Cache where you want! reconciling predictability and coherent caching. *arXiv preprint arXiv:1909.05349*, 2019.

4 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 *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, 2016.

5 Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

6 M. Chisholm, N. Kim, B. C. Ward, N. Otterness, J. H. Anderson, and F. D. Smith. Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems. In *IEEE Real-Time Systems Symposium (RTSS)*, 2016.

7 B. Cilku, B. Frömel, and P. Puschner. A dual-layer bus arbiter for mixed-criticality systems with hypervisors. In *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*, pages 147–151, July 2014. doi:10.1109/INDIN.2014.6945499.

8 Leonardo Ecco and Rolf Ernst. Improved dram timing bounds for real-time dram controllers with read/write bundling. In *2015 IEEE Real-Time Systems Symposium*, pages 53–64. IEEE, 2015.

9 Leonardo Ecco, Sebastian Tobuschat, Selma Saidi, and Rolf Ernst. A mixed critical memory controller using bank privatization and fixed priority scheduling. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTAS)*, 2014.

10 Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. A survey on cache management mechanisms for real-time embedded systems. *ACM Comput. Surv.*, 2015.

11 Giovani Gracioli and Antônio Augusto Fröhlich. On the design and evaluation of a real-time operating system for cache-coherent multicore architectures. *ACM SIGOPS Oper. Syst. Rev.*, 2015.

12 Danlu Guo, Mohamed Hassan, Rodolfo Pellizzoni, and Hiren Patel. A comparative study of predictable dram controllers. *ACM Transactions on Embedded Computing Systems (TECS)*, 2018.

13 D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *IEEE Real-Time Systems Symposium (RTSS)*, 2009.

14 M. Hassan, A. M. Kaushik, and H. Patel. Predictable cache coherence for multi-core real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

15 M. Hassan and H. Patel. Criticality- and requirement-aware bus arbitration for multi-core mixed criticality systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

16 Mohamed Hassan. Heterogeneous mpsocs for mixed-criticality systems: Challenges and opportunities. *IEEE Design & Test*, 2018.

17 Mohamed Hassan and Hiren Patel. A framework for scheduling DRAM accesses for multi-core mixed-time critical systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

18 Mohamed Hassan, Hiren Patel, and Rodolfo Pellizzoni. PMC: A requirement-aware DRAM controller for multicore mixed criticality systems. *ACM Trans. Embed. Comput. Syst.*, 2017.

19 Farouk Hebbache, Mathieu Jan, Florian Brandner, and Laurent Pautet. Shedding the shackles of time-division multiplexing. In *IEEE Real-Time Systems Symposium (RTSS)*, 2018.

20 John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

21 Intel. Intel 64 and IA-32 architectures software developer's manual. *Volume 3A: System Programming Guide, Part*, 1(64), 64.

22 Javier Jalle, Eduardo Quinones, Jaume Abella, Luca Fossati, Marco Zulianello, and Francisco J Cazorla. A dual-criticality memory controller (dcmc): Proposal and evaluation of a space case study. In *IEEE Real-Time Systems Symposium (RTSS)*, 2014.

23 Anirudh M. Kaushik, Paulos Tegegn, Zhuanhao Wu, and Hiren Patel. Carp: A data communication mechanism for multi-core mixed-criticality systems. In *IEEE Real-Time Systems Symposium (RTSS)*, 2019.

24 Sung-Kwan Kim, Sang Lyul Min, and Rhan Ha. Efficient worst case timing analysis of data caching. In *Proceedings Real-Time Technology and Applications*, pages 230–240. IEEE, 1996.

25 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 *IEEE International Conference on Computer Design (ICCD)*, 2014.

26 Benjamin Lesage, Damien Hardy, and Isabelle Puaut. Shared Data Caches Conflicts Reduction for WCET Computation in Multi-Core Architectures. In *International Conference on Real-Time and Network Systems*, 2010.

27 Benjamin Lesage, Isabelle Puaut, and André Seznec. PRETI: Partitioned real-time shared cache for mixed-criticality real-time systems. In *Proceedings of the 20th International Conference on Real-Time and Network Systems (RTNS)*, 2012.

28 Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40(6), pages 190–200. ACM, 2005.

29 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, 2013.

30 MILO MK MARTIN, MARK D HILL, and DANIEL J SORIN. Why on-chip cache coherence is here to stay. *Communications of ACM*, 2012.

31 Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *ACM Annual International Symposium on Computer Architecture (ISCA)*, 2009.

32 Marco Paolieri, Eduardo Quiñones, Fransisco J. Cazorla, and Mateo Valero. An analyzable memory controller for hard real-time CMPs. *Embedded System Letters (ESL)*, 1:86–90, 2009.

33 Jason Poovey et al. Characterization of the EEMBC benchmark suite. *North Carolina State University*, 2007.

34 Jan Reineke, Isaac Liu, Hiren D Patel, Sungjun Kim, and Edward A Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ ISSS)*, 2011.

35 Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Sym-*

posium on Performance Analysis of Systems and Software (ISPASS), pages 101–111. IEEE, 2016.

36 Martin Schoeberl, Wolfgang Puffitsch, and Benedikt Huber. Towards time-predictable data caches for chip-multiprocessors. In *Springer International Workshop on Software Technolgies for Embedded and Ubiquitous Systems (IFIP)*, 2009.

37 Lui Sha, Marco Caccamo, Renato Mancuso, Jung-Eun Kim, Man-Ki Yoon, Rodolfo Pellizzoni, Heechul Yun, Russel Kegley, Dennis Perlman, Greg Arundale, et al. Single core equivalent virtual machines for hard real—time computing on multicore processors, 2014.

38 Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 2011.

39 N. Sritharan, A. M. Kaushik, M. Hassan, and H. Patel. Hourglass: Predictable time-based cache coherence protocol for dual-critical multi-core systems. *CoRR*, 2017. URL: `https://arxiv.org/abs/1706.07568`.

40 Nivedita Sritharan, Anirudh Mohan Kaushik, Mohamed Hassan, and Hiren Patel. Enabling predictable, simultaneous and coherent data sharing in mixed criticality systems. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 433–445, 2019.

41 Per Stenstrom. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 1990.

42 Vivy Suhendra and Tulika Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *ACM Annual Design Automation Conference (DAC)*, 2008.

43 B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson. Making shared caches more predictable on multicore platforms. In *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

44 Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966–978, 2009.

Impact of AS6802 Synchronization Protocol on Time-Triggered and Rate-Constrained Traffic

Anaïs Finzi
TTTech Computertechnik AG, Wien, Austria
anais.finzi@tttech.com

Luxi Zhao
Technical University of Denmark, Lyngby, Denmark
luxzha@dtu.dk

Abstract

TTEthernet is an Ethernet-based synchronized network technology compliant with the AFDX standard. It supports safety-critical applications by defining different traffic classes: Time-Triggered (TT), Rate-Constrained (RC), and Best-Effort traffic. The synchronization is managed through the AS6802 protocol, which defines so-called Protocol Control Frames (PCFs) to synchronize the local clock of each device. In this paper, we analyze the synchronization protocol to assess the impact of the PCFs on TT and RC traffic. We propose a method to decrease the impact of PCFs on TT and a new Network Calculus model to compute RC delay bounds with the influence of both PCF and TT traffic. We finish with a performance evaluation to i) assess the impact of PCFs, ii) show the benefits of our method in terms of reducing the impact of PCFs on TT traffic and iii) prove the necessity of taking the PCF traffic into account to compute correct RC worst-case delays and provide a safe system.

2012 ACM Subject Classification Networks → Network performance analysis

Keywords and phrases AS6802, TTE, Modeling, Performance analysis

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.17

1 Introduction

For safety-critical application domains, proof of the correct temporal behaviour of critical communication flows is required. For example, in the aerospace domain, but also in emerging industrial automation systems, authorities require the proof of correctness as part of the certification process with respect to critical traffic fulfilling end-to-end delay requirements. These requirements have been guaranteed through analysis methods like Network Calculus [8, 7, 6] or the more recent Compositional Performance Analysis [20], for technologies like Avionics Full DupleX (AFDX) [1]. Network Calculus [8] is a well-known mathematical framework based on min-plus algebra that is widely used in the certification process to derive worst-case end-to-end delays for individual asynchronous communication flows.

TTEthernet (SAE AS6802 [11, 16]) is a standard designed to offer strict deterministic guarantees to real-time traffic through the synchronous Time-Triggered (TT) traffic and two traffic classes of asynchronous Rate-Constrained (RC) traffic inherited from the AFDX standard. TTEthernet also considers non-time-sensitive Best-Effort (BE) traffic, and the Protocol Control Frame (PCF) traffic, which is used to keep the local clocks synchronized.

For the TT traffic class, determinism is ensured via an offline communication schedule that enforces a contention-free and precise delivery of critical frames across a switched multi-hop network within defined delay and jitter bounds. For RC traffic, determinism is ensured via a strict shaping and policing of the traffic in the devices of the network.

There are several works proposing methods to compute the RC real-time guarantees within a TTEthernet network, e.g. [15] [19] [22]. However, none consider PCF traffic, which

has the highest priority and can impact the RC delays. Consequently, the interference of PCFs on RC could impact the safety of the systems validated with the existing methods.

The synchronization is a core feature of TTEthernet. While several works have studied the synchronization protocol of TTEthernet to prove its validity [12, 18, 17], to the best of our knowledge, there has not been a study showing the impact of the synchronization protocol on TT and RC traffic.

Hence, the main contributions are threefold, we propose: i) the first formal model of the PCF traffic in Section 6; ii) the study of the PCF impact on TT and a method to mitigate such impact, i.e. *improved TT Sending Windows* in Section 7; iii) the first model to consider the impact of PCF traffic, as well as the impacts of TT and BE traffic, to compute worst-case RC delays in a formal timing analysis in Section 8.

We present the background in Section 2 and related work in Section 3. Then, the problem and our analysis strategy are formulated in Section 4. The first steps of the strategy is presenting our system model and timing analysis in Section 5. Next, we define the PCF model in Section 6. It is followed by analyses of the PCF impact on TT in Section 7 and on RC in Section 8. Finally, a performance evaluation and a conclusion are detailed respectively in Sections 9 and 10.

2 Background

2.1 Network Calculus

The timing analyses detailed in this paper are based on the Network Calculus [10]. This framework is well recognized and has been successfully used for the certification of AFDX networks [8] on the A380 and A350. It is used to compute upper bounds of delay and backlog. These bounds depend on i) the traffic arrival described by the so-called *arrival curve* $\alpha(t)$, i.e. the maximum amount of data that can arrive in any time interval, and on ii) the availability of the crossed node described by the so-called minimum *service curve* $\beta(t)$, i.e. the minimum amount of data that can be sent in any time interval.

▶ **Definition 1** (Arrival curve [10]). *$\alpha(t)$ is an arrival curve for a flow with an input cumulative function $A(t)$, i.e., the number of bits received until time t, iff: $\forall t \geqslant 0, A(t) \leqslant A(t) \otimes \alpha(t)$, with $\forall$ f, g: $f(t) \otimes g(t) = \inf_{0 \leqslant s \leqslant t}\{f(t-s) + g(s)\}$.*

▶ **Definition 2** (Minimum service curve [10]). *The function $\beta(t)$ is the minimum service curve for a data flow with an input cumulative function $A(t)$ and an output cumulative function $A^*(t)$, iff : $\forall t \geqslant 0$, $A^*(t) \geqslant A(t) \otimes \beta(t)$.*

With the definitions of the arrival and service curves, Th. 1 and Th. 2 are given to compute the main performance metrics.

▶ **Theorem 1** (Performance bounds [10]). *Consider a flow f constrained by an arrival curve α crossing a system $\mathcal{S}$ that offers a minimum service curve β. We denote v (resp. h) the maximal vertical (resp. horizontal) distance. The performance bounds at any time t are:*
Backlog: $\forall\ t:\ q(t) \leqslant v(\alpha, \beta)$; Delay: $\forall\ t:\ d(t) \leqslant h(\alpha, \beta)$
Output arrival curve: $\alpha^*(t) = (\alpha \oslash \beta)(t)$, with $\forall$ f, g: $(f \oslash g)(t) = \sup_{s \geqslant 0}\{f(t+s) - g(s)\}$

▶ **Theorem 2** (Left-over service curve - Non-Preemptive Static Priority (NP-SP) multiplexing [2]). *Consider a system with the output capacity C_{out} and m flows crossing it, f_1,f_2,..,f_m. The maximum packet length of f_i is $l_{i,max}$ and f_i is α_i-constrained. The flows are scheduled by the NP-SP policy, where priority of f_i > priority of $f_j \Leftrightarrow i > j$. For each $i \in \{1, .., m\}$, the service curve of f_i is given by: $\beta_i(t) = (C_{out} \cdot t - \sum_{j>i} \alpha_j - \max_{k<i} l_{k,max})_{\uparrow}$, with $\forall$ g: $g_{\uparrow}(t) = \left(\sup_{0 \leqslant s \leqslant t} g(s)\right)^+$, and $\forall$ x: $(x)^+ = \max(0, x)$.*

2.2 TTEthernet

The TTEthernet [9] standard is based on the use of global time synchronization to send TT frames within precise, predefined windows to ensure the lowest contention and delays. A TT flow i is defined by a maximum frame size MFS_i, period P_i and Sending Window (window during which the transmission must start and end), which starts at the offset o_i^p (the earliest time the transmission can start), in each output port p.

TTEthernet inherits the virtual link (VL) concept from the AFDX standard [1]. This concept provides a way to reserve a guaranteed bandwidth for each traffic flow. A VL represents a multicast communication from one sender to one or more receivers. Each VL is characterized by: i) a Bandwidth Allocation Gap (BAG), ranging in powers of 2 from 1 to 128 milliseconds, which represents the minimal inter-arrival time between two consecutive frames in a sender; ii) a Maximal Frame Size (MFS), ranging from 64 to 1518 bytes, which represents the size of the largest frame sent during each BAG iii) the maximum initial jitter J in the sender.

The synchronization uses the PCFs within specific VLs to keep the local clocks of all participants synchronized. These PCF flows have the highest priority in TTEthernet networks, the next priority is used by the TT flows. The next two priorities are used by the RC traffic and are denoted RC_{high} and RC_{low} respectively. The RC classes are compatible with the AFDX standard [1] and use the AFDX concept of VL. The remaining 4 lowest priorities are reserved for BE traffic.

In the next sections we present two parts of TTEthernet: the AS6802 Synchronization Protocol [11] and the TT scheduling characteristics.

2.2.1 AS6802 Synchronization Protocol

The AS6802 protocol defines three types of synchronization roles: synchronization clients (SC), synchronization masters (SM) and compression masters (CM).

The goal of AS6802 is to compensate the clock drifts and keep the difference between two clocks smaller than the synchronization *precision* of the network ϵ [17]. It is important to note that when the network is out of synchronization, no TT frame is sent. Hence, in this paper we will focus on a synchronized network, since we analyse the impact of PCFs on TT and RC.

When the network is synchronized, each SM synchronously sends PCFs to the CMs. Each CM collects the PCFs within an *acceptance window* (2 x ϵ for a fault-tolerance level 0 or 1). After the acceptance window, each CM then triggers the *compression function* computing an average between the estimated local clock drifts. Based on this average, each CM generates a frame to all SM and SC nodes at a fix offset. All SM and SC nodes will receive this PCF and correct their local clocks based on the difference between the time they expected the frame and the actual reception time. This process continues cyclically at a predefined interval called *Integration Cycle* (IC). The length of IC as well as the topology has a direct effect on the achievable synchronization precision.

2.2.2 TT scheduling

The inputs to the schedule generator are the PCF flows, TT flows, the TT flow constraints, the network constraints (see [14]). The outputs of the schedule generator are the offsets of the TT frames in each output port and the selected integration policy for each TT frame in each output port.

The goal of the TT schedule is to maximize the maximum utilization rate of the TT traffic by preventing contention between TT frames and thus minimizing the TT delays. This is achieved by computing TT Sending Windows such as they cannot overlap, and having the devices enforce the constraint that the transmission of a frame must start and end within its Sending Window. However, while this prevents contention due to other TT frames, this cannot prevent the delays due to other priority traffic:

i) higher priority shuffling delays: if frames of higher priority (i.e. PCFs) are queued in the port, the TT frame must wait until they are sent;

ii) lower priority shuffling delays: if a frame of lower priority (i.e. RC or BE) has started its transmission, a TT frame must either wait for its preemption or the end of its transmission. To manage the impact of lower priority traffic, three integration policies have been implemented: shuffling, preemption and guard band [22].

Hence, in addition to the frame transmission duration, the TT Sending Window duration may need to consider higher and lower priority shuffling. Thus, the number of PCFs impacting TT frames can have a large impact on the schedule generation.

3 Related Work

The AS6802 protocol has been studied in several works [12, 18, 17], but they have been focused on the validation of the protocol and the network precision. To the best of our knowledge, this is the first work studying the AS6802 protocol to evaluate the impact of PCF traffic on TT and RC traffic.

Concerning the evaluation of the RC delays, the earliest analysis of RC traffic [15] in TTEthernet assumes pessimistically that all RC frames in an output port of a switch will delay the current RC frame under analysis, and considers that the TT schedule contains periodically alternating phases for TT traffic and for RC traffic. However, realistic schedules do not necessarily contain such periodic phases.

Later, a Network Calculus-based analysis was proposed in [21] to compute the worst-case delays of RC frames by considering a variable size of TT frames and focusing on the shuffling integration policy. However, the protocol uses fixed-sized frames. Additionally, the preemption and guard band integration policies had not been considered in [21].

A recent analysis of RC flows in TTEthernet has been proposed in [19]. The authors use a response-time analysis based on the concept of "busy period" (time interval starting when the frame arrives at the incoming network node, and ending when the frame was transmitted on the dataflow link to the next network node). Their analysis shows that they are able to significantly reduce the pessimism compared to previous approaches. However, their analysis computes the delays for each time instant of the TT schedule, which is time-consuming. As shown in [22], their method does not scale for large problem sizes.

The most recent analysis [22] is based on Network Calculus and the analysis is done for all three integration policies. It shows good performances even compared to [19]. This work [22] considers that all frames are subject to the same integration policy. However, it is not always the case, especially when considering the shuffling and guard band policies. For instance, an implementation of the approach based on Satisfiability Modulo Theories (SMT) proposed in [14, 5] can determine if the guard band is needed and activate it for specific TT flows in specific ports.

More importantly, a common point of all these works [15, 19, 22, 21] is that none of them takes into account the impact of the PCF traffic, which may lead to optimistic RC bounds. The first work to consider PCF traffic is [3]. It focuses on the impact of the interactions

between TT and RC by changing the type of a flow between TT and RC and studying the effects of this change. Additionally, their TT Sending Windows always consider one PCF, which may not always be true, as a port can contain several PCF flows.

On the contrary, this paper considers realistic PCF traffic and studies the impact of the synchronization protocol on TT and RC traffic, while keeping fix flow types for each flow.

4 Preliminary Analysis

The main notations used in this paper are presented in Table 1. Since we are studying data transported in the network links, from now on, MFS will include the preamble, start of frame delimiter and interframe gap.

Table 1 Main notations.

PCF, TT, BE	Protocol Control Frame, Time-Triggered traffic, Best-Effort traffic
RC	Rate-Constrained traffic, composed of two priority classes: RC_{high} and RC_{low}
PTT	Policed Time-Triggered traffic (see Definition 4)
$\alpha_i^p(t)$, $\alpha_i^{*,p}(t)$	Input and output arrival curves of flow i in port p [in bits per second]
$\alpha_I^p(t)$	Input arrival curve of aggregate flow I in port p [in bits per second]
$\beta_I^p(t)$	Minimum service curve of aggregate flow of class I in port p [in bits per second]
C_{out}^p, C_{in}^p	Output and input capacities of port p [in bits per second]
MFS_i	Maximum Frame Size of frame, flow or classes i [in bits]
MFS_I^p	Maximum Frame Size of flows of classes I in port p [in bits]
BAG_i, J_i	Bandwidth Allocation Gap and initial Jitter of flow i [in seconds]
CM, SM, SC	Compression Master, Synchronization Master and Synchronization Client
ϵ	Max. synchronization error (i.e. precision) between two local clocks [in seconds]
IC, IW, SW	Integration Cycle, PCF Integration Window, TT Sending Window [in seconds]
EA, LA	Earliest and latest arrival times [in seconds]
EF, LF	Earliest and latest finish transmission times [in seconds]
o_i^p, P_i	Offset in output port p and period of flow i [in seconds]
Δ_i^p, δ_i^p	Worst-case and best-case delay of flow i in output port p [in seconds]
$\Delta_i^{[src,n]}$, $\delta_i^{[src,n]}$	Worst-case and best-case delay of flow i from source to node n [in seconds]
N_{PCF}^p	Number of PCF flows in port p
$N_{PCF,i\in TT}^p$	Number of PCF impacting TT flow i
τ^p	Hyperperiod of TT∪PCF flows in port p [in seconds] (see Eq. (5))
$N^{p,\tau}$	Total number of frames of classes PCF and TT within a hyperperiod τ^p
$f_g^{p,\tau}$, $f_{bm\oplus g}^{p,\tau}$	g-th frame within a hyperperiod τ^p, and g-th frame after frame $f_{bm}^{p,\tau}$

Our goal is to assess the impact of PCFs on TT and RC traffic and propose a method to reduce this impact.

4.1 Initial Analysis

TT traffic: PCF traffic impacts TT through the higher priority shuffling, which is part of the TT Sending Windows. Hence, by reducing the higher priority shuffling, we should be able to limit the impact of PCFs on TT.

RC traffic: PCF has a higher priority than RC and as such must be taken into account in the RC service curves (see Th. 2), which impact the RC worst-case end-to-end delays.

PCF traffic: PCFs are treated by switches as event-based (RC-like) traffic. However, similarly to TT traffic, when the network is synchronized they are generated at specific points

in time, e.g. PCFs are generated by the SCs at the beginning of each IC. Thus, knowing the best-case and worst-case delays within the network, it is possible to compute the so-called PCF Integration Window (IW), during which a PCF can be present in an output port (in the time reference of the local clock). This means the PCF traffic can only impact TT and RC inside these IWs.

In this paper, we propose to use the characteristics of the PCF traffic to minimize the impact of PCFs on TT and RC.

4.2 Mitigation and analysis strategies

Impact of PCFs on TT traffic: the *current TT Sending Window* computation consider the total number of PCF flows crossing a port, as the higher priority shuffling, regardless of the worst-case sphere of influence of PCFs on TT, i.e. IWs (see Fig. 1(a)). However, as mentioned in Section 4, outside the IWs, no PCF can interfere with a TT frame. Hence, we propose *improved TT Sending Windows* to mitigate the impact of PCFs on TT traffic, by considering the maximum number of PCFs which can impact a TT frame (see Fig. 1(b)).

The improved method is detailed in Section 7.2 by defining the constraints that must be applied when computing a TT schedule (with a heuristic scheduler [13] or SMT [14, 5] for example). The impact of PCFs on TT traffic is assessed in Section 9, using three metrics described in Section 7, i) worst-case end-to-end delay, ii) maximum jitter, and iii) reserved bandwidth.

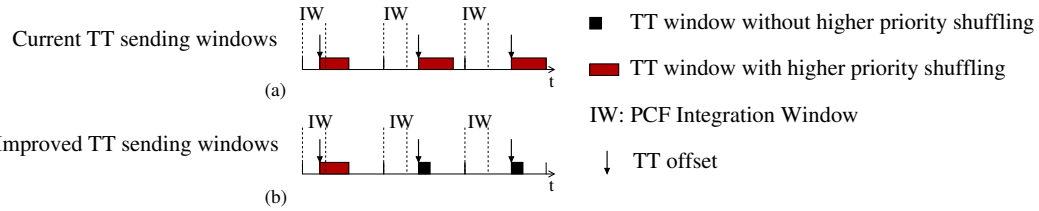


Figure 1 Strategy to mitigate the impact of PCFs on TT traffic.

Impact of PCFs on RC traffic: a direct application of Th. 2 can be done by considering PCF as an event-based (RC-like) traffic and computing the arrival curves separately from RC and TT. However, we have shown that, when the network is synchronized, the arrival times of the PCFs can be computed, similarly to TT traffic. The existing analysis in [22], of the impact on RC of TT traffic and the TT integration policies, is based on the knowledge of the TT offsets. It considers the time a TT frame arrives in an output port (i.e. is released for transmission selection by a switch) to compute a precise arrival curve. In this paper, we propose to extend the analysis to PCFs and TT traffic. Hence, the TT Sending Windows and IWs (detailed in Sections 5.4 and 6 respectively) are used together to compute the precise impact of PCF and TT on RC traffic. Thus, we compute a single arrival curve for both PTT and PCF classes, i.e. $\alpha_{PTT \cup PCF}(t)$ detailed in Section 8, in Th. 11, where PTT is a so-called Policed TT (PTT) class (detailed in Section 8, in Definition 4 and Th. 9), encompassing TT traffic and TT integration policies.

The impact of PCFs on RC traffic will be assessed in Section 9 by comparing our RC end-to-end delay results to the results of implementing the model presented in [22].

First, in the next section we present our system model and delay computation overview, which are needed to detail the computation of IW in Section 6. The IWs are in turn use in the TT mitigation method and and RC analysis in Sections 7 and 8.

5 System model and delay computation overview

5.1 Traffic model

PCF and RC traffic use VLs, so each flow i is defined by a priority, MFS_i, BAG_i and J_i. For example, the leaky bucket arrival curve for such a flow i in the source node src is:

$$\alpha^{src}_{i\in PCF\cup RC}(t) = \begin{cases} 0,\ t \leqslant 0 \\ \frac{MFS_i}{BAG_i} \cdot (t + J_i) + MFS_i,\ t > 0 \end{cases}$$

The above parameters of an RC flow are user-variables, whereas the PCF flows are described in Th. 4 in Section 6.

5.2 Device model

A device (end-system or switch) is characterized by the ports (which can be used as input and output), the forwarding process between the input port and output ports, and the priority queues. The forwarding process in a device n is characterized by known best-case and worst-case forwarding delays, denoted δ^n_{fwd} and Δ^n_{fwd}. For any switch sw, the forwarding delays are between an input port in sw (after the frame has been fully received) and the arrival in an output port in sw. For any end-system es, the forwarding delays are between i) the host in es and an output port in es (source es) or ii) an input port in es and the host in es (destination es).

5.3 Delays between a source and a device for PCF and RC

We denote $\delta_i^{[src,n]}$ (resp. $\Delta_i^{[src,n]}$) the best-case (resp. worst-case) delay of flow $i \in \{PCF, RC_{high}, RC_{low}\}$ between the source node and node n. If n is not a destination node, we consider the delay between the generation of the frame, and its arrival in the output port of n, otherwise the arrival in the destination host. For RC traffic, the source and destination nodes are end-systems. For PCF traffic, source and destination nodes can be either switches or end-systems depending on the synchronization roles.

The delay $\Delta_i^{[src,n]}$ (resp. $\delta_i^{[src,n]}$) is obtained by summing all the sources of worst-case (resp. best-case) delays along the path, which are defined as follows,

1) *delay in an input port q:* it is the reception time a frame in an input port, i.e. amount of time needed for a frame of a flow i to fully arrive in an input port q at a rate C^q_{in}: $\frac{MFS_i}{C^q_{in}}$;
2) *forwarding delays in a node n:* δ^n_{fwd} and Δ^n_{fwd} defined in Section 5.2;
3) *worst-case queuing delay Δ^p_i in an output port p:* we use Th. 1 to compute the worst-case delay bound of a flow i of class $I \in \{PCF, RC_{high}, RC_{low}\}$ in an output port p: $\Delta^p_i = h(\alpha^p_I, \beta^p_I)$, with i) the aggregate input arrival curve $\alpha^p_I(t)$ and ii) the left-over service curve $\beta^p_I(t)$, as follows:
   - **(i)** the input arrival curve $\alpha^p_i(t)$ of flow i of class I, in an output port p of device n, depends on: a) the output arrival curve exiting the preceding output port $p \ominus 1$, denoted $\alpha_i^{*,p\ominus 1}(t)$, which can be calculated using Th. 1; b) the jitter due to the forwarding delays:

$$\alpha^p_i(t) = \begin{cases} \alpha_i^{*,p\ominus 1}(t + \Delta^n_{fwd} - \delta^n_{fwd}),\ t > 0 \\ 0,\ t \leqslant 0 \end{cases} \quad (1)$$

   Then the input arrival curve of the aggregate flow of class I, $\alpha^p_I(t)$, is the sum of the input arrival curves of the flows $i \in I$ crossing p: $\alpha^p_I(t) = \sum_{i\in I, i\in p} \alpha^p_i(t)$

(ii) the service curves are computed with Th. 2. For the RC traffic, the service curves are derived when considering higher priority traffic PCF and TT together. Also, as mentioned in Section 4, the impact of the integration policies will be taken into account in the Policed TT frames (PTT), in Th. 9. Thus, in Section 8, we will consider the aggregate flow of PCF and PTT traffic, to compute the arrival curve $\alpha^p_{PTT\cup PCF}(t)$. Hence, with Th. 2, we obtain the following service curves:

$$\begin{aligned} \beta^p_{RC_{low}}(t) &= \left(C^p_{out}\cdot t - MFS^p_{BE} - \alpha^p_{RC_{high}}(t) - \alpha^p_{PTT\cup PCF}(t)\right)_{\uparrow} \\ \beta^p_{RC_{high}}(t) &= \left(C^p_{out}\cdot t - MFS^p_{BE\cup RC_{low}} - \alpha^p_{PTT\cup PCF}(t)\right)_{\uparrow} \\ \beta^p_{PCF}(t) &= \left(C^p_{out}\cdot t - MFS^p_{TT\cup RC\cup BE}\right)_{\uparrow} \end{aligned} \quad (2)$$

4) *best-case queuing delay δ^p_i in an output port p:* due to the serialization effect [7, 4], the best-case delay of flow i in a node $n \in \{es, sw\}$ is: $\delta^n_{fwd} + \frac{MFS_i}{C^p_{out}}$. When separating the node into input port, forwarding delay and queuing delay, as mentioned before, the delays for each part are considered separately. Hence, the best-case delay in output port p, i.e. with no contention and with C^q_{in} the link capacity shaping in input port q, is: $\delta^p_i = \frac{MFS_i}{C^p_{out}} - \frac{MFS_i}{C^q_{in}}$: In the source nodes, we set $C^q_{in} = +\infty$;

5) *propagation delay:* known latency on physical links.

Thus, PCF delays can now be computed, but $\alpha^p_{PTT\cup PCF}(t)$ must be defined (see Section 8) to compute RC delays.

5.4 Modeling TT Sending Windows

The size of a TT sending window is a very important parameter as it represents the impact of lower (i.e. RC and BE) and higher (i.e. PCF) priorities on TT.

▶ **Definition 3** (TT Sending Window). *A TT Sending Window $SW_{i\in TT}$ of a flow i in an output port p with an output capacity C^p_{out} is given by :*

$$SW^p_{i\in TT} = \left[EA^p_i, LF^p_i\right]$$

$EA^p_{i\in TT} = o^p_{i\in TT}$ the earliest arrival time (i.e. the offset, which is the earliest sending time);
$LF^p_{i\in TT} = EA^p_i + MFS_i/C^p_{out} + LPS^p + HPS^p_i$, i.e. the latest transmission finish time;
with LPS^p the lower priority shuffling defined in Th. 3 and HPS^p_i the higher priority shuffling defined in Th. 8.

To define the TT Sending Windows, we detail the lower priority shuffling in Th. 3 for the different integration policies. The higher priority shuffling will be defined in Th. 8 for the current computation and proposed mitigation method.

▶ **Theorem 3** (Lower priority shuffling). *The lower priority shuffling LPS^p in output port p is such as:*

$$LPS^p = \begin{cases} MFS_{RC\cup BE}/C^p_{out}, \text{ shuffling} \\ PO/C^p_{out}, \text{ preemption, with } PO \text{ the preemption overhead size} \\ 0, \text{ guard band} \end{cases}$$

Proof. With shuffling, a TT frame may be delayed until a lower priority frame finishes its transmission. Hence, a TT frame may experience the full impact of lower priority traffic.

With preemption, an interfering lower priority frame is preempted, and its transmission is restarted from the beginning after the TT frame finished transmitting. Hence, the TT frame may be delayed by the preemption time due to the preemption overhead.

With guard band, a lower priority frame is blocked (postponed) from transmission if a TT frame is scheduled to be sent before the RC frame would complete its transmission. Hence, the TT frame is not impacted by lower priority traffic. ◀

6 Modeling PCF traffic

In order to characterize the PCF traffic, we have studied AS6802 and defined Th. 4.

▶ **Theorem 4** (PCF traffic characteristics). *A PCF flow has the highest priority, and is defined by $MFS_{PCF} = 84$ bytes (incl. preamble, start of frame delimiter and interframe gap), $BAG_{PCF} = IC$, and $J_{PCF} = \epsilon$.*

Proof. The PCF traffic is assigned the highest priority [11], with MFS equals to the minimal Ethernet frame size [11]. Since one PCF is sent per flow per IC, the BAG of a PCF flow is the IC duration [11]. Concerning J_{PCF} for a PCF from either SM or CM, it is equal to ϵ:

PCF from SM to CM: the flows are generated by the SMs at the start of IC. As the maximal time drift between two clocks is lower than or equal to ϵ, the clock correction between two ICs is lower than ϵ. Hence, the initial jitter is equal to ϵ.

PCF from CM to SM and SC: for the flows sent by the CMs, a function ensures that the delays experienced by the incoming PCFs between the SMs and the CMs have no impact on the PCFs sent by the CMs. Hence, the jitter is due to the clock drift correction function. The clock drift is bounded by ϵ, thus the initial jitter is equal to ϵ. ◀

Then, to assess the presence or absence of PCFs at a time t, and the resulting impact on TT Sending Windows, we compute the IWs.

▶ **Theorem 5** (PCF Integration Window). *The PCF Integration Window of flow $i \in PCF$ in output port p of device n (i.e. when a frame of flow i can be present in p) is given by:*

$$IW^{p}_{i \in PCF} = [EA^{p}_{i}, LF^{p}_{i}] \tag{3}$$

with $EA^{p}_{i \in PCF}$ the earliest arrival time, and $LF^{p}_{i \in PCF}$ the latest finish time of transmission, of frame of flow $i \in PCF$ in output port p, such as:

$$\begin{aligned} EA^{p}_{i \in PCF} &= EA^{psrc}_{i} + \delta^{[src,n]}_{i} - \epsilon \\ LF^{p}_{i \in PCF} &= LA^{psrc}_{i} + \Delta^{[src,n]}_{i} + \Delta^{p}_{i} + \epsilon \end{aligned} \tag{4}$$

with: $\Delta^{[src,n]}_{i}$ and $\delta^{[src,n]}_{i}$, defined in Section 5.3; psrc the output port of node src in the path of flow i;

$$EA^{psrc}_{i} = \begin{cases} 0, \ PCF \ from \ SM \\ -\epsilon + MTC + \Delta CM, \ PCF \ from \ CM \end{cases}$$

$$LA^{psrc}_{i} = \begin{cases} 0, \ PCF \ from \ SM \\ +\epsilon + MTC + \Delta CM, \ PCF \ from \ CM \end{cases}$$

ΔCM, the Compression Master delay (known constant) [11]; MTC, the maximum transparent clock, a statically computed worst-case travel time of any PCF in the network.

Proof. The proof is based on i) the known sending times of the PCFs and ii) the study of the PCF delays from the source to device n. The full proof is detailed in the Appendix. ◀

7 Impacts of PCFs on TT class

In this section, we assess the impact of PCFs on TT, with and without mitigation method.

7.1 Assessing the impacts of PCFs on TT

As presented in Section 4, the PCFs impact the TT traffic through the higher priority shuffling. Hence, in Theorem 6 we present the impact of the PCFs on TT. We define the impact of PCFs on a metric as the increase of the metric due to PCFs.

▶ **Theorem 6** (PCF impacts on TT). *The PCF impact on TT is measured using three metrics i) worst-case end-to-end delay, ii) maximum jitter and iii) reserved bandwidth, as follows:*

- **i)**
  - *The maximum impact of PCF on the worst-case end-to-end delay of a flow $i \in TT$ with a path $path_i$ is: $\sum_{p \in path_i} N^p_{PCF} \cdot \frac{MFS_{PCF}}{C^p_{out}}$, where N^p_{PCF} is the number of PCF flows crossing output port p.*
  - *The minimum impact of PCF on the worst-case end-to-end delay of a flow $i \in TT$ is: $\sum_{p \in path_i} HPS^p_i$.*
- **ii)** *The impact of PCF on the maximum jitter of a flow $i \in TT$ in destination node $dest$ is: $HPS^{pdest \ominus 1}_i$, with $pdest \ominus 1$ the output port preceding node $dest$.*
- **iii)** *The impact of the PCFs on the reserved bandwidth of class TT in output port p is: $\sum_{\forall i \in TT, i \in p} \frac{HPS^p_i}{BAG_i}$.*

Proof. In each output port p of each switch n along the path $path_i$ of flow $i \in TT$, the Sending Window is scheduled after the latest arrival time of the frame in the switch (computed in Lemma 7): $o^p_i \geqslant LA^n_{i \in TT} = LF^{p \ominus 1}_i + \Delta^{p \ominus 1, n}_i + \epsilon$.

Hence, the minimum impact of PCFs on the end-to-end delay is obtained by summing the variations of $LA^n_{i \in TT}$ in each device n in the path of flow i, with PCFs versus without PCFs. Only $LF^{p \ominus 1}_i$ is impacted by the PCFs in $HPS^{p \ominus 1}_i$ (see Definition. 3). In the case of the maximum impact on the end-to-end delays, with the mitigation method, the scheduler may select offsets greater than $LA^n_{i \in TT}$ to avoid IWs. This delay is limited in each output port by the full impact of the PCFs, i.e. the transmission time of all the PCFs, which also corresponds to the impact of the PCFs for the current method.

The maximum jitter in node $dest$, $J^{dest}_{i \in TT}$, is computed using Lemma 7. As the ϵ does not impact the relative arrival times in n (since there is only one clock to consider), we obtain:
$J^{dest}_{i \in TT} = LA^{dest}_i - EA^{dest}_i - 2 \cdot \epsilon = HPS^{pdest \ominus 1}_i + LPS^{pdest \ominus 1} + \Delta^{dest}_{fwd} - \delta^{dest}_{fwd}$
with $pdest \ominus 1$ the output port preceding the destination node $dest$.

Hence, the PCFs only impact the jitter in $HPS^{dest \ominus 1}_i$.

The bandwidth reserved for each TT flow i crossing p depends on the Sending Window and is equal to $\frac{LF^p_i - EA^p_i}{BAG_i}$, which depends on the PCFs through HPS^p_i (see Definition 3). ◀

As the impact of the PCFs depends on the TT offsets constrained by the previous nodes along the path of the flow, we present next the impact of the Sending Windows in the preceding output port $p \ominus 1$, on the arrival times of flow i in a succeeding device.

It is important to note that for a TT flow, the arrival time in a device n is different from the arrival time in an output port p of n. The arrival time of a TT frame i in the device n depends on the delays after the frame was sent from the preceding output port. On the contrary, the arrival time in the output port p in device n depends on the offset of the flow i in p, i.e. when the device n releases the frame and makes it available to be selected for transmission in output port p.

▶ **Lemma 7** (Arrival time in a node). *For a flow $i \in TT$ the latest arrival time and the earliest arrival time in $n \in \{es, sw\}$ are:*

$$LA^n_{i \in TT} = LF^{p \ominus 1}_i + \Delta^{p \ominus 1, n}_i + \epsilon$$

$$EA^n_{i \in TT} = EA^{p \ominus 1}_i + \frac{MFS_i}{C^{p \ominus 1}_{out}} + \delta^{p \ominus 1, n}_i - \epsilon$$

with $LF_i^{p\ominus 1}$ *in Definition 3;* $p \ominus 1$ *the output port preceding* n*;* $EA_{i\in TT}^{p\ominus 1} = o_{i\in TT}^{p\ominus 1}$*;* $\Delta_{i\in TT}^{p\ominus 1,n} = \Delta_{prop} + \frac{MFS_{i\in TT}}{C_{out}^{p\ominus 1}} + \Delta_{fwd}^{n}$*;* $\delta_{i\in TT}^{p\ominus 1,n} = \Delta_{prop} + \frac{MFS_{i\in TT}}{C_{out}^{p\ominus 1}} + \delta_{fwd}^{n}$*;* Δ_{prop} *is the link propagation time;* δ_{fwd}^{n} *and* Δ_{fwd}^{n} *defined in Section 5.2.*

Proof. The latest (resp. earliest) arrival time is obtained by summing the delays between the latest (resp. earliest) transmission finish time in the previous node, i.e. $LF_i^{p\ominus 1}$ (resp. $EA_i^{p\ominus 1} + MFS_i/C_{out}^{p\ominus 1}$) and the arrival of the frame in n. Hence, $\Delta_{i\in TT}^{p\ominus 1,n}$ (resp. $\delta_{i\in TT}^{p\ominus 1,n}$) is the sum of the link propagation time, input port delay, forwarding delay. Due to the local clocks, ϵ is added (resp. removed) in the latest (resp. earliest) arrival time. ◀

7.2 Mitigating the impact of PCFs on TT

The method to mitigate the impact on PCFs on TT (improved TT sending windows) proposed in Section 4 is detailed as follows: the goal of this method is to limit the impact of PCFs on the TT within *HPS* (see Definition 3), by computing the number of PCFs that can impact a TT frame i, denoted $N_{PCF,i\in TT}^{p}$.

$N_{PCF,i\in TT}^{p}$ is the number of IWs opened between the earliest start of transmission, i.e. the earliest arrival time, and the latest start of transmission, due to the impact of shuffling. Hence, $N_{PCF,i\in TT}^{p}$ is the number of IWs opened within $[EA_i^p,\ EA_i^p + LPS^p + HPS_i^p]$.

Computing $N_{PCF,i\in TT}^{p}$ depends on HPS_i^p, which in turns depends on $N_{PCF,i\in TT}^{p}$. Thus, with a heuristic scheduler [13], we search for a stable value of $N_{PCF,i\in TT}^{p}$, starting at 0. With SMT [14, 5], we express the relations through first-order logic assertions.

Finally, we present the computation of the higher priority shuffling, used in Th. 6.

▶ **Theorem 8** (Higher priority shuffling). *The higher priority shuffling* HPS_i^p *of a TT flow* i *in output port* p *is such as:*

$$HPS_{i\in TT}^{p} = \begin{cases} N_{PCF}^{p} \cdot \frac{MFS_{PCF}}{C_{out}^{p}}, \text{ current TT window} \\ N_{PCF,i}^{p} \cdot \frac{MFS_{PCF}}{C_{out}^{p}}, \text{ improved TT Sending Windows} \end{cases}$$

with N_{PCF}^{p} *the number of PCF flows crossing* p*, and* $N_{PCF,i\in TT}^{p}$ *the number of IWs opened between earliest and latest start of transmission of TT flow* i*.*

Proof. Currently, the impact of higher priority shuffling on a frame $i \in TT$ in a port p always considers the total number of PCF flows crossing p. However, in Sections 4 and 7.2, we presented the improved TT Sending Windows to mitigate the impact of PCFs. Hence, HPS_i^p is equal to the transmission time of the PCFs impacting a TT frame. ◀

8 Impact of PCFs and TT on RC classes

As shown in Section 4, the impact of PCF, TT flows and TT integration policies, on RC classes is expressed in arrival curve $\alpha_{PCF\cup PTT}(t)$. We start by defining in details the PTT frames before detailing the computation of $\alpha_{PCF\cup PTT}(t)$.

8.1 Policed Time-Triggered frames

We consider the $PCF \cup TT$ flows crossing output port p. The impact of the TT integration policies is taken into account to define Policed TT frames (PTT), in Definition 4.

▶ **Definition 4** (Policed Time-Triggered Frame). *A Policed Time-Triggered (PTT) frame* $f_{g\in PTT}^{p}$ *is composed of a TT frame* $f_{g\in TT}^{p}$ *and its integration policy in output port* p*.*

To define the characteristics of PTT frames and compute the input arrival curve $\alpha_{PCF\cup PTT}(t)$ in Th. 11, we need the periodicity of the $PCF \cup TT$ flows. The behavior of the $PCF \cup TT$ flows crossing an output port p is periodic with a so-called hyperperiod τ^p such as:

$$\tau^p = LCM(P_{i\in\{PCF,TT\},i\in p}) \tag{5}$$

with LCM the least common multiple, $P_{i\in\{PCF,TT\}}$ the periods of the flows i crossing p, and $P_{i\in PCF} = BAG_{PCF}$.

For $\alpha_{PCF\cup PTT}(t)$ and the guard band policy in Th. 9, we need to characterize the earliest finish time and the arrival of the $PCF \cup TT$ flows within τ^p. We denote:

- $N^{p,\tau}$ the number of $PCF \cup TT$ frames in τ^p;
- $f_g^{p,\tau}$ the g-th frame within τ^p.

We define $f_{i(k)}^{p,\tau}$ the k-th frame of flow $i \in \{PCF, TT\}$ within τ^p with $k \in \{0, .., N_i^{p,\tau} - 1\}$, with $N_i^{p,\tau}$ the number of frames of flow $i \in \{PCF, TT\}$ in a hyperperiod τ^p. Hence, $\exists\ g \in \{0, .., N^{p,\tau} - 1\}$ such as $f_{i(k)}^{p,\tau} = f_g^{p,\tau}$. Thus, for TT and PCF, the earliest finish time, the earliest and latest arrival times of frame $f_g^{p,\tau}$ within τ^p are:

$$EA_{g\in PCF\cup TT}^{p,\tau} = EA_{i\in PCF\cup TT}^{p} + k \cdot P_i \tag{6}$$

$$LA_{g\in PCF\cup TT}^{p,\tau} = LA_{i\in PCF\cup TT}^{p} + k \cdot P_i \tag{7}$$

$$EF_{g\in PCF\cup TT}^{p,\tau} = EA_{g}^{p,\tau} + MFS_g/C_{out}^{p} \tag{8}$$

where EA_i^p and LA_i^p are the earliest and latest arrival times of flow i in an output port p.

For PCF flow i, $EA_{i\in PCF}^p$ is defined in Eq. (4), $P_{i\in PCF} = BAG_{PCF}$ and:
$LA_{i\in PCF\cup TT}^p = LF_i^p - \Delta_i^p$, with LF_i^p and Δ_i^p defined in Th. 5 and Section 5.3.

For a TT flow i, as TT frames are released for transmission selection at the offset, so:
$LA_{i\in PCF\cup TT}^p = EA_{i\in PCF\cup TT}^p = o_i^p$

Finally, the characteristic of a PTT frames are detailed in Th.9.

▶ **Theorem 9** (PTT characteristics). *A PTT frame can be characterized by a MFS, a period P, an earliest and latest arrival times, by considering different integration policies:*

$$MFS_{g\in PTT}^{p,\tau} = \begin{cases} MFS_{g\in TT},\ \textit{shuffling} \\ MFS_{g\in TT} + PO,\ \textit{preemption} \\ \Delta GB_{g\in TT}^{p} \cdot C_{out}^{p} + MFS_{g\in TT},\ \textit{guard band} \end{cases}$$

$$P_{g\in PTT}^{p,\tau} = \tau^p$$

$$EA_{g\in PTT}^{p,\tau} = \begin{cases} EA_{g\in TT}^{p,\tau},\ \textit{shuffling \& preemption} \\ EA_{g\in TT}^{p,\tau} - \Delta GB_{g\in TT}^{p},\ \textit{guard band} \end{cases}$$

$$LA_{g\in PTT}^{p,\tau} = EA_{g\in PTT}^{p,\tau}$$

where τ^p is in Eq. (5); PO is the preemption overhead size; $\Delta GB_{g\in TT}^p$ is the guard band duration of the frame $f_{g\in TT}^p$, and $g \ominus 1$ represents the frame preceding $f_{g\in TT}^p$:

$$\Delta GB_{g\in TT}^{p} = \min(\frac{MFS_{RC\cup BE}^{p}}{C_{out}^{p}}, EA_{g\in TT}^{p,\tau} - EF_{g\ominus 1\in TT}^{p,\tau})$$

with $EA_{g\in TT}^{p,\tau}$ the earliest arrival time and $EF_{g\in TT}^{p,\tau}$ the earliest finish time, of a frame g within τ^p, defined in Eq. (6) and Eq. (8).

Proof. Similarly to Th. 3, this theorem is based on the definitions of integration policies [22] and results from a direct application of the impact of integration policies on RC traffic. The full proof is detailed in the Appendix. ◀

8.2 Computation of $\alpha^p_{PCF \cup PTT}(t)$

The behavior of the $PCF \cup PTT$ flows is periodic with a period τ^p (see Eq (5)). However, a duration τ^p can start at the arrival of any frame within τ^p. We call the initial frame the "benchmark" frame. To obtain $\alpha^p_{PCF \cup PTT}(t)$, we compute the arrival curves for all benchmark frames and keep the maximum values (see Eq. (9)). As an arrival curve is due to the minimum amount of time between two arrivals of any arbitrary amount of data, we detail in Lemma 10 the minimum inter-arrival times used in Th. 11.

▶ **Lemma 10** (Minimum inter-arrival time). *We shall consider the minimum time between the arrival of benchmark frame $f^{p,\tau}_{bm} \in \{PCF, PTT\}$ and the arrival of frame $f^{p,\tau}_{bm \oplus g}$*[1]$\in \{PCF, PTT\}$*, with $g \in [0..N^{p,\tau} - 1]$ as follows:*

$$\delta^p_{bm,g} = \begin{cases} 0, \text{ if } LA^{p,\tau}_{bm} \in [EA^{p,\tau}_{bm \oplus g}, LA^{p,\tau}_{bm \oplus g}] \text{ or } LA^{p,\tau}_{bm \oplus g} \in [EA^{p,\tau}_{bm}, LA^{p,\tau}_{bm}] \\ EA^{p,\tau}_{bm \oplus g} - LA^{p,\tau}_{bm}, \text{ otherwise} \end{cases}$$

with $EA^{p,\tau}_{bm \oplus g}$ and $LA^{p,\tau}_{bm \oplus g}$ the arrival times of $f^{p,\tau}_{bm \oplus g}$ (see Eq. (6), Eq. (7) and Th. 9).

Proof. We divide our analysis in two parts: we consider 1) never overlapping arrivals; 2) potentially overlapping arrivals:

1) *never overlapping arrivals:* when considering the PTT or PCFs arriving after the latest arrival time of the PTT or PCF benchmark bm, the minimum time interval occurs when the benchmark arrives closest to the next frame, i.e. we consider the benchmark starting time to be $LA^{p,\tau}_{bm}$ and $\delta^p_{bm,g} = EA^{p,\tau}_{bm \oplus g}$- $LA^{p,\tau}_{bm}$;

2) *potentially overlapping arrivals:* two frames $f^{p,\tau}_{bm}$ and $f^{p,\tau}_{bm \oplus g}$ can overlap iif:

$LA^{p,\tau}_{bm \oplus g} \in [EA^{p,\tau}_{bm}, LA^{p,\tau}_{bm}]$ or $LA^{p,\tau}_{bm} \in [EA^{p,\tau}_{bm \oplus g}, LA^{p,\tau}_{bm \oplus g}]$

The minimum interval is when $f^{p,\tau}_{bm}$ and $f^{p,\tau}_{bm \oplus g}$ arrive at the same time, i.e. $\delta^p_{bm,g} = 0$. ◀

Knowing Lemma 10, the arrival curve of the aggregate PCF and PTT flow is as follows.

▶ **Theorem 11** (Arrival curve of $PCF \cup PTT$ traffic). *The arrival curve of $PCF \cup PTT$ traffic in output port p is:*

$$\alpha^p_{PCF \cup PTT}(t) = \max_{bm \in \{0..N^{p,\tau}-1\}} \left\{ \alpha^{p,bm}_{PCF \cup PTT}(t) \right\} \tag{9}$$

with arrival curve of the $PCF \cup PTT$ flow when considering benchmark frame $f^{p,\tau}_{bm}$ such as:

$$\alpha^{p,bm}_{PCF \cup PTT}(t) = \sum_{g \in \{0..N^{p,\tau}-1\}} \alpha^p_{g \in PCF \cup PTT}(t - \delta^p_{bm,g})$$

where $N^{p,\tau}$ and $\delta^p_{bm,g}$ are defined respectively in Section 8.1 and Lemma 10, and

$$\alpha^p_{g \in PCF \cup PTT}(t) = \begin{cases} MFS^{p,\tau}_{g \in PCF \cup PTT} \cdot \left\lceil \frac{t + LA^p_i - EA^p_i}{\tau^p} \right\rceil, t > 0 \\ 0, t \leqslant 0 \end{cases}$$

is the arrival curve of frame $f^{p,\tau}_g$ of flow $i \in \{PCF, PTT\}$ with a maximum arrival jitter $LA^p_i - EA^p_i$ (defined in Eq. (6), Eq. (7) and Th. 9), a period τ^p (see Eq. (5)), a MFS such as $MFS^{p,\tau}_{g \in PTT}$ is defined in Th. 9 and $MFS^{p,\tau}_{g \in PCF} = MFS_{PCF}$ (see Th. 4).

[1] $f^{p,\tau}_{bm \oplus g}$, the g-th frame after a benchmark frame $f^{p,\tau}_{bm}$

Proof. For $\alpha^{p}_{PCF \cup PTT}(t)$, the proof is identical to the proof of Eq. (12) in [22]. For $\alpha^{p,bm}_{PCF \cup PTT}(t)$, the proof is similar to the proof of Theorem 2 in [22], with two differences, i) because of Lemma 10, we consider here $\delta^{p}_{bm,g}$ instead of the relative offsets between benchmark frame and adjacent frames; ii) because of the IW, we must also consider the arrival jitter $LA^{p}_{i} - EA^{p}_{i} > 0$ in the arrival curve $\alpha^{p}_{g \in PCF \cup PTT}(t)$, rather than $LA^{p}_{i} - EA^{p}_{i} = 0$ like in [22] (due to having only fixed scheduling time instants in [22]). ◀

9 Performance Evaluation

We start by presenting the case study. Then, we present the results of the impact of PCFs on TT and RC traffic.

9.1 Case Study

We study a realistic adaptation of the Orion CEV use case, the 100 Mbps network described in [19, 22] and illustrated in Fig. 2. The TT and RC flows are defined in [22] in Table 2 (MFS without preamble, start of frame delimiter, interframe gap or necessary padding to reach the minimum frame size). We define for the synchronization a level-2 synchronization redundancy. We consider one RC class and the RC initial jitter is null. The PCF flows are as follows: one flow from each CM to the SMs and SCs, one flow from each SM to the CMs. The TT schedule is generated using the SMT-based approach proposed in [14, 5].

We consider linear arrival curves described in Section 5.1 and staircase service curves resulting from Th. 2 with the arrival curves described in Th. 11, as explained in Section 5.3.

We consider the following delays: the compression master delay $\Delta CM = 25.040\ \mu s$, the propagation delay = 5 ns and the forwarding delays in an end-system es: $\Delta^{es}_{fwd} = 2.44\ \mu s$, $\delta^{es}_{fwd} = 1.38\ \mu s$; and in a switch sw: $\Delta^{sw}_{fwd} = 11.45\ \mu s$, $\delta^{sw}_{fwd} = 9.22\ \mu s$.

Concerning the PCFs characteristics, we study two values for IC: an average value, i.e. 10 ms, and a small value, i.e. 1 ms. The corresponding calculated values of the network precision [18] (for the following hardware: ^{TTE} End System A664 Lab and ^{TTE} Switch A664 Lab) are respectively ϵ=4 μs for IC=1 ms and ϵ=6.7 μs for IC=10 ms. To study the impacts of IC and ϵ separately, we define 3 use cases: UC_1={IC=1 ms & ϵ=4 μs}, UC_2={IC=10 ms & ϵ=4 μs}, UC_3={IC=10 ms & ϵ=6.7 μs}. In the case of the impact of PCFs on RC, we consider a fourth use case: UC_4={IC=1 ms & ϵ=5 μs}, where the selected precision is slightly larger than the minimum achievable precision.

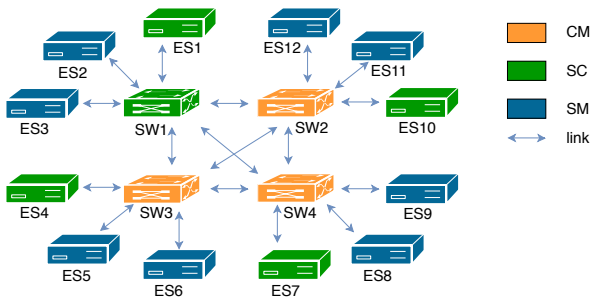


Figure 2 Case study: network description.

9.2 Impact of PCFs on TT

In this section, we compute the end-to-end delay, jitter and reserved bandwidth impacts defined in Section 7.1 without PCFs and with PCFs for both the current method and the proposed mitigation method: *improved TT Sending Window* method, applied to all the TT flows. We also study the impact of our mitigation method on the computation time. We consider two TT integration policies: shuffling and guard band. Depending on the TT flow constraints, a guard band may be automatically activated by the scheduler.

9.2.1 Minimum worst-case end-to-end delay

Our aim is to find the minimum worst-case end-to-end delay achievable using deadline constraints, 1) without PCF; 2) with PCF for the current method; 3) with PCF for the mitigation method, denoted respectively 1) *without*, 2) *current* and 3) *mitigate* in Table 2. For the case *without* PCFs, we consider that a theoretical external synchronization is set up with the defined precision ϵ. The results are in Table 2 for six representative TT flow examples in terms of various MFS, periods and number of potentially interfering PCFs.

Table 2 Minimum worst-case end-to-end delays (μs).

	without PCF		UC_1 IC=1 ms ϵ=4 μs		UC_2 IC=10 ms ϵ=4 μs		UC_3 IC=10 ms ϵ=6.7 μs	
Flow	ϵ=4 μs	ϵ=6.7 μs	current	mitigate	current	mitigate	current	mitigate
TT5	55	63	96	61	96	55	104	63
TT6	223	233	276	276	276	230	287	240
TT9	408	419	460	408	460	408	472	419
TT12	65	73	98	72	98	65	107	73
TT17	385	392	419	419	419	385	426	392
TT20	79	86	132	79	132	79	141	86

First, the current PCF impact increases with the number of PCFs along the path. We note that the deltas between the results of *current* vs. *without* are coherent with Th. 6 and the number of PCFs in the path of the flows (within 1μs due integer rounding margin). For instance, there are 6 PCFs in the output ports of the path of TT5, resulting in a theoretical delta of 40.32 μs, which is coherent with the deltas of 41 μs in Table 2 for all use cases, i.e. 96-55=41 for $\epsilon = 4$ μs (i.e. UC_1 and UC_2), and 104-63=41 for $\epsilon = 6.7$ μs (i.e. UC_3).

Consequently, compared to without PCFs, the current TT end-to-end worst-case delays are increased between 8.67% and 74.5%, i.e. (426-392)/392=8.67% for TT17 with ϵ=6.7 μs, and (96-55)/55=74.5% for TT5 with ϵ=4 μs.

With the mitigation method, the impact of the PCFs on TT depends on the number of interfering PCFs and the possible delays to avoid scheduling a frame inside IWs (see Th. 6).

Firstly, in Table 2, we find that the *mitigate* values are within 1 μs of the values expected with the minimum impact of PCFs defined in Th. 6, due to integer rounding margin. For instance, for UC_1 and TT 5, there is one PCF impacting the TT flow along its path, so the minimum impact of PCF on TT is $1 \cdot 84 \cdot 8/100 = 6.724$ μs, which is coherent with the delta of *mitigate* vs. *without*: 61-55=6 μs. However, in an additional test, i.e. with TT6, UC_4={IC=1 ms & ϵ=5 μs}, we found that the *mitigate* value is +3 μs larger than the value expected with the minimum impact of PCFs, due to a delay to avoid a IW. This brings the PCF impact in the *mitigate* delay between the minimum and maximum impacts of PCFs on worst-case end-to-end delays as defined in Th. 6.

Secondly, the probability of an offset being inside a IW depends on:

1) the hyperperiod of the TT period and the PCF period (i.e. least common multiple between TT period and IC). For example for IC=1 ms and TT20, which has a period of 5 ms, each IW must be considered five times (i.e. one for each IC within the hyperperiod). For TT6, which has a period of 3.125 ms, however, each IW have to be considered 25 times. Hence, it is much more likely that the offset will fall within a IW in this second case. In fact, we can see in Table 2, in UC_1, that TT6 is impacted by all the PCFs, whereas TT20 is not impacted by any PCF;
2) when IC increases, there are less PCFs frames being sent and the intervals between the IWs increases. Thus, there are more possibilities to schedule the TT frames outside the IWs. For example, we can see in Table 2 that with UC_1, TT6 has a delay of 276 μs due to 8 interfering PCFs, whereas with UC_2, TT6 has a delay of 230 μs due to one PCFs;
3) when ϵ increases, it increases the duration of the IWs (see Th. 5). Hence, the intervals between IWs decrease, which decreases the possibility of scheduling TT outside IWs.

Also, ϵ impacts the TT end-to-end delays: when ϵ increases, the latest arrival time increases (see Lemma 7), leading to an increase of the end-to-end delays when the number of interfering PCF is constant, as illustrated in Table 2 for both *without* and *current* results.

Thus, when IC is increased to reduce the number of interfering PCFs, ϵ increases, which increases the end-to-end delays. Sometimes, the reduction of PCFs out-weights the delay increase due to the precision change, e.g. TT6 and TT17, but other times, the increase of delays are larger than the gains due to the PCF impact reduction, e.g. TT5, TT9, TT12.

Finally, we have shown that the mitigation method is effective in reducing the impact of the PCFs on the TT end-to-end delays by limiting the number PCFs interfering with TT. With the mitigation method, the impact of the PCFs varies from 0% to 22.7% with UC_1 and from 0% to 3.0% for UC_3, i.e. (276-223)/223=22.7% for TT6 with UC_1, and (240-233)/233=3.0% for TT6 with UC_3.

This represents a large improvement over the current method, where the impact of PCFs varies between 8.67% and 74.5%.

However, when implementing the mitigation method, IC must be selected carefully to manage the best trade-off between limiting the number of interfering frames and the delays due to ϵ resulting from the selected IC.

9.2.2 Maximum jitter

For flows constrained to have a very low jitter, the guard band is activated in the last output port of the path to remove the jitter due to the lower priority traffic. Thus, the jitter of a TT flow is entirely due to the higher priority shuffling. In our use case, each ES receives 3 PCFs, so the impact of the AS6802 standard on the TT flow jitter is currently equal to the transmission times of 3 PCFs, i.e. 20.164 μs. With the mitigation method, if a low jitter constraint is set, the scheduler will find a solution without PCF interfering frame in the last output port, if it exists. In our use cases, such a solution exists, hence the maximum jitter reduction is up to 100% with the mitigation method compared to the current method.

9.2.3 Reserved bandwidth

By assessing the additional bandwidth reservation due to the PCFs, we can assess the amount of bandwidth that cannot be used by other TT and thus assess the reduction the maximum TT load caused by the PCFs, We have selected the output port with the most TT flows, i.e. $[SW2, ES12]$. Without PCFs, the bandwidth reserved by the TT flows is 38.35 Mbps,

whereas with the current method, the bandwidth reserved is 42.58 Mbps, which represents an increase of 11.04%. With the mitigation method and UC_1={IC=1 ms & ϵ=4 μs}, the bandwidth reserved is 41.80 Mbps, which represents an increase of 9.00% compared to without PCFs. This increase of the reserved bandwidth is reduced down to 1.75% with UC_2={IC=10 ms & ϵ=4 μs}, and down to 2.87% with UC_3={IC=10 ms & ϵ=6.7 μs}. By increasing IC ($UC_1 \rightarrow UC_2$), we decrease the number of PCFs inside the TT period, thus decreasing the likelihood of a PCF interfering with the TT frames, leading to a reduced reserved bandwidth compared to UC_1. By increasing ϵ ($UC_2 \rightarrow UC_3$), the IWs are enlarged and thus the likelihood a PCF interfering with a TT frame increases, leading to an increased reserved bandwidth compared to UC_2.

Hence, with the mitigation and an appropriate IC, we are able to schedule more TT traffic than with the current method. In our case study, compared to the current method, we estimate that for UC_3, the increase of maximum traffic load is around 7%, which corresponds to the reduction of bandwidth reserved with the mitigation, from 42.6 Mbps to 39.45 Mbps.

9.2.4 Computation time

The improvements of the TT metrics compared to the current method are obtained at the cost of an increase of the computation time (i.e. run time of the schedule generator), which is multiplied by up to 15 times, from about 3 s to 45 s when applied to all the flows. The computation times can be drastically improved by only applying the mitigation method to very low deadline constraints, jitter constraints or highly loaded ports. For instance, for UC_1 with a deadline of 96 μs for TT5, the average computation time is 3.25 s with the current method, 3.93 s with the mitigation method only applied to TT5, and 3.91 with the mitigation method only applied to output port $[SW2, ES12]$. With a deadline of 61 μs, average computation time is 4.76 s with the mitigation method only applied to TT5. We obtain the same results as Sections 9.2.1, 9.2.2 and 9.2.3 when applying the mitigation method only on a flow or port of interest. Hence, by selectively applying the mitigation method, we are able to achieve large improvements for a low computational cost.

To conclude, the AS6802 standard has currently a large impact on the TT traffic, with an increase of the end-to-end delays up to 74.5 %, an increase of the maximum jitter from 0 to 20 μs and an increase of the reserved bandwidth of 11%, in our case study.

The mitigation method has good results in decreasing these impacts. The impact on the end-to-end delays is reduced by up to 100 % (from 8.67% to 0%), the impact on the jitter is reduced up to 100% (from 20 μs to 0 μs), and the impact on the reserved bandwidth is divided by up to 6 (from 11% to 1.75%), in the most loaded port. By reducing the reserved bandwidth, we free bandwidth for additional flows. However, the actual increase of schedulable TT flows depends on the flows added to the network, which will be explored in future work.

We have shown that selecting appropriate IC and calculating the corresponding network precision ϵ is very important to obtain the best results possible with the mitigation method. Additional work is needed to explore the trade-off between the effects of increasing IC to reduce the number of PCFs, and the effect of the resulting decrease of the network precision.

9.3 Impact of PCFs on RC

We compare the worst-case delays computed with the method proposed in the paper, with the delays computed using the method described in [22], which does not consider PCFs. Then we discuss the effects of the PCF parameters, mitigation method and scheduler on RC.

9.3.1 Impact for a fixed schedule and fixed (IC, ϵ)

The TT integration policy guard band is set for all flows in all ports. The deltas between RC worst-case delays without (modeled from [22]) and with (see Th. 11) PCF traffic for UC_4 are detailed in Fig. 3. The RC flows are generally multicast and their worst-case delays are presented individually by destination: for an RC flow X, with a destination ESY, the flow is denoted $X.Y$. For instance, 2.7 refers to flow RC 2, from ES4 to ES7.

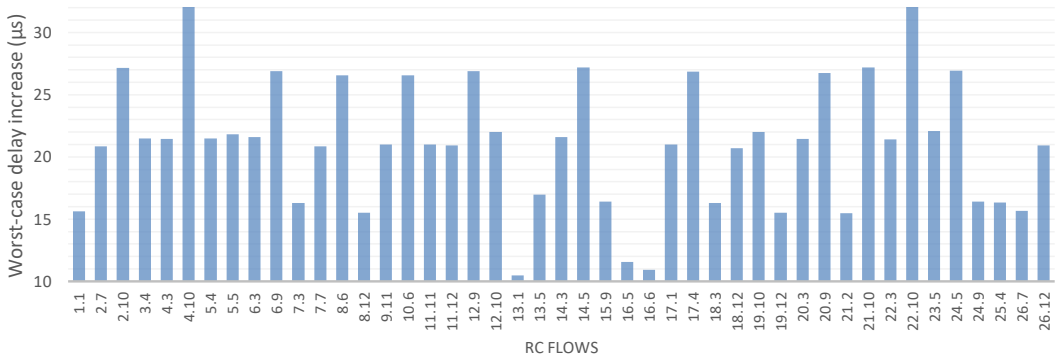


Figure 3 RC worst-case delay increase with PCF (Th. 11), compared to without PCF (cf. [22]) for UC_4.

Results show that for UC_4={IC=1 ms & ϵ=5 μs}, the RC delays are increased between 10.49 μs and 32.56 μs (or between 0.55 % and 1.99 %) by the added PCF traffic. This increase depends mainly on the number of PCFs in the RC flow path: e.g. an increase of about i) 11 μs is due to 3 PCFs, ii) 21 μs is due to 5 PCFs, iii) 32 μs is due to 7 PCFs. For instance, for RC flow 16.6 between ES4 and ES6, there are 3 PCFs along the path and the worst-case delay is increased by 10.91 μs. For RC flow 4.10 between ES9 and ES10, there are 7 interfering PCFs, and the worst-case delay is increased by 32.56 μs. The increase also depends on the IW placements with regard to the TT schedule. For example, both RC 13.5 and RC 1.1 encounter 4 PCFs, but the worst-case delay increases by 16.98 μs for RC 13.5, and by 15.61 μs for RC 1.1.

To conclude, in this experiment the PCFs increase the worst-case delay of the RC traffic (up to 32.56 μs or 1.99%), depending on the number of PCFs in the path and the interactions between IWs and TT offsets. Hence, not taking PCFs into account may lead to optimistic bounds and result in an unsafe system.

9.3.2 Discussion: impact of (IC, ϵ)

When IC increases, the number of PCFs in a hyperperiod decreases, which decreases the arrival curve of the $PCF \cup TT$ traffic (see Th. 11). So, the impact $PCF \cup TT$ traffic decreases and the RC delays are smaller.

As presented in Section 9.2.3, IW is larger when ϵ increases. Hence, in Lemma 10, the arrival window of a PCF is larger. Consequently, it is more likely for the minimum inter-arrival times to be equal to 0, which increases the arrival curve of the $PCF \cup TT$ traffic. So, the impact $PCF \cup TT$ traffic increases and the RC delays are larger.

Hence, when IC increases and ϵ is constant, the RC delays decrease. When IC is constant and ϵ is increases, the RC delays increase.

As increasing of IC also increases the corresponding minimum ϵ, selecting a correct couple (IC, ϵ) is to find a good balance between the two effects of IC and ϵ, similarly to the effects discussed in Section 9.2.1.

9.3.3 Discussion: impact of mitigation method and schedule generator

The mitigation method computes differently the TT sending windows. As such, the schedule selected by the chosen solver may be different with the mitigation method compared to without. However, the impact is only due to the different selected offsets. Hence, the selection of a different schedule generator could have a similar impact. In this evaluation, we have selected an SMT solver, but our mitigation method could be applied with other solutions. So comparing the solution with and without mitigation method is similar to comparing different schedules obtained by different schedule generators, which, while interesting, is not the objective of this paper.

10 Conclusion

In this paper, we have proposed a novel Network Calculus model of the Protocol Control Frame (PCF) defined in the AS6802 synchronization protocol. Using this model, we have computed the impact of the PCF traffic on the Time-Triggered (TT) traffic. In particular, we proposed a method to reduce the impact of PCFs, by improving the TT Sending Window computation. Moreover, we have used the PCF model to compute a novel Network Calculus model of both TT and PCF traffic to assess the impact on Rate-Constrained (RC) traffic.

Our test results show that with the current method, the impact of the PCFs on TT represents an increase of up to 74.5% of the end-to-end delays, up to 11% of the reserved bandwidth, and up to 20 μs of the maximum jitter. We have also shown that the PCF traffic increases the RC worst-case delays, up to 32.56 μs or 1.99% in our case study. Hence, PCFs must be taken into account to obtain correct worst-case delays for RC traffic and ensure a safe system.

Finally, in our case study, with our proposed mitigation method, i.e. improved TT Sending Windows method, we obtain a reduction of the impact of the PCFs on TT on the end-to-end delay (up to 100%), jitter (up to 100%) and reserved bandwidth (up to 6 times) compared to the current method. Throughout the evaluation, we have showed that finding the correct tuple (IC, ϵ) is of paramount importance to obtain good performances and reduce the impact of PCFs on TT and RC.

In future work, we plan to further study the impact of the integration cycle and network precision on both the mitigation method and on the impact of PCFs on RC and TT, and to validate the mitigation method on a larger industrial network.

References

1 Airlines Electronic Engineering Committee. Aircraft Data Network Part 7, Avionics Full Duplex Switched Ethernet (AFDX) Network, ARINC Specification 664. In *Aeronautical Radio*, 2002.

2 Anne Bouillard, Laurent Jouhet, and Eric Thierry. Service curves in Network Calculus: dos and don'ts. Research report, INRIA, 2009.

3 M. Boyer, H. Daigmorte, N. Navet, and J. Migge. Performance impact of the interactions between time-triggered and rate-constrained transmissions in TTEthernet. In *European Congress on Embedded Real Time Software and Systems*, 2016.

4 Marc Boyer, Jörn Migge, and Nicolas Navet. An efficient and simple class of functions to model arrival curve of packetised flows. In *Proc. of the 1st Int. Workshop on Worst-Case Traversal Time (WCTT)*, 2011.

5 Silviu S. Craciunas and Ramon Serna Oliver. Combined task- and network-level scheduling for distributed time-triggered systems. *Real-Time Systems*, 52(2):161–200, 2016.

6 J. Diemer, D. Thiele, and R. Ernst. Formal worst-case timing analysis of Ethernet topologies with strict-priority and AVB switching. In *Proc. International Symposium on Industrial Embedded Systems (SIES)*. IEEE Computer Society, 2012.

7 Fabrice Frances, Christian Fraboul, and Jérôme Grieu. Using network calculus to optimize the AFDX network. In *Embeeded Real Time Software and Systems (ERTS)*, 2006.

8 Jérôme Grieu. *Analyse et évaluation de techniques de commutation Ethernet pour l'interconnexion des systèmes avioniques*. PhD thesis, INPT, 2004.

9 Hermann Kopetz, Astrit Ademaj, Petr Grillinger, and Klaus Steinhammer. The Time-Triggered Ethernet (TTE) Design. *8th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC), Seattle, Washington*, 2005.

10 J.Y. Le Boudec and P. Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer-Verlag, 2001.

11 SAE International. SAE AS6802 Time-Triggered Ethernet. `http://standards.sae.org/as6802/`, 2011.

12 Miladin Sandić, Ivan Velikić, and Aleksandar Jakovljević. Calculation of number of integration cycles for systems synchronized using the AS6802 standard. In *2017 Zooming Innovation in Consumer Electronics International Conference (ZINC)*, pages 54–55. IEEE, 2017.

13 Eike Schweissguth, Peter Danielis, Dirk Timmermann, Helge Parzyjegla, and Gero Mühl. ILP-based joint routing and scheduling for time-triggered networks. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 8–17. ACM, 2017.

14 Wilfried Steiner. An Evaluation of SMT-based Schedule Synthesis For Time-Triggered Multi-Hop Networks. In *RTSS*. IEEE, 2010.

15 Wilfried Steiner. Synthesis of static communication schedules for mixed-criticality systems. In *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, pages 11–18. IEEE, 2011.

16 Wilfried Steiner, Günther Bauer, Brendan Hall, and Michael Paulitsch. TTEthernet: Time-Triggered Ethernet. In Roman Obermaisser, editor, *Time-Triggered Communication*. CRC Press, August 2011.

17 Wilfried Steiner and Bruno Dutertre. Automated formal verification of the TTEthernet synchronization quality. In *NASA Formal Methods Symposium*, pages 375–390. Springer, 2011.

18 Wilfried Steiner and Bruno Dutertre. The TTEthernet synchronisation protocols and their formal verification. *International Journal of Critical Computer-Based Systems 17*, 4(3):280–300, 2013.

19 Domitian TamasSelicean, Paul Pop, and Wilfried Steiner. Timing analysis of rate constrained traffic for the TTEthernet communication protocol. In *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*, pages 119–126. IEEE, 2015.

20 Daniel Thiele, Philip Axer, and Rolf Ernst. Improving formal timing analysis of switched ethernet by exploiting FIFO scheduling. In *Proceedings of the 52nd Annual Design Automation Conference*, page 41. ACM, 2015.

21 L. X. Zhao, H. G. Xiong, Z. Zheng, and Q. Li. Improving worst-case latency analysis for rate-constrained traffic in the time-triggered ethernet network. *IEEE Communications Letters*, 18(11):1927–1930, 2014.

22 Luxi Zhao, Paul Pop, Qiao Li, Junyan Chen, and Huagang Xiong. Timing analysis of rate-constrained traffic in TTEthernet using network calculus. *Real-Time Systems*, 53(2):254–287, 2017.

A Appendix

A.1 Proof of Theorem 5: PCF Integration Window

Proof. To compute a IW, i.e. $EA^p_{i \in PCF}$ and $LF^p_{i \in PCF}$, we need to consider when a frame can arrive in the output port *psrc* of a source node, i.e. between the earliest arrival time

($EA^{psrc}_{i\in PCF}$) and the latest arrival time ($LA^{psrc}_{i\in PCF}$). We must also consider the delays $\delta^{[src,n]}_{i\in PCF}$ and $\Delta^{[src,n]}_{i\in PCF}$ defined in Section 5.3, and the precision of the network synchronization ϵ:

$EA^{p}_{i\in PCF} = EA^{psrc}_{i} + \delta^{[src,n]}_{i} - \epsilon$ and $LF^{p}_{i\in PCF} = LA^{psrc}_{i} + \Delta^{[src,n]}_{i} + \Delta^{p}_{i} + \epsilon$.

It is necessary to consider ϵ since EA^{psrc}_{i} and LA^{psrc}_{i} are from the point of view of the local time of the source node, while EA^{p}_{i} and LF^{p}_{i} are from the point of view of the local time of p. Next, we compute EA^{psrc}_{i} and LA^{psrc}_{i} for the different PCF origins.

PCFs from SM to CM: the PCFs of flow i sent by the SMs are released for transmission selection at the beginning of each IC. Each IC starts at local time 0, thus:

$EA^{psrc}_{i\in PCF} = LA^{psrc}_{i\in PCF} = 0$.

PCFs from CM to SM and SC: to compensate for the asymmetric reception of PCFs (due to different path sizes for example), AS6802 implements the so-called *permanence function* inside the CMs. This service function applies a delay to the PCF reception times equal to the statically computed worst-case travel time of any PCF in the network (called the *maximum transparent clock* MTC), ensuring that the CMs process the PCFs according to the PCFs release times, rather than reception time. For this, each PCF accumulates its travel time in its *transparent clock* TC field, which is increased at each intermediate hop it traverses. Upon arrival of a frame at time t_A in the CM input port, the $permanence function$ computes the delta between the maximum transparent clock MTC and the transparent clock TC, denoted δ_{TC}. Then, it delays the delivery of the PCF to the CM host until the expiration of this delta. When the frame is delivered, it is said to become *permanent*. This so-called permanent time also depends on δ_{clock}, the clock drift between the values of the clock of the source and the clock of the destination, as the sending time is from the point of view of the source and the permanent time is from the point of view of the destination.

Hence, the permanent time is: $t_A + \delta_{TC} = TC + \delta_{clock} + MTC - TC = \delta_{clock} + MTC$.

The maximum (resp. minimum) value of the clock drift δ_{clock} (and consequently the value of the clock compensation delay) is equal to $+\epsilon$ (resp. $-\epsilon$).

Next, the CM compute the clock compensation delay, during a known Compression Master delay ΔCM. Then, after taking into account the clock compensation, it sends the PCFs to the SMs and CMs.

So, we obtain: $EA^{psrc}_{i\in PCF} = -\epsilon + MTC + \Delta CM$ and $LA^{psrc}_{i\in PCF} = +\epsilon + MTC + \Delta CM$ ◀

A.2 Proof of Theorem 9: Policed Time-Triggered Frames

Proof. With shuffling, the TT frame is delayed until the lower priority frame (RC or BE) finishes its transmission. Hence, the earliest arrival time EA and the maximum frame size MFS of a PTT frame remain the same as EA and MFS of the corresponding TT frame.

With guard band (GB), a RC or BE frame is blocked from transmission if a TT frame is scheduled to be sent before the RC frame would complete its transmission. The length of the guard band before a TT frame can be upper bounded by either the maximal transmission time of a lower priority frame, i.e. RC or BE, competing on the output port p, or the idle time distance with the previous TT frame. Thus, the guard band duration of $f^{p,\tau}_{g}$ is:
$\Delta GB^{p}_{g\in TT} = \min(\frac{MFS^{p}_{RC\cup BE}}{C^{p}_{out}}, EA^{p,\tau}_{g\in TT} - EF^{p,\tau}_{g\ominus 1\in TT})$, where $g \ominus 1$ represents the frame preceding $f^{p,\tau}_{g}$. As the guard band only appears immediately before the TT frame, and RC∪BE frames are delayed by both guard bands and TT frames, it is assumed that "GB+TT" is taken as an entirety. So, we have a new MFS and a new earliest arrival time:
$MFS^{p,\tau}_{g\in PTT} = MFS_{g\in TT} + \Delta GB^{p}_{g\in TT} \cdot C^{p}_{out}$ and $EA^{p,\tau}_{g\in PTT} = EA^{p,\tau}_{g\in TT} - \Delta GB^{p}_{g\in TT}$, with $g \in PTT$ when considering the TT frame $f^{p,\tau}_{g}$ with the integration policy, $g \in TT$ without the integration policy.

With preemption (PR), even though the lower priority frame is preempted by a TT frame, the TT frame will suffer a delay of the fixed-sized preemption overhead PO. As the PO postpones and appears immediately before the TT frame, we assume an entirety "PR+TT" of PTT which impacts on RC traffic. Then, PTT ("PR+TT") obtains the same earliest arrival time as TT and $MFS^{p,\tau}_{g\in PTT} = MFS_{g\in TT} + PO$.

All three integration policies have a fixed impact on the arrival time of a PTT frame, and a TT frame is always released by the device to the output port p at $EA^{p,\tau}_{g\in TT}$. Hence, $LA^{p,\tau}_{g\in TT} = EA^{p,\tau}_{g\in TT}$ and the period is τ^p. ◀

Offloading Safety- and Mission-Critical Tasks via Unreliable Connections

Lea Schönberger
Design Automation for Embedded Systems Group, Faculty of Computer Science, TU Dortmund University, Germany
lea.schoenberger@tu-dortmund.de

Georg von der Brüggen
Design Automation for Embedded Systems Group, Faculty of Computer Science, TU Dortmund University, Germany
georg.von-der-brueggen@tu-dortmund.de

Kuan-Hsun Chen
Design Automation for Embedded Systems Group, Faculty of Computer Science, TU Dortmund University, Germany
kuan-hsun.chen@tu-dortmund.de

Benjamin Sliwa
Communication Networks Institute, Faculty of Electrical Engineering, TU Dortmund University, Germany
benjamin.sliwa@tu-dortmund.de

Hazem Youssef
Chair of Material Handling and Warehousing, Faculty of Mechanical Engineering, TU Dortmund University, Germany
hazem.youssef@tu-dortmund.de

Aswin Karthik Ramachandran Venkatapathy
Chair of Material Handling and Warehousing, Faculty of Mechanical Engineering, TU Dortmund University, Germany
aswin.ramachandran@tu-dortmund.de

Christian Wietfeld
Communication Networks Institute, Faculty of Electrical Engineering, TU Dortmund University, Germany
christian.wietfeld@tu-dortmund.de

Michael ten Hompel
Chair of Material Handling and Warehousing, Faculty of Mechanical Engineering, TU Dortmund University, Germany
michael.tenHompel@tu-dortmund.de

Jian-Jia Chen
Design Automation for Embedded Systems Group, Faculty of Computer Science, TU Dortmund University, Germany
jian-jia.chen@tu-dortmund.de

Abstract

For many cyber-physical systems, e.g., IoT systems and autonomous vehicles, offloading workload to auxiliary processing units has become crucial. However, since this approach highly depends on network connectivity and responsiveness, typically only non-critical tasks are offloaded, which have less strict timing requirements than critical tasks. In this work, we provide two protocols allowing to offload critical and non-critical tasks likewise, while providing different service levels for non-critical tasks in the event of an unsuccessful offloading operation, depending on the respective system requirements. We analyze the worst-case timing behavior of the local cyber-physical system and, based on these analyses, we provide a sufficient schedulability test for each of the proposed protocols. In the course of comprehensive experiments, we show that our protocols have reasonable acceptance ratios under the provided schedulability tests. Moreover, we demonstrate that the system behavior under our proposed protocols is strongly dependent on probability of unsuccessful offloading operations, the percentage of critical tasks in the system, and the amount of offloaded workload.

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

Keywords and phrases internet of things, cyber-physical systems, real-time, mixed-criticality, self-suspension, computation offloading, scheduling, communication

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.18

Funding This work is supported by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876, projects A1, A3, A4, and B4.

Acknowledgements The authors thank Jui-Lin Liang for his support and Niklas Ueter for his valuable feedback.

1 Introduction

Processing large amounts of sensor data within short, pre-defined intervals of time is crucial for many cyber-physical systems as, e.g., IoT systems, robots, drones, and autonomous vehicles, in order to accomplish their mission or even to maintain their operability. However, it may be the case that a system does not have sufficient resources at its disposal to always perform the necessary operations fast enough and to deliver the required results in time if these computations are performed merely locally. Since the system hardware can only be enhanced up to a certain level due to constraints in terms of cost, energy efficiency, size, and other noteworthy factors, such insufficiencies can be overcome by offloading a share of the system workload via, e.g., 4G/5G or IEEE802.11p-based [18] wireless connections, while providing a local fallback mechanism. Nevertheless, wireless connections exhibit a certain level of unreliability, which must be factored in when deciding which tasks to offload. As an example, the resulting end-to-end performance within cellular vehicular communication systems is severely impacted by highly dynamic channel conditions related to shadowing effects, multipath fading, handover situations, and even technology switches [24]. In addition, as the available cell resources are shared by the different network participants, the achievable network performance significantly depends on the traffic patterns of the other active cell users and the implemented resource scheduling policy of the cell. Different network quality indicators can be utilized to estimate the end-to-end behavior of data transmissions in terms of delay and data rate. However, as pointed out in recent analyses [25], the interdependencies of these factors can significantly differ among multiple mobile network operators due to varying strategies for network configuration and infrastructure deployment.

Against this background, we aim to allow resource-constrained systems to offload computation shares not only of non-criticial, but also of safety- and mission-critical tasks, while ensuring that the timing requirements of safety- and mission-critical tasks are not violated even in the case of connectivity issues and providing as much service for non-critical tasks as possible. Accordingly, we strive to specify the system's offloading behavior in such a way that it can be verified at design time for all potential scenarios.

Self-Suspension. From a modeling perspective, the considered cyber-physical system can be reduced to the local system, on which the performed offloading operations are perceived as tasks being executed, paused, and (in the successful case) resumed after an upper-bounded interval of time. One concept allowing to model this particular local system view is *self-suspension* [7], which characterizes tasks temporarily interrupting their execution and proceeding as soon as a certain operation is finished, i.e., as soon as a response from the auxiliary processing unit is received. In fact, the actual time elapsing between the moment a message is sent to an auxiliary processing unit and the latest safe moment in which a response may be received could be simply modeled as additional computation time rather than as so-called *suspension time*, but this would lead to a pessimistic under-utilization of computation resources [23, 28]. Instead, for the sake of accuracy, one of the state-of-the-art models can be applied such as the dynamic self-suspension model (cf. e.g. [12], [15]), the segmented self-suspension model (cf. e.g. [22]), or a hybrid model, e.g. [27], (for a detailed overview refer to [6, 7]). In this work, we make use of the *segmented self-suspension model*, which allows to precisely depict a specific suspension pattern, i.e., to specify the exact point in time in which an offloading operation starts as well as a legal upper bound on its duration (formal descriptions will be given in Sec. 2).

Mixed-Criticality. Modeling a local system as well as successful offloading operations, however, does not suffice as a basis for verifying the behavior of a cyber-physical system as considered in this work. De facto, the question remains, how to model and how to handle unsuccessful offloading operations. To this effect, we benefit from the notion of so-called *mixed-criticality systems*, which were formally introduced for the first time by Vestal [26] and received much attention thenceforward (a comprehensive survey can be found in [5]). This concept describes systems integrating tasks with different *criticality levels* on the same platform and providing distinct system modes, usually one per criticality level. In case of special events such as fault-occurrence, mixed-criticality systems perform a *mode change*, i.e., switch to another system mode, which permits to maintain the system safety by ensuring that the timing requirements of all tasks corresponding to the current or a higher system mode can still be met. For all lower-criticality tasks, however, no more timing guarantees are provided. Analogously to mode changes in mixed-criticality systems, it is necessary to carefully anticipate all events that may occur during an offloading operation, e.g., a missing response due to an unreliable wireless connection, and to specify mechanisms to handle these deterministically. However, we do *not* model the contemplated type of cyber-physical systems as a mixed-criticality system, but rather exploit the characteristics exhibited by the latter. Namely, we classify the overall set of tasks in the system into a set of *critical tasks* (comprising safety- as well as mission-critical tasks) and a set of *non-critical tasks*, but we do not specify explicit system modes. Instead, we consider the system to exhibit different *execution behaviors* under different circumstances: either a *normal* execution behavior, under which timing constraints are satisfied for all tasks, or a *local* one, under which timeliness is only guaranteed for critical tasks. The actual system behavior in each possible scenario, however, needs to be clearly defined and to follow a pre-specified, analyzable, and verifiable protocol, which will be developed hereinafter.

Related Work. The idea of cloud-based control for automotive systems is not new, but has already been addressed in 2012 by Kumar et al. [14], who proposed a cloud-assisted system for autonomous driving, which, however, is not used to offload control applications, but rather to provide additional information to the vehicle. In 2015, Esen et al. [9] presented a software architecture named *Control as a Service* according to which all control functions are completely moved to the cloud, while only sensors, actuators, and communication infrastructure remain in the vehicle. Network latencies have been pointed as a challenge, but no concrete solution or mechanism to handle suchlike has been proposed, and, moreover, connection losses have not been addressed at all. In a proof of concept, the authors have modeled network latencies using discrete stochastic models. In 2018, Adiththan et al. [1] proposed an adaptive offloading technique for control applications that makes all offloading decisions online based on a network performance monitor. However, due to the heuristic nature of the approach, the timing behavior of the system cannot be verified. Beyond that, the authors mentioned the necessity to handle connectivity losses and large communication delays and stated that in such cases task executions must be redirected to the local system. Nevertheless, no concrete mechanism or protocol has been suggested for this purpose. Moreover, the potential consequences have not been further discussed. Recently, Al Maruf and Azim [17] proposed a strategy for task offloading in multiprocessor mixed-criticality systems with dynamic scheduling policies under overload conditions. More precisely, they suggested to select low-criticality tasks based on a machine learning approach that are offloaded to the cloud and executed in parallel aiming to reduce the number of deadline misses. Potential connectivity issues have not been taken into consideration.

Contributions. In a nutshell, we contribute the following:

- We provide two protocols for cyber-physical systems allowing to safely offload critical and non-critical tasks, which address different system requirements: i) the *service protocol* provides as much service for non-critical tasks as possible in any point in time, and ii) the *return protocol* allows a fast return to the normal system behavior in the case of an unsuccessful offloading operation.
- In Sec. 4 and Sec. 5, we analyze the worst-case timing behavior of the local system and examine our proposed protocols by considering all possible transient and ready states regarding the normal and the local execution behavior. Based on these analyses, we provide a sufficient schedulability test for each of the proposed protocols.
- By means of comprehensive simulations and a case study, we i) show that the acceptance ratios of our proposed protocols under the provided schedulability tests are reasonable, and ii) demonstrate that the system behavior under our proposed protocols is strongly dependent on probability of unsuccessful offloading operations, the percentage of critical tasks in the system, and the amount of offloaded workload.

2 System Model

We consider a cyber-physical system comprising a set of tasks $\mathcal{T}$ that can be divided into two subsets with different requirements, namely, the set of *critical* tasks $\mathcal{T}_{crit}$, and the set of *non-critical* tasks $\mathcal{T}_{non}$, such that $\mathcal{T} = \mathcal{T}_{crit} \cup \mathcal{T}_{non}$ and $\mathcal{T}_{crit} \cap \mathcal{T}_{non} = \emptyset$. While for each $\tau_k \in \mathcal{T}_{crit}$ timing constraints must be satisfied at any point in time, for each $\tau_k \in \mathcal{T}_{non}$ timing violations may be unpleasant but not hazardous. According to the classification of tasks into two subsets, we specify two different system execution behaviors, i.e., *normal* and *local* execution behavior. When the system exhibits normal execution behavior, all timing requirements of all tasks are satisfied at any point in time, whereas, if the system exhibits local execution behavior, timing guarantees can only be given for all critical tasks $\tau_k \in \mathcal{T}_{crit}$. The degree of service provided with respect to the non-critical tasks $\tau_k \in \mathcal{T}_{non}$ depends on the particular recovery protocol implemented in the system (cf. Sec. 3).

Each recurrent real-time task $\tau_k \in \mathcal{T}$ in the considered cyber-physical system is assumed to have a sporadic arrival pattern and is characterized[1] by a tuple $(C_{k,1}, C_{k,s}, C_{k,2}, S_k, p_k, q_k, D_k, T_k)$:

- Each τ_k releases an infinite number of task instances denoted as *jobs*. T_k indicates the minimum inter-arrival time of τ_k, i.e., the arrival times of any two consecutive jobs of τ_k must be separated by at least T_k.
- D_k describes the relative deadline of τ_k. The absolute deadline of a job of task τ_k arriving at time r_k is given by $r_k + D_k$ if *it must be guaranteed that the job meets its deadline.* Throughout this paper, we assume a constrained-deadline task system, in which $D_k \leq T_k$ for each task τ_k.
- $C_{k,1}$ and $C_{k,2}$ denote the worst-case execution times of the first and second *computation segments*, respectively.
- $C_{k,s}$ is the worst-case execution time of the typically offloaded task share if executed on the *local* system.
- p_k and q_k are the worst-case execution times of the pre- and post-processing routines, which are executed before and after the offloading operation of a job of task τ_k, respectively.
- S_k is the offloading or *suspension* time of τ_k.

[1] To provide a better overview, the notation is additionally summarized in Table 1.

Table 1 An overview about the notation.

Notation	Meaning
$C_{k,1}$	worst-case execution time (WCET) of the first computation segment of τ_k
$C_{k,2}$	WCET of the second computation segment of τ_k
$C_{k,s}$	WCET of the computation segment of τ_k that is typically offloaded if executed locally
$C_k^\flat$	$C_{k,1} + p_k + q_k + C_{k,2}$
$C_k^\sharp$	$C_{k,1} + S_k + C_{k,2}$
D_k	relative deadline of τ_k
$hp(\tau_k)$	set of tasks with higher priority than τ_k
p_k	WCET of the offloading pre-processing routine of τ_k
q_k	WCET of the offloading post-processing routine of τ_k
S_k	offloading/suspension time of τ_k
T_k	minimum inter-arrival time of τ_k
$\mathcal{T}$	complete task set
$\mathcal{T}_{crit}$	set of critical tasks
$\mathcal{T}_{non}$	set of non-critical tasks

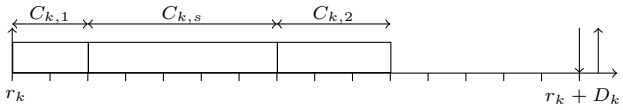


Figure 1 A job of task τ_k is executed locally (local execution behavior).

We assume that $T_k \geq D_k > 0$ and $C_{k,1}, C_{k,s}, C_{k,2}, S_k, p_k, q_k \geq 0$. Moreover, we make the natural assumption that $p_k + q_k \leq C_{k,s}$, since offloading is not meaningful otherwise. Furthermore, the worst-case execution time of a job of task τ_k under any possible execution scenario is greater than 0, i.e., $C_{k,1} + C_{k,s} + C_{k,2} > 0$ and $C_{k,1} + p_k + q_k + C_{k,2} > 0$. For notational brevity, we denote $C_k^\sharp = C_{k,1} + C_{k,s} + C_{k,2}$ and $C_k^\flat = C_{k,1} + p_k + q_k + C_{k,2}$.

Throughout this paper, we assume that the local cyber-physical real-time system, termed *local system*, is a uniprocessor system, in which tasks are scheduled according to a preemptive fixed-priority policy. More precisely, each task is assigned a unique priority, i.e., all jobs of task τ_k have the same priority. If at any point in time multiple jobs are ready, i.e., eligible for being executed on the local system, the job having the highest priority is executed. For each task τ_k, the unique set of the higher-priority tasks is denoted as $hp(\tau_k)$.

For a job of task τ_k arriving at time r_k the following execution scenarios are possible:

- The job is *executed locally* (cf. Fig. 1). In this case, the worst-case execution time of the job released at time r_k is $C_{k,1} + C_{k,s} + C_{k,2}$, i.e., $C_k^\sharp$.
- The job is *offloaded*. In this case, the job is first executed locally for up to $C_{k,1}$ execution time units and thereon enters the pre-processing routine for offloading for up to p_k execution time units. Suppose that the first computation segment as well as the pre-processing routine are finished at time ρ. Then, the considered job is offloaded to the remote system at time ρ. The actual offloading operation can be either *successful* or *unsuccessful*:
  - *Offloading is successful* if the computation result or *offloading response* is returned to the local system until time $\rho + S_k$. In this case, the offloading response is post-processed for up to q_k time units and the second computation segment is executed for up to $C_{k,2}$

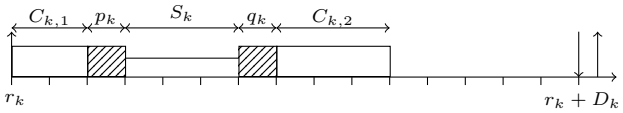


Figure 2 An offloading operation of a job of task τ_k is performed successfully (normal execution behavior).

time units (cf. Fig. 2). Accordingly, the execution time of the job of τ_k on the local system is at most $C_k^{\flat}$.

- *Offloading is unsuccessful* otherwise. In this case, at time $\rho + S_k$, a local re-execution of the offloaded task share is performed for up to $C_{k,s}$ time units followed by the execution of the second computation segment for up to $C_{k,2}$ time units. In this case, the execution time of the job of τ_k on the local system is at most $C_k^{\sharp} + p_k$. This scenario will be discussed more in detail hereinafter.

3 Recovery Protocols

Cyber-physical systems can be encountered throughout a broad range of application areas, each exhibiting individual requirements and thus a need for situationally appropriate system behavior. For safety-critical cyber-physical systems, the timeliness of critical tasks must be guaranteed under any circumstances - even in the event of an unsuccessful offloading operation. Since in this case a larger amount of local resources is required, as explained with respect to the possible execution scenarios in Sec. 2, less resources remain to serve the non-critical tasks. However, depending on the actual system characteristics, timing constraints for non-critical tasks tend to be less strict. It is, for instance, possible that a non-critical task misses its deadline, but the results are still useful up to a certain degree [4, 3]. Nevertheless, it may be desirable to return to the normal execution behavior and to re-establish timing guarantees for both critical and non-critical tasks as soon as possible, especially since a non-critical task is not necessarily unimportant and thus should provide functionally and temporally correct results most of the time (further discussion on the relation between criticality and importance can be found in [10]).

Against this backdrop, we propose two recovery protocols allowing the system to satisfy its requirements under local execution behavior and to return to normal execution behavior:

- The *service protocol* aims to provide as much service as possible for non-critical tasks, even under local execution behavior.
- The *return protocol* aims to minimize the amount of time, in which the system exhibits local execution behavior after an unsuccessful offloading operation.

Independent of the actual protocol, we assume that the local system exhibits normal execution behavior at time 0, such that offloading is enabled for all tasks in $\mathcal{T}$. The schedule considers the execution of all tasks until the first moment $\gamma_{1,\searrow}$, in which the offloading operation of a certain task τ_k is unsuccessful, i.e., a job of task τ_k, which has offloaded its computation at time $\gamma_{1,\searrow} - S_k$, does not receive the offloading response until time $\gamma_{1,\searrow}$ (cf. Fig. 3). Immediately after $\gamma_{1,\searrow}$, the local system exhibits local execution behavior. Until time $\gamma_{1,\searrow}$, three scenarios are possible for each incomplete job of all critical tasks τ_i in $\mathcal{T}_{crit}$:

- *The job of τ_i has not been offloaded*: In this case, no offloading operation will be performed for this job, but it is executed locally instead. Since it is possible that the pre-processing routine for offloading is already active at time $\gamma_{1,\searrow}$, the worst-case execution time of this job is upper-bounded by $C_{i,1} + p_i + C_{i,s} + C_{i,2}$, i.e., $C_i^{\sharp} + p_i$.

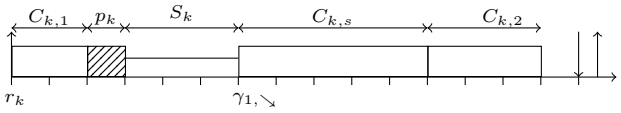


Figure 3 An unsuccessful offloading operation of τ_k resulting in the transition to the local system behavior at time $\gamma_{1,\searrow}$.

- *The job of τ_i is already offloaded, but no offloading response was received until time $\gamma_{1,\searrow}$*: In this case, the offloading process is aborted and the job is executed locally as of time $\gamma_{1,\searrow}$. Therefore, the worst-case execution time of this job is upper-bounded by $C_{i,1} + p_i + C_{i,s} + C_{i,2}$, i.e., $C_i^\sharp + p_i$.
- *The job of τ_i is already offloaded and the offloading response has been received prior to time $\gamma_{1,\searrow}$*: In this case, the job continues its final processing. Therefore, the worst-case execution time of this job is upper-bounded by $C_{i,1} + p_i + q_i + C_{i,2}$, i.e., $C_i^\flat$.

After $\gamma_{1,\searrow}$, timing guarantees are only provided for $\mathcal{T}_{crit}$. Moreover, offloading is inhibited for all critical tasks in the near future of $\gamma_{1,\searrow}$, due to the currently unreliable connection leading to the missing offloading response. The offloading decision for non-critical tasks, however, depends on the applied recovery protocol:

- **Service Protocol**: Under the *service protocol*, offloading is inhibited for all instances of all tasks that are active as long as the system exhibits local execution behavior. The task share of each $\tau_i \in \mathcal{T}$ that is offloaded under normal execution behavior is executed locally within $C_{i,s}$ units of execution time. Since this leads to a higher workload on the local system, timeliness cannot be guaranteed for any non-critical task. Nevertheless, no non-critical task is aborted.
- **Return Protocol**: The *return protocol* does not inhibit offloading for *all* tasks, but only for critical ones under local execution behavior. Non-critical tasks, in contrast, are offloaded regardless, but neither a re-execution nor a re-transmission is performed if an offloading response is not received in time. More precisely, the second subtask of τ_i is only executed if an offloading response is received, and aborted otherwise. Moreover, a job of τ_i in $\mathcal{T}_{non}$ is aborted whenever it misses its deadline.

As of time $\gamma_{1,\searrow}$, the local system exhibits local execution behavior until the point in time $\gamma_{1,\nearrow}$, in which timing guarantees can be given again for all tasks in $\mathcal{T}$. In the proposed protocols, two options are considered for the transit from local to normal execution behavior, which should be chosen depending on the actual system requirements:

- **Abort-Transit**: This option aims to re-establish the normal system execution behavior as quickly as possible. Suppose that $\gamma_{1,\nearrow}$ is the earliest moment (after $\gamma_{1,\searrow}$) in which there is no incomplete job from $\mathcal{T}_{crit}$ at $\gamma_{1,\nearrow}$. All released but not yet finished instances of non-critical tasks are discarded.
- **Idle-Transit**: This option re-establishes the normal system execution behavior at the earliest moment $\gamma_{1,\nearrow}$ (after $\gamma_{1,\searrow}$) in which there is no incomplete job from $\mathcal{T}$ at $\gamma_{1,\nearrow}$.

We note that the above transitions are well-defined and the local system exhibits normal and local execution behavior in an interleaving manner.

4 Existing Analysis and Workload Characteristics

When the system exhibits normal execution behavior, the same task execution patterns are identifiable under both proposed protocols, i.e., an offloading operation is performed for each

task. Hence, each $\tau_k \in \mathcal{T}$ is a (segmented) self-suspending task consisting of two computation segments as well as of one *suspension interval* of length S_k, and can therefore be analyzed applying any suitable technique.

▶ **Definition 1.** *Suppose that the system always exhibits normal execution behavior. Then, for each task $\tau_k \in \mathcal{T}$, the worst-case response time R_k^{normal} is the worst-case response time of task τ_k and R_k^1 is the worst-case response time of the first computation segment of task τ_k. By definition, $R_k^1 \leq R_k^{normal}$. This paper assumes that $R_k^{normal} \leq D_k \leq T_k$, $\forall \tau_k \in \mathcal{T}$.*

▶ **Lemma 2.** *If τ_k is in $\mathcal{T}_{non}$, the worst-case response time of τ_k under normal execution behavior is upper-bounded by R_k^{normal} regardless of the adopted protocol.*

Proof. This is based on the definition. ◀

Regarding the analysis of self-suspending tasks, several misconceptions exist in the literature. Detailed and correct treatments can be found in a recent survey paper by Chen et al. [6]. The latest result was developed by Schönberger et al. in [22]. However, instead of going into detail regarding the analysis of uniprocessor segmented self-suspending task systems, we assume that one of the existing analyses has been used and each task in $\mathcal{T}$ has been validated to meet its deadline if the system *always exhibits normal execution behavior* by applying the analysis given in [22].

The following lemma characterizes the maximum workload of a task τ_i that can be executed in a time interval $[t, t+\Delta)$ under the assumption that the local system resumes from idling at time t, i.e., it idles at $t-\varepsilon$ for an infinitesimal ε, under normal execution behavior and it does not switch from the local execution behavior to the normal execution behavior before $t+\Delta$ for $\Delta > 0$.

▶ **Lemma 3.** *Suppose that the local system resumes from idling at time t, i.e., it idles at $t-\varepsilon$ for an infinitesimal ε and executes a certain job at time t, under normal execution behavior and it does not switch from the local execution behavior to the normal execution behavior before $t+\Delta$ for $\Delta > 0$. For a task τ_i, in which*

- *τ_i is in $\mathcal{T}$ under the service protocol or*
- *τ_i is in $\mathcal{T}_{crit}$ under the return protocol,*

the amount of execution time for which task τ_i is executed in time interval $[t, t+\Delta)$ on the local system is upper-bounded by $\max\{f_1(\tau_i, \Delta), f_2(\tau_i, \Delta)\}$, where

$$f_1(\tau_i, \Delta) = p_i + \left\lceil \frac{\Delta}{T_i} \right\rceil C_i^\sharp \tag{1}$$

and

$$f_2(\tau_i, \Delta) = C_{i,s} + C_{i,2} + \left\lceil \frac{\Delta - (T_i - (R_i^1 + S_i))}{T_i} \right\rceil C_i^\sharp \tag{2}$$

Recall that $C_i^\sharp$ is defined as $C_{i,1} + C_{i,s} + C_{i,2}$.

Proof. We first consider the simpler case, in which the local system stays in the normal execution behavior from t to $t+\Delta$. In this case, there are two scenarios:

- Case 1a: if τ_i does not have any unfinished job before t (cf. Fig. 4), then, the workload of task τ_i executed in time interval $[t, t+\Delta)$ is at most $\left\lceil \frac{\Delta}{T_i} \right\rceil C_i^\flat \leq f_1(\tau_i, \Delta)$.

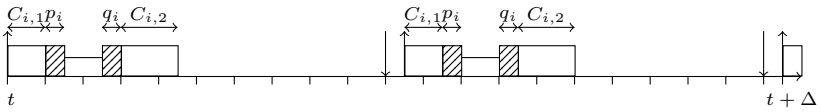


Figure 4 Execution of a task τ_i under analysis in $[t, t + \Delta)$ in case 1a of Lemma 3.

- Case 1b: If τ_i does have an unfinished job before t, then, by the definition that the local system returns from idling at time t, task τ_i has been suspended (cf. Fig. 5). Since the system still exhibits normal execution behavior prior to time t, we know that there is at most one such suspended job of τ_i and its arrival time r_i cannot be earlier than $t - (R_i^1 + S_i)$. Therefore, the first job of task τ_i released after t is released no earlier than $t - (R_i^1 + S_i) + T_i$. Since the system exhibits normal execution behavior from t to $t + \Delta$, the workload of task τ_i executed in time interval $[t, t + \Delta)$ is at most $q_i + C_{i,2} + \left\lceil \frac{\Delta - (T_i - (R_i^1 + S_i))}{T_i} \right\rceil C_i^\flat \leq f_2(\tau_i, \Delta)$.

We then consider another case, in which the local system switches to local execution behavior at time γ, where $t \leq \gamma < t + \Delta$. There are also two scenarios:

- Case 2a: There is no job of τ_i arriving before t that has not been finished yet by time t (cf. Fig. 6). From t to γ, at most $\left\lceil \frac{\gamma - t}{T_i} \right\rceil$ jobs of task τ_i are released. Specifically, among them, the last job of τ_i offloaded prior to γ may be offloaded unsuccessfully. For this particular job, its worst-case execution time is $C_{i,1} + p_i + C_{i,s} + C_{i,2} = C_i^\sharp + p_i$, whilst the worst-case execution time of the other $\left\lceil \frac{\gamma - t}{T_i} \right\rceil - 1$ jobs is at most $C_i^\flat \leq C_i^\sharp$. Moreover, for any job of τ_i released after γ, under the service protocol or the return protocol when $\tau_i \in \mathcal{T}_{crit}$, its worst-case execution time is at most $C_i^\sharp$. Therefore, the workload of τ_i from t to $t + \Delta$ is

$$\left(\left(\left\lceil \frac{\Delta}{T_i} \right\rceil - 1 \right) C_i^\sharp \right) + (C_i^\sharp + p_i) \leq \; p_i + \left\lceil \frac{\Delta}{T_i} \right\rceil C_i^\sharp \quad \stackrel{\text{def}}{=} \quad f_1(\tau_i, \Delta)$$

- Case 2b: A job of τ_i suspended prior to t and its second computation segment is released at or after t (cf. Fig. 7). Identically to the discussion in Case 1b, the next job of task τ_i arrives no earlier than $t - (R_i^1 + S_i) + T_i$. If the job suspended prior to t is offloaded unsuccessfully, i.e., its execution time after t is at most $C_{k,s} + C_{k,2}$, the other subsequent jobs of τ_i will be executed only locally, until the moment in which the local system switches to the normal execution behavior again, under the service protocol or under the return protocol when τ_i is in $\mathcal{T}_{crit}$. Therefore, the workload of τ_i from t to $t + \Delta$ is as defined in $f_2(\tau_i, \Delta)$. If the job suspended prior to t is offloaded successfully, at most of one of the jobs released after $t - (R_i^1 + S_i) + T_i$ is offloaded unsuccessfully. In this case, the workload of τ_i from t to $t + \Delta$ is at most

$$q_i + C_{i,2} + p_i + \left\lceil \frac{\Delta - (T_i - (R_i^1 + S_i))}{T_i} \right\rceil C_i^\sharp \leq f_2(\tau_i, \Delta),$$

since $p_i + q_i \leq C_{i,s}$. ◀

So far, the maximum workload that can be contributed to a time interval $[t, t + \Delta)$ by a task $\tau_i \in \mathcal{T}$ under the service protocol and by a task $\tau_i \in \mathcal{T}_{crit}$ under the return protocol has been analyzed. For the missing case that τ_i is in $\mathcal{T}_{non}$ under the return protocol, the workload in the time interval $[t, t + \Delta)$ can be reduced based on the definition of the protocol (cf. Sec. 3), as given in the following lemma:

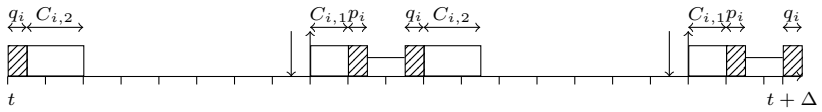


Figure 5 Execution of a task τ_i under analysis in $[t, t + \Delta)$ in case 1b of Lemma 3.

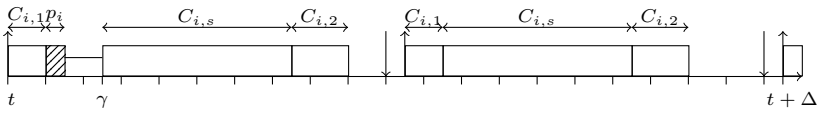


Figure 6 Execution of a task τ_i under analysis in $[t, t + \Delta)$ in case 2a of Lemma 3.

▶ **Lemma 4.** *For a task τ_i in $\mathcal{T}_{non}$ under the return protocol, under the same condition for t and $t + \Delta$ as specified in Lemma 3, the amount of execution time that task τ_i is executed in the time interval $[t, t + \Delta)$ is upper-bounded by $\left(\left\lceil \frac{\Delta}{T_i} \right\rceil + 1\right) C_i^\flat$.*

Proof. Under the return protocol, a job of task τ_i in $\mathcal{T}_{non}$ is aborted whenever it misses its deadline. Therefore, the number of jobs of τ_i, which have not yet missed their deadlines in a time interval $[t, t + \Delta)$, is at most $\left(\left\lceil \frac{\Delta}{T_i} \right\rceil + 1\right)$ since $D_i \leq T_i$. Under the return protocol, a task τ_i in $\mathcal{T}_{non}$ is always offloaded if the first computation segment is completed before the job's deadline. Any execution of such a job on the local system requires up to $C_i^\flat$ execution time units by definition. Therefore, $\left(\left\lceil \frac{\Delta}{T_i} \right\rceil + 1\right) C_i^\flat$ is the upper bound of the workload. ◀

5 Timing Analysis

To analyze the worst-case timing behavior of the local system, the timing behavior of each task τ_k must be analyzed beginning with the highest-priority task. In the course of this, two scenarios need to be analyzed:

- If τ_k is in $\mathcal{T}_{crit}$, the worst-case response time under local and normal execution behavior must be analyzed.
- If τ_k is in $\mathcal{T}_{non}$, only the worst-case response time under normal execution behavior must be analyzed.

We note that the worst-case response time of τ_k in $\mathcal{T}_{non}$ under local execution behavior is not of interest, since its jobs may be aborted (cf. Sec. 3).

In our analysis, we assume a concrete fixed-priority preemptive schedule σ for the task set $\mathcal{T}$ from time 0 onwards. For the concrete schedule σ, let $\gamma_{h,\searrow}$ be the h-th moment in which σ switches from normal to local execution behavior. Moreover, let $\gamma_{h,\nearrow}$ be the h-th moment in which σ switches from the local behavior to the normal behavior. By the definition of our protocols (cf. Sec. 3), $\gamma_{h,\searrow} < \gamma_{h,\nearrow} < \gamma_{h+1,\searrow}$.

We consider the j-th job of task τ_k, denoted as τ_k^j, in schedule σ and assume that at the arrival time of job τ_k^j there exists no incomplete job of task τ_k in the schedule σ. Now, we

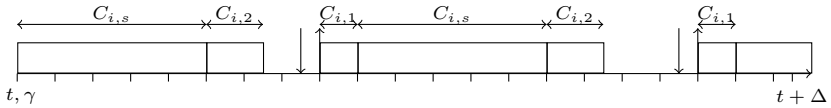


Figure 7 Execution of a task τ_i under analysis in $[t, t + \Delta)$ in case 2b of Lemma 3.

remove all lower-priority jobs from the schedule σ. Since σ is a fixed-priority preemptive schedule and all tasks in $\mathcal{T}$ are independent from each other, the removal of these jobs does not have any impact on the execution of any remaining job in the schedule σ. Thereon, we remove all jobs of task τ_k having arrived before the release of τ_k^j at time r_k^j from the schedule σ. Due to the assumption that the jobs of τ_k released before r_k^j have been finished before r_k^j, the removal of these jobs of τ_k does not have any impact on the execution of any remaining job in the schedule σ.

For simplicity of notation, we remove the index j for the rest of the proof, when the context is clear. Accordingly, r_k denotes the arrival time of the job of τ_k under analysis and f_k its finishing time. By definition, the next job of τ_k cannot arrive before $r_k + T_k$.

For the rest of this section, we will focus on the analysis of the case that τ_k is in $\mathcal{T}_{crit}$.

▶ **Lemma 5.** *Under both service and return protocol as well as both abort- and idle-transit, for any $\tau_k \in \mathcal{T}_{crit}$, in the interval $[r_k, f_k)$, the local system switches at most once from normal to local behavior. That is, at most one $\gamma_{h,\searrow}$ exists in $[r_k, f_k)$.*

Proof. This property results from the definition of the protocols and abort- and idle-transits. That is, the local system only switches from the local to normal execution behavior when there is no unfinished job of $\mathcal{T}_{crit}$. ◀

Based on Lemma 5, only four cases need to be considered:

- σ is executed under local execution behavior at time r_k and under normal execution behavior at time f_k, denoted as *L2N*.
- σ is executed under normal execution behavior at time r_k and under normal execution behavior at time f_k, denoted as *N2N*.
- σ is executed under normal execution behavior at time r_k and under local execution behavior at time f_k, denoted as *N2L*.
- σ is executed under local execution behavior at time r_k and under local execution behavior at time f_k, denoted as *L2L*.

In the following, we examine each of these cases individually, beginning with those that do not depend on the implemented recovery protocol, i.e., L2N and N2N. Thereon, the remaining cases are considered first under the service protocol in Sec. 5.1 and consecutively under the return protocol in Sec. 5.2.

▶ **Lemma 6.** *The case L2N is not possible under Abort-Transit and Idle-Transit.*

Proof. In both transitions from local to normal execution behavior, no incomplete job exists in the local system at time $\gamma_{h,\nearrow}$ for any $h \geq 0$. ◀

▶ **Lemma 7.** *The response time $f_k - r_k$ in the case N2N is at most R_k^{normal}, as defined in Definition 1.*

Proof. This is identical to self-suspension task systems. Suppose that $\gamma_{h,\nearrow} \leq r_k < \gamma_{h,\searrow}$. We can remove all the jobs in the schedule σ before $\gamma_{h,\nearrow}$ without changing any execution in σ after $\gamma_{h,\nearrow}$. Therefore, the jobs arriving in $[\gamma_{h,\nearrow}, f_k)$ are exactly the same as the task system analyzed in Definition 1. ◀

5.1 Analysis of the Service Protocol

Unlike the cases L2N and N2N, the cases N2L and L2L must be analyzed under each protocol separately, since the timing behavior of a task τ_k in $\mathcal{T}_{crit}$ under analysis differs depending on

how the actual system execution behavior is specified. For case N2L, the worst case response time of task τ_k can be obtained by means of the following lemma:

▶ **Lemma 8.** *Under the service protocol, the response time $f_k - r_k$ in the case N2L is upper-bounded by the minimum positive value of Δ, for which*

$$\Delta = p_k + C_k^\sharp + \sum_{\tau_i \in hp(\tau_k)} \max\{f_1(\tau_i, \Delta), f_2(\tau_i, \Delta)\} \tag{3}$$

if $\Delta \leq T_k$.

Proof. By definition of case N2L, the execution behavior of the local system changes at time $\gamma_{h,\searrow}$, in which $r_k \leq \gamma_{h,\searrow} < f_k$.

There are two cases to be considered:

- **Case 1**: In the interval $[r_k, \gamma_{h,\searrow})$, the schedule σ does not idle at all.
- **Case 2**: In the interval $[r_k, \gamma_{h,\searrow})$, the schedule σ idles at some time prior to $\gamma_{h,\searrow}$.

We note that the schedule σ is busy from $\gamma_{h,\searrow}$ to f_k, since σ is a work-conserving schedule and there is no suspending behavior between $\gamma_{h,\searrow}$ and f_k under the service protocol.

Proof of Case 1: Let t be the earliest moment such that the schedule σ is busy from t to r_k. We note that such t exists. Under the above construction, the schedule σ is busy from t to f_k and idles right prior to t. If we alter the arrival time of the job τ_k from r_k to t, its response time becomes $f_k - t$, which is no less than $f_k - r_k$.

If the job τ_k^j is not offloaded, its execution time is at most $C_{k,1} + C_{k,s} + C_{k,2}$. If the job τ_k^j is offloaded successfully, its execution time is at most $C_{k,1} + p_k + q_k + C_{k,2}$. If the job τ_k^j is offloaded unsuccessfully, its execution time is at most $C_{k,1} + p_k + C_{k,s} + C_{k,2}$. Under the assumption that $p_k + q_k \leq C_{k,s}$ in Section 2, we know that its execution time is upper-bounded by the maximum of the above three scenarios, which is at most $C_{k,1} + p_k + C_{k,s} + C_{k,2} = p_k + C_k^\sharp$.

Since the local system idles prior to t under normal execution behavior, the interference of the higher-priority tasks can be derived from Lemma 3. Therefore, the worst-case response time of τ_k in this case is the minimum positive value of Δ such that

$$\Delta = p_k + C_k^\sharp + \sum_{\tau_i \in hp(\tau_k)} \max\{f_1(\tau_i, \Delta), f_2(\tau_i, \Delta)\} \tag{4}$$

Proof of Case 2: Let t' be the latest moment such that schedule σ is busy from t' to f_k. We note that such t' exists since the schedule idles at some moment in $[r_k, \gamma_{h,\searrow})$. By definition, $t' - r_k \leq R_k^1 + S_k$.

We now analyze an upper bound of $f_k - t'$. Since the schedule σ idles prior to time t', the job of τ_k must have been offloaded. If the offloading operation is successful, the execution time of task τ_k in the interval $[t', f_k)$ is at most $q_k + C_{k,2}$. If the job τ_k^j is offloaded unsuccessfully, its execution time in the interval $[t', f_k)$ is at most $C_{k,s} + C_{k,2}$. Under our assumption that $p_k + q_k \leq C_{k,s}$ in Section 2, we know that its execution time in the interval $[t', f_k)$ is at most $C_{k,s} + C_{k,2}$.

The interference of the higher-priority jobs in the time interval $[t', f_k)$ is obtained using Lemma 3. Since $t' - r_k \leq R_k^1 + S_k$ and the interference of a higher-priority task τ_i from t' to $t' + \Delta$ is at most $\max\{f_1(\tau_i, \Delta), f_2(\tau_i, \Delta)\}$, the worst-case response time of τ_k in Case 2 is $R_k^1 + S_k + \Delta$, where Δ is the minimum positive value with

$$\Delta = C_{k,s} + C_{k,2} + \sum_{\tau_i \in hp(\tau_k)} \max\{f_1(\tau_i, \Delta), f_2(\tau_i, \Delta)\} \tag{5}$$

Because $C_{k,s} + C_{k,2} \leq C_k^\sharp$, the worst-case response time obtained by Eq. (4) dominates the one from Eq. (5), which concludes this lemma. ◀

Having examined the case N2L under the service protocol, we subsequently consider the case L2L for a task τ_k in $\mathcal{T}_{crit}$ under analysis. The worst-case response time of task τ_k in this case can be determined by the following lemma:

▶ **Lemma 9.** *Under the service protocol, the response time $f_k - r_k$ in the case L2L is upper-bounded by the worst-case response time derived in Lemma 8 if $\Delta \leq T_k$.*

Proof. We note that the schedule σ is busy from r_k to f_k, since σ is a work-conserving schedule and there is no suspending behavior between r_k and f_k under the service protocol. Based on the schedule σ, we examine the following two moments[2]:

- Let t be the earliest moment such that schedule σ is busy from t to r_k.
- Let t' be the latest moment such that local system switches from normal to local execution behavior before or at r_k.

We note that both t and t' exist. There are two scenarios to be analyzed:

- $t \leq t'$: The local system exhibits normal execution behavior prior to time t'. We can change the release time of job τ_k to t' without decreasing its response time. Then, the analysis of Case 1 in Lemma 8 can be applied directly, since the worst-case execution time of τ_k is at most $C_k^\sharp$ in this scenario.
- $t > t'$: The schedule σ idles at $t - \varepsilon$ for an infinitesimal ε and the local system exhibits local execution behavior from t' to t. Therefore, the idle time in schedule σ is due to the removal of the lower-priority tasks from the original schedule. In this case, there exists no unfinished job of any $hp(\tau_k)$ at time t. All jobs released by τ_k and $hp(\tau_k)$ are executed locally. Therefore, the classical critical instant theorem by Liu and Layland [16] can be applied. The worst-case response time in this case is at most the minimum positive value of Δ, for which

$$\Delta = C_k^\sharp + \sum_{\tau_i \in hp(\tau_k)} \left\lceil \frac{\Delta}{T_i} \right\rceil C_i^\sharp \tag{6}$$

  if $\Delta \leq T_k$. This case is dominated by Eq. (4). ◀

Resulting from the above analyses, the schedulability of a task τ_k in $\mathcal{T}_{crit}$ under the service protocol can be verified by the following theorem:

▶ **Theorem 10.** *Consider the service protocol. Suppose that Definition 1 holds, i.e., every task τ_k in $\mathcal{T}$ meets its deadline under normal execution behavior. Every task τ_k in $\mathcal{T}_{crit}$ meets its deadline under local execution behavior if there exists a Δ with $0 < \Delta \leq D_k$ such that the condition in Eq. (3) holds.*

Proof. According to Lemma 9, the scenario L2L is dominated by N2L. By Lemma 6, L2N is not possible under both protocols in this paper. By Lemma 7, the worst-case response time due to N2N is at most R_k^{normal}. By Definition 1, $R_k^{normal} \leq D_i$. Therefore, the only condition to check the feasibility is to verify the scenario N2L based on Eq. (3) in Lemma 8. ◀

5.2 Analysis of the Return Protocol

The return protocol is designed to reduce the workload on the local system so that a faster transit from local to normal execution behavior is possible. Under the return protocol, the execution time of a job of τ_i in $\mathcal{T}_{non}$ on the local system is always no more than $C_i^\flat$

[2] We note that the schedule σ is already modified by removing lower-priority jobs. Therefore, it is possible that the reduced schedule idles but the local system exhibits local execution behavior.

independent of the system execution behavior. The analysis of the service protocol in Lemma 8 and Lemma 9 can be slightly changed to accommodate such workload reduction under the return protocol, as stated in the following lemmata:

▶ **Lemma 11.** *Under the return protocol, the response time $f_k - r_k$ in case N2L is upper bounded by the minimum positive value of Δ, for which*

$$\Delta = p_k + C_k^\sharp + \sum_{\tau_i \in hp(\tau_k) \subset \mathcal{T}_{crit}} \max\{f_1(\tau_i, \Delta), f_2(\tau_i, \Delta)\} + \sum_{\tau_i \in hp(\tau_k) \subset \mathcal{T}_{non}} \left(\left\lceil \frac{\Delta}{T_i} \right\rceil + 1 \right) C_i^\flat \tag{7}$$

if $\Delta \leq T_k$.

Proof. The proof is almost identical to the proof of Lemma 8 by considering different interferences of the higher-priority task τ_i using Lemma 3 when τ_i is in $\mathcal{T}_{crit}$ or Lemma 4 when τ_i is in $\mathcal{T}_{non}$. We note that the main argument in the proof of Lemma 8 that the local system is busy from $\gamma_{h,\searrow}$ to f_k remains valid, since task τ_k is in $\mathcal{T}_{crit}$ and cannot offload from $\gamma_{h,\searrow}$ to f_k. ◀

▶ **Lemma 12.** *Under the return protocol, the response time $f_k - r_k$ in the case L2L is upper bounded by the worst-case response time derived in Lemma 11 if $\Delta \leq T_k$.*

Proof. The proof is identical as a patch of Lemma 9 by considering different interferences of the higher-priority task τ_i using Lemma 3 when τ_i is in $\mathcal{T}_{crit}$ or Lemma 4 when τ_i is in $\mathcal{T}_{non}$. ◀

Resulting from the above analyses, the schedulability of a task τ_k in $\mathcal{T}_{crit}$ under the return protocol can be verified by the following theorem:

▶ **Theorem 13.** *Consider the return protocol. Suppose that Definition 1 holds, i.e., every task τ_k in $\mathcal{T}$ meets its deadline under normal execution behavior. Every task τ_k in $\mathcal{T}_{crit}$ meets its deadline under local execution behavior if there exists a Δ with $0 < \Delta \leq D_k$ such that the condition in Eq. (7) holds.*

Proof. According to Lemma 12, the case L2L is dominated by N2L. By Lemma 6, L2N is not possible under both protocols in this paper. By Lemma 7, the worst-case response time due to N2N is at most R_k^{normal}. By Definition 1, $R_k^{normal} \leq D_i$. Therefore, the only condition to check the feasibility is to verify the case N2L under the return protocol based on Eq. (7) in Lemma 11. ◀

6 Evaluation

To evaluate our proposed protocols, we perform comprehensive experiments using synthesized data as well as a case study regarding a robot. In the following, we first clarify the setup for both experiments in Sec. 6.1, before we discuss our findings in Sec. 6.2 and Sec. 6.3, respectively.

6.1 Experiment Setup

In our simulations based on synthetic data, we examine the system behavior under each protocol depending on different aspects, namely, 1a) the system utilization under normal execution behavior, 1b) the probability that an offloading operation is performed unsuccessfully, 1c) the percentage of critical tasks in the task set (CT), and 1d) the interval out of

which the task periods are generated. More precisely, we measure the amount of time the considered system exhibits local execution behavior in experiments 1a-I) and 1b) - 1d), and the number of synthesized task sets that pass the schedulability tests in Theorem 10 and Theorem 13, respectively, i.e., the so-called acceptance ratio, in experiment 1a-II). For the purpose of analyzing the system execution behavior, we developed an event-based miss rate simulator, which will be released together with the submitted paper.

For each scenario, i.e., 1a) - 1d)[3], we generate sets of 10 non-suspending sporadic tasks, whereat the task utilization values for a given system utilization[4] are generated by means of the UUniFast method [2]. The task periods in experiments 1a) - 1c) are specified according to a log-uniform distribution over the interval $[1ms, 100ms]$ (detailed explanations regarding this approach can be found in [8]) and according to a uniform distribution over the intervals $[1ms, 2ms], [1ms, 10ms], [1ms, 20ms], [1ms, 50ms]$, and $[1ms, 100ms]$ in experiment 1d). The worst-case execution time under normal execution behavior is given as $C_k = T_k \cdot U_k$. Moreover, we set deadlines as implicit, i.e., $D_k = T_k$. Thereon, we transform all non-suspending sporadic tasks into self-suspending tasks consisting of two computation segments as well as one suspension interval, as predefined in the system model in Section 2. For this purpose, we choose the length of each task's suspension interval according to a uniform random distribution out of the interval $S_k \in [0.01 \cdot (T_k - C_k), 0.1 \cdot (T_k - C_k)]$. To generate the corresponding local computation segments $c_{k,s}$, we choose a scaling factor $\alpha = 2$ such that $C_{k,s} = S_k \cdot \alpha$, whereas C_k is divided into $C_{k,1}$ and $C_{k,2}$. Priorities are assigned on the task-level according to the rate-monotonic (RM) approach. We assume the probability that a task τ_k offloads unsuccessfully to follow a Poisson distribution, i.e., $1 - exp(-\lambda \cdot S_k)$, which models that for an offloading operation with longer suspension time the probability of experiencing an unsuccessful offloading operation is higher. We set λ to $0.01 \cdot \frac{1}{ms}, 0.05 \cdot \frac{1}{ms}, 0.1 \cdot \frac{1}{ms}, 0.5 \cdot \frac{1}{ms}$, and $1 \cdot \frac{1}{ms}$ in experiment 1a) and 1b), and to $0.1 \cdot \frac{1}{ms}$ otherwise. The percentage of critical tasks in the system (CT) is set to $10\%, 20\%, 30\%, 40\%, 50\%$, and 60% in experiment 1c) and to 20% otherwise. Each experiment was repeated 100 times, except experiment 1a-I), which was repeated 10 times. For experiments 1a-I) and 1b)-1d), only task sets were considered that passed the schedulability tests in Theorem 10 and Theorem 13.

In experiment 2), we consider a Robotnik RB-1 Base robot platform [19], which is capable of performing loading operations of logistics objects in a highly unstructured environment, using the Robot Operating System (ROS) [20]. We simulated the navigation of the robot in a virtual map in a Gazebo-based environment [11] and measured the timing data of the `move_base` node during a time frame of 60 seconds using the Real-Time Scheduling Framework for ROS (ROSCH) [21] and RESCH [13]. More precisely, the execution times of the topics the `move_base` node, i.e., the main node used for the robot navigation in ROS, subscribed to, were measured, i.e., `/rb1_base/front_laser/scan`, which is the laser scanner data topic, `/rb1_base/robotnik_base_control/odom`, which is the odometry topic containing motor encoder readings and is used for localization, and /tf, which contains the transformations between different ROS 3D coordinate frames.

Resulting from this, three periodic, implicit-deadline tasks have been obtained, as shown in Table 2, which are transformed into self-suspending tasks analogously to the tasks in experiment 1), while considering the cases that 20%, 40%, and 60% of the task workload are

[3] Please note that we also tested other configurations, but since the results were similar, we restrain from discussing them in this section.

[4] Please note that the utilization indicated in all figures always refers to the utilization under normal execution behavior. The system utilization under local execution behavior is in all cases higher and varies depending on the properties of the individual task sets.

offloaded. Moreover, we assume that $\mathcal{T}_{crit} = \{\tau_{odom}\}$ and $\mathcal{T}_{non} = \{\tau_{laser}, \tau_{tf}\}$. We simulate the system behavior using the event-based miss rate simulator from experiments 1) with $\lambda = 0.1 \cdot \frac{1}{ms}$. For each offloading case, the simulation was repeated 100 times.

Table 2 Periodic, implicit-deadline tasks, measurements of a Robotnik RB-1 Base robot platform. Note that the frequency of task τ_{laser} is 15.5 Hz.

Task	Worst-Case Execution Time [ms]	Period [ms]
τ_{laser}	6.732	64.516
τ_{odom}	1.046	60.0
τ_{tf}	0.333	60.0

6.2 Simulation Results

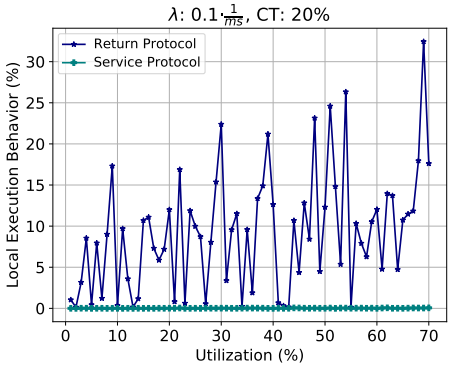


Figure 8 Experiment 1a-I): The percentage of time the system exhibits local execution behavior depending on the system utilization.

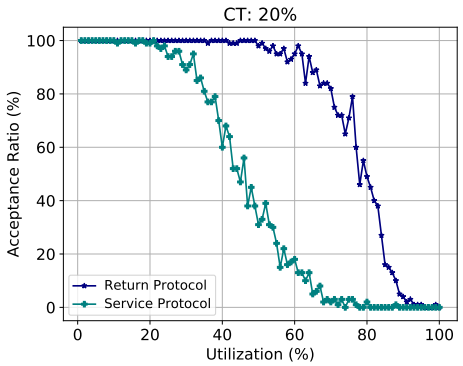
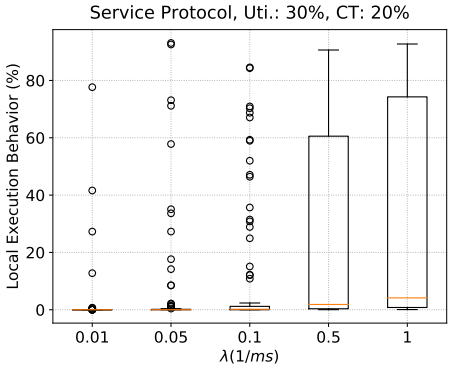
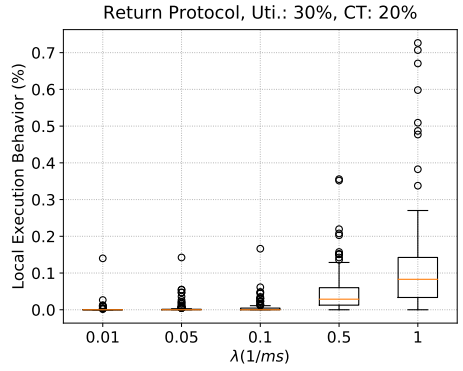


Figure 9 Experiment 1a-II): The acceptance ratios of the schedulability tests of the service and the return protocol.



(a) The percentage of local execution behavior for different probabilities of unsuccessful offloading operations under the service protocol.



(b) The percentage of local execution behavior for different probabilities of unsuccessful offloading operations under the return protocol.

Figure 10 Experiment 1b): The percentage of time the system exhibits local execution behavior during the simulation for different probabilities of unsuccessful offloading operations under the service and the return protocol with a system utilization of 30% and 20% critical tasks.

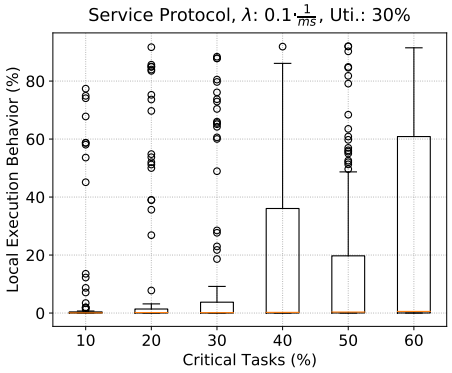
In Fig. 8, the results of experiment 1a-I) are depicted, namely, the mean value over all experiment repetitions of the percentage of simulation time, in which the system exhibits local execution behavior, as a function of the system utilization under normal execution behavior. While the percentage of time the system exhibits local execution behavior under the return protocol is always 0 or close to 0, the values under the service protocol vary largely between 0 and approximately 30%. From these results, we conclude that the percentage of time the system exhibits local execution behavior is not only dependent on the system utilization, but very likely also on other factors, which are discussed hereinafter.

As the outcome of experiment 1a-II), Fig. 9 portrays the acceptance ratios for the schedulability tests in Theorem 10 and Theorem 13, i.e., the percentage of generated task sets passing the schedulability tests for the service and the return protocol, respectively. The service protocol achieves an acceptance ratio of (close to) 100% until approximately 20% system utilization, which approaches 0% at approximately 70% system utilization. The acceptance ratio of the return protocol, in contrast, is (close to) 100% until approximately 40%, before it decreases and finally reaches 0% at approximately 95% system utilization. Please note that similar results were obtained under different configurations.

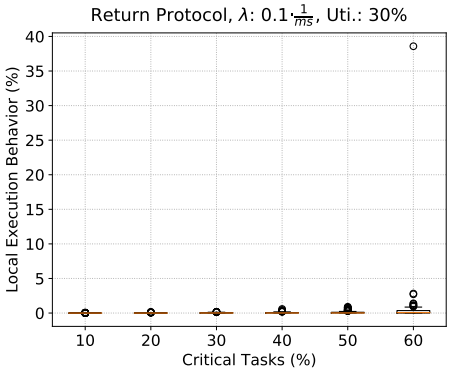
The following figures, i.e., Fig. 10 - Fig. 13, can be understood as follows: An orange line marks the median, whereas a box comprise three quartiles of the data points, i.e., the lower margin indicates the 25-percent-mark Q_1 and the upper margin indicates the 75-percent-mark Q_3. The distance between the 25- and 75-mark is denoted interquartile range IQ. The lower whisker indicates $Q_1 - 1.5 \cdot IQ$, the upper whisker $Q_3 + 1.5 \cdot IQ$, and circles mark outlier points.

In Fig. 10, the percentage of time the system exhibits local execution behavior under different probabilities of unsuccessful offloading operations, i.e., the outcome of experiment 1b), is presented with respect to the service protocol (cf. Fig. 10a) and the return protocol (cf. Fig. 10b). In general, despite some outliers, the time the system exhibits local execution behavior increases with an increasing probability of unsuccessful offloading operations. If λ is low, i.e., $0.01 \cdot \frac{1}{ms}$ and $0.05 \cdot \frac{1}{ms}$, both protocols lead in the majority of cases to a quite low percentage of time with local execution behavior. If, however, λ is high, i.e., $1 \cdot \frac{1}{ms}$, the service protocol leads to a significantly higher percentage of time under local execution behavior (median approximately 5%, Q_3 approximately 75%, upper whisker approximately 95%) than the return protocol (median approximately 0.08%, Q_3 approximately 0.15%, upper whisker approximately 0.28%). This follows from the different handling of non-critical tasks under the particular protocol if the system exhibits local execution behavior. The outliers can, in general, be explained by the fact that different tasks can suffer from unsuccessful offloading operations, leading to different consequences (consider, e.g., a task with a short period in contrast to a task with a long period).

Fig. 11 illustrates the percentage of time the system exhibits local execution behavior under different percentages of critical tasks in the system with respect to the service protocol (cf. Fig. 11a) and the return protocol (cf. Fig. 11b), i.e., the results of experiment 1c). It is evident that the percentage of time the system exhibits local execution behavior increases with an increasing percentage of critical tasks in the system, although the increase is larger under the service protocol than under the return protocol (except one outlier under a percentage of critical tasks of 60% in Fig. 11b). However, comparing the medians in Fig. 11a to those in Fig. 10a, it can be stated that the effect the percentage of critical tasks in the system has on the percentage the system exhibits local execution behavior is less than the impact of the probability of unsuccessful offloading operations.

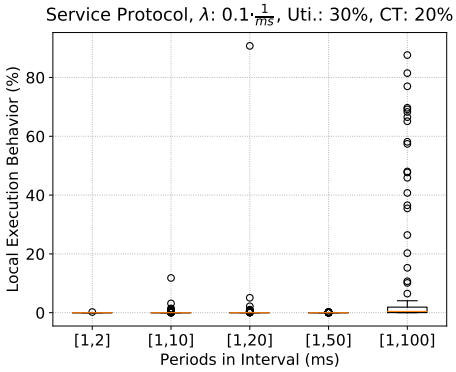


(a) The percentage of local execution behavior for different percentages of critical tasks in the system under the service protocol.

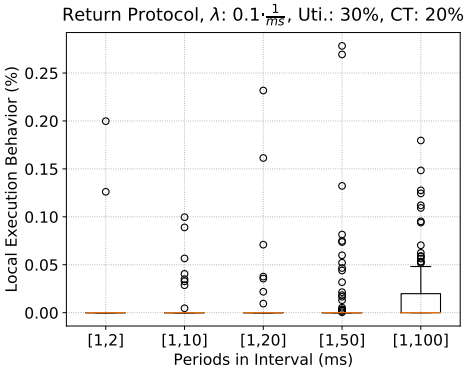


(b) The percentage of local execution behavior for different percentages of critical tasks in the system under the return protocol.

Figure 11 Experiment 1c): The percentage of time the system exhibits local execution behavior during the simulation for different percentages of critical tasks in the system under the service and the return protocol with a system utilization of 30% and $\lambda = 0.1 \cdot \frac{1}{ms}$.



(a) The percentage of local execution behavior for different period intervals under the service protocol.



(b) The percentage of local execution behavior for different period intervals under the return protocol.

Figure 12 Experiment 1d): The percentage of time the system exhibits local execution behavior during the simulation for different intervals used for the period generation with UUnifast under the service and the return protocol with a system utilization of 30%, 20% critical tasks and $\lambda = 0.1 \cdot \frac{1}{ms}$.

In Fig. 12, the percentage of time the system exhibits local execution behavior under different probabilities of unsuccessful offloading operations, i.e., the outcome of experiment 1d), is depicted with respect to the service protocol (cf. Fig. 12a) and the return protocol (cf. Fig. 12b). Under both protocols, no clear correlation is discernible between the percentage of time the system exhibits local execution behavior and the intervals out of which the task periods are generated except with respect to the interval $[1, 100]$. Although the medians are close to 0% under each protocol in this case, a slight increase of the percentage of time with local execution behavior is visible. As mentioned regarding the results of experiment 1b), this may result from widely differing task periods leading to an increased amount of time under local execution behavior if a task with a long period offloads unsuccessfully.

6.3 Case Study

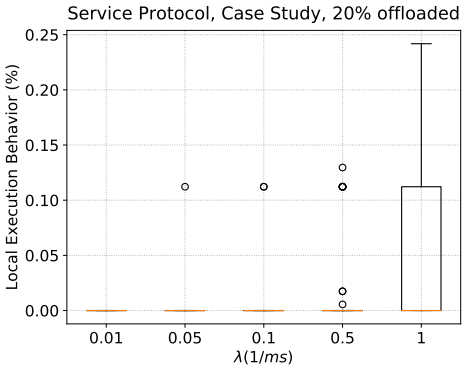
In Fig. 13, the results of experiment 2), i.e., of our case study considering the task set obtained from a Robotnik RB-1 Base robot (cf. Sec. 6.1), are visualized. From Fig. 13b, Fig. 13d, and Fig. 13f, it is discernible that the amount of offloaded workload per task has no significant impact on the percentage of time the system exhibits local execution behavior. Under the service protocol, in contrast, a clear increase of the time the system exhibits local execution behavior for higher probabilities of unsuccessful offloading operations, i.e. for $\lambda \in \left\{0.5 \cdot \frac{1}{ms}, 1 \cdot \frac{1}{ms}\right\}$, is visible with an increasing amount of offloaded workload per task (cf. Fig. 13a, Fig. 13c, and Fig. 13e). In consequence, it can be concluded that the amount of offloaded workload per task has strong impact on the system execution behavior under the service protocol and thus should be taken into consideration at system design time.

7 Conclusion

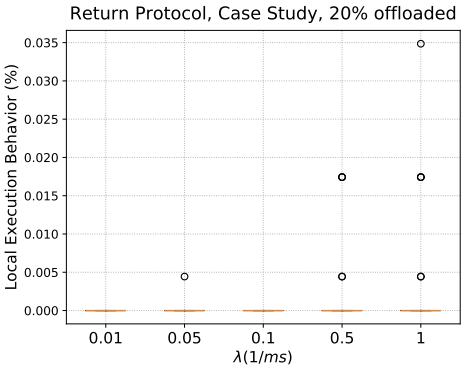
In this work, we proposed two protocols for cyber-physical systems by means of which critical and non-critical tasks can be offloaded safely, namely, the service protocol and the return protocol (cf. Sec. 3). We analyzed the worst-case timing behavior of the local cyber-physical system and, based on these analyses, we provided a sufficient schedulability test for each of the proposed protocols (cf. Sec. 4 and Sec. 5). In the course of comprehensive experiments and a case study involving a Robotnik RB-1 Base robot (cf. Sec. 6), we showed that our protocols have reasonable acceptance ratios under the provided schedulability tests. Moreover, we demonstrated that the system behavior under our proposed protocols depends on different factors, namely, on the probability of unsuccessful offloading operations, the percentage of critical tasks in the system, and the amount of offloaded workload. Not least, evidence was found that the percentage of time the system exhibits local execution behavior also depends on the actual task experiencing an unsuccessful offloading operation and, in consequence, also on the interval out of which task periods are chosen.

References

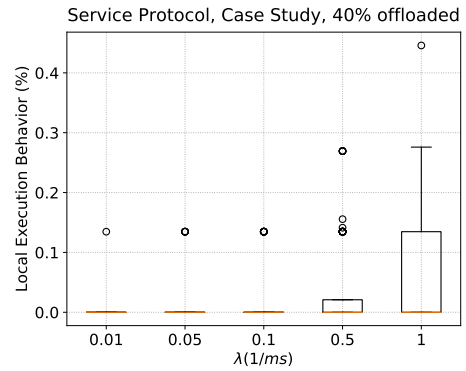
1. A. Aditthan, S. Ramesh, and S. Samii. Cloud-assisted control of ground vehicles using adaptive computation offloading techniques. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 589–592, 2018. doi:10.23919/DATE.2018.8342076.
2. Enrico Bini and Giorgio C Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.
3. Georg von der Brüggen, Kuan-Hsun Chen, Wen-Hung Huang, and Jian-Jia Chen. Systems with dynamic real-time guarantees in uncertain and faulty execution environments. In *Real-Time Systems Symposium (RTSS)*, Porto, Portugal, 2016. URL: https://ieeexplore.ieee.org/document/7809865.
4. Georg von der Brüggen, Lea Schönberger, and Jian-Jia Chen. Do nothing, but carefully: Fault tolerance with timing guarantees for multiprocessor systems devoid of online adaptation. In *The 23rd IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2018)*, Taipei, Taiwan, 2018. URL: https://ieeexplore.ieee.org/document/8639554.
5. Alan Burns and Robert I. Davis. A survey of research into mixed criticality systems. *ACM Comput. Surv.*, 50(6):82:1–82:37, 2018. doi:10.1145/3131347.
6. 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*, 2018. doi:10.1007/s11241-018-9316-9.



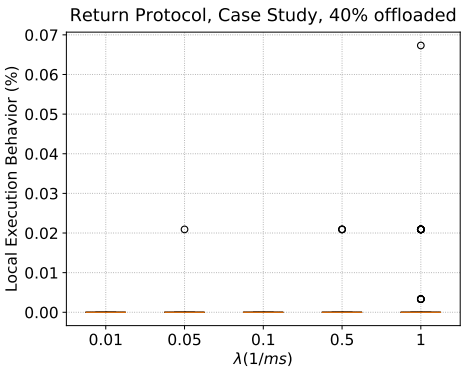
(a) The percentage of local execution behavior for different probabilities of unsuccessful offloading operations under the service protocol and 20% offloaded workload per task.



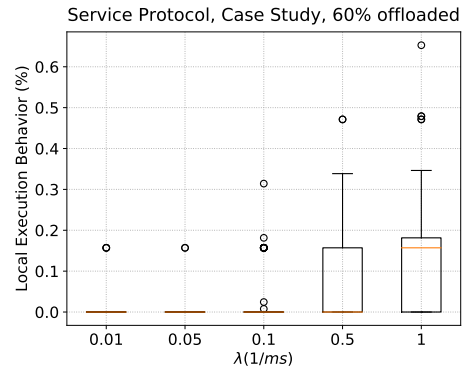
(b) The percentage of local execution behavior for different probabilities of unsuccessful offloading operations under the return protocol and 20% offloaded workload per task.



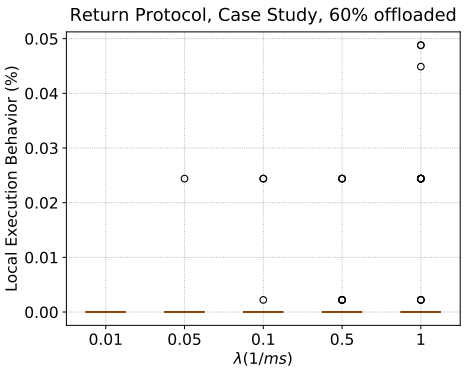
(c) The percentage of local execution behavior for different probabilities of unsuccessful offloading operations under the service protocol and 40% offloaded workload per task.



(d) The percentage of local execution behavior for different probabilities of unsuccessful offloading operations under the return protocol and 40% offloaded workload per task.



(e) The percentage of local execution behavior for different probabilities of unsuccessful offloading operations under the service protocol and 60% offloaded workload per task.



(f) The percentage of local execution behavior for different probabilities of unsuccessful offloading operations under the return protocol and 60% offloaded workload per task.

Figure 13 Experiment 2): The percentage of time the robot exhibits local execution behavior during the simulation for different probabilities of unsuccessful offloading operations and different percentages of offloaded workload under the service and the return protocol.

7 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*, pages 1–10, 2017. doi:10.1109/RTCSA.2017.8046321.

8 Robert I. Davis, Attila Zabos, and Alan Burns. Efficient exact schedulability tests for fixed priority real-time systems. *Computers, IEEE Transactions on*, 57(9):1261–1276, 2008.

9 Hasan Esen, Masakazu Adachi, Daniele Bernardini, Alberto Bemporad, Dominik Rost, and Jens Knodel. Control as a service (caas): Cloud-based software architecture for automotive control applications. In *Proceedings of the Second International Workshop on the Swarm at the Edge of the Cloud*, SWEC '15, page 13–18, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2756755.2756758.

10 Alexandre Esper, Geoffrey Nelissen, Vincent Nélis, and Eduardo Tovar. How realistic is the mixed-criticality real-time system model? In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS '15, page 139–148, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2834848.2834869.

11 Gazebo. Gazebo. robot simulation made easy. http://gazebosim.org/.

12 W.-H. Huang, J. Chen, and C. Liu. Pass: Priority assignment of real-time tasks with dynamic suspending behavior under fixed-priority scheduling. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2015. doi:10.1145/2744769.2744891.

13 Shinpei Kato, R Rajkumar, and Yutaka Ishikawa. A loadable real-time scheduler suite for multicore platforms. *Technical Report CMU-ECE-TR09-12*, 2009.

14 Swarun Kumar, Shyamnath Gollakota, and Dina Katabi. A cloud-assisted design for autonomous driving. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, page 41–46, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2342509.2342519.

15 C. Liu and J. Chen. Bursty-interference analysis techniques for analyzing complex real-time task models. In *Real-Time Systems Symposium (RTSS)*, pages 173–183, 2014.

16 C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973. doi:10.1145/321738.321743.

17 M. A. Maruf and A. Azim. Extending resources for avoiding overloads of mixed-criticality tasks in cyber-physical systems. *IET Cyber-Physical Systems: Theory Applications*, 5(1):60–70, 2020.

18 Institute of Electrical and Electronic Engineers. 802.11p-2010 - IEEE standard for information technology– local and metropolitan area networks– specific requirements– part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications amendment 6: Wireless access in vehicular environments. https://standards.ieee.org/standard/802_11p-2010.html, 2017.

19 Robotnik. Mobile robot RB-1 base. https://www.robotnik.eu/mobile-robots/rb-1-base-2/.

20 ROS. Robot operating system (ROS). https://www.ros.org/.

21 Y. Saito, F. Sato, T. Azumi, S. Kato, and N. Nishio. Rosch:real-time scheduling framework for ROS. In *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 52–58, 2018. doi:10.1109/RTCSA.2018.00015.

22 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 *The 24th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Hakodate, Japan, 2018.

23 Lea Schönberger, Wen-Hung Huang, Georg von der Brüggen, and Jian-Jia Chen. Schedulability analysis and priority assignment for segmented self-suspending tasks. Technical report, TU Dortmund, 2018.

24 Benjamin Sliwa, Robert Falkenberg, Thomas Liebig, Nico Piatkowski, and Christian Wietfeld. Boosting vehicle-to-cloud communication by machine learning-enabled context prediction. *IEEE Transactions on Intelligent Transportation Systems*, 2019.

25 Benjamin Sliwa and Christian Wietfeld. Empirical analysis of client-based network quality prediction in vehicular multi-MNO networks. In *2019 IEEE 90th Vehicular Technology Conference (VTC-Fall)*, Honolulu, Hawaii, USA, 2019.

26 S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 239–243, 2007. doi:10.1109/RTSS.2007.47.

27 Georg von der Brüggen, Wen-Hung Huang, and Jian-Jia Chen. Hybrid self-suspension models in real-time embedded systems. In *23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA*, pages 1–9, 2017. doi:10.1109/RTCSA.2017.8046328.

28 Georg von der Brüggen, Wen-Hung Huang, Jian-Jia Chen, and Cong Liu. Uniprocessor scheduling strategies for self-suspending task systems. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems (RTNS)*, pages 119–128, 2016. URL: http://dl.acm.org/ft_gateway.cfm?id=2997497&ftid=1804918&dwn=1&CFID=691780547&CFTOKEN=64912419.

The Time-Triggered Wireless Architecture

Romain Jacob
ETH Zurich, Switzerland

Licong Zhang
TU Munich, Germany

Marco Zimmerling
TU Dresden, Germany

Jan Beutel
ETH Zurich, Switzerland

Samarjit Chakraborty
University of North Carolina at Chapel Hill, NC, United States

Lothar Thiele
ETH Zurich, Switzerland

Abstract

Wirelessly interconnected sensors, actuators, and controllers promise greater flexibility, lower installation and maintenance costs, and higher robustness in harsh conditions than wired solutions. However, to facilitate the adoption of wireless communication in cyber-physical systems (CPS), the functional and non-functional properties must be similar to those known from wired architectures. We thus present Time-Triggered Wireless (TTW), a wireless architecture for multi-mode CPS that offers reliable communication with guarantees on end-to-end delays among distributed applications executing on low-cost, low-power embedded devices. We achieve this by exploiting the high reliability and deterministic behavior of a synchronous transmission based communication stack we design, and by coupling the timings of distributed task executions and message exchanges across the wireless network by solving a novel co-scheduling problem. While some of the concepts in TTW have existed for some time and TTW has already been successfully applied for feedback control and coordination of multiple mechanical systems with closed-loop stability guarantees, this paper presents the key algorithmic, scheduling, and networking mechanisms behind TTW, along with their experimental evaluation, which have not been known so far. TTW is open source and ready to use: `https://ttw.ethz.ch`.

2012 ACM Subject Classification Computer systems organization → Real-time system architecture; Computer systems organization → Sensors and actuators; Networks → Sensor networks

Keywords and phrases Time-triggered architecture, wireless bus, synchronous transmissions

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.19

Related Version Prior 4-page publication [25]: `https://doi.org/10.23919/DATE.2018.8342127`

Supplementary Material ECRTS 2020 Artifact Evaluation approved artifact available at `https://doi.org/10.4230/DARTS.6.1.5`.
Project webpage – TTW Artifacts – Authors' Contributions (CRediT)

Funding *Marco Zimmerling*: German Research Foundation (DFG) within the Emmy Noether project NextIoT (grant ZI 1635/2-1)
Lothar Thiele: Swiss National Science Foundation program "NCCR Automation"

1 Introduction

For decades, real-time distributed control systems have mostly relied on fieldbuses like CAN, FlexRay, and PROFIBUS. Following the design principles of the Time-Triggered Archi-

tecture (TTA) [31], these systems provide predictability of functional and non-functional properties. Yet, wired systems are increasingly reaching their limits as future cyber-physical systems (CPS) demand higher flexibility and cost efficiency. Low-power wireless communication promises to meet these demands by allowing for an unprecedented degree of mobility and deployment flexibility, avoiding cable breaks or faulty connections, and providing greater robustness to heat, humidity, abrasive substances, and undamped vibrations. However, to be viable for mission-critical CPS, a wireless system must feature timing predictability similar to traditional wired systems. Moreover, short end-to-end delays (tens of milliseconds), high reliability, and multiple years of battery lifetime are required for many applications [3].

The past years have seen significant progress in this direction. In particular, the concept of *synchronous transmissions* [14] has enabled highly reliable and efficient low-power wireless communication protocols that are robust to the unpredictable dynamics of wireless systems. As further detailed in [50], this is because of two main reasons: (*i*) synchronous transmissions enable the design of protocols whose logic is *independent of the time-varying network state*, which leads to a highly deterministic protocol execution regardless of changes in the network; (*ii*) synchronous-transmission-based communication protocols inherently exploit *different forms of diversity*, including sender and receiver diversity as well as temporal and spatial diversity, leading to a reliability as high as 99.9999 % in certain scenarios [14]. In fact, since its inception in 2016, all top three teams at the annual EWSN Dependability Competition have built on synchronous transmissions, demonstrating dependable multi-hop communication even in extreme wireless interference scenarios [40]. Building on this dependable base, protocols have been designed that provide sub-microsecond network-wide time synchronization while abstracting from the complexity of the underlying dynamic network topology, thus allowing to reason about a low-power wireless network as if it were a shared bus [13].

In light of this recent progress, we ask: *Is it possible to design a reliable, adaptive, and efficient time-triggered architecture for low-power wireless multi-hop networks with formal guarantees on end-to-end delays among distributed application tasks?* If so, it would take wireless systems a decisive step closer to wired systems by providing similar abstractions and guarantees as those researchers, engineers, and operators are used to, thus enabling powerful new CPS applications in industry, healthcare, energy, and many other domains.

Challenges. There are four key technical challenges standing in the way of a time-triggered architecture for low-power wireless multi-hop networks.

1. *Real-time communication in the face of radio duty cycling.* In a fieldbus, a node can listen idly without incurring costs, allowing it to react immediately to a request. In a low-power wireless bus [13], instead, a node must turn its radio off whenever possible to save energy, which renders the node unreachable until the next scheduled wake-up. To reduce energy costs due to idle listening, wireless protocols schedule communication rounds, where all nodes wake up, exchange messages, and go back to sleep (e.g., [13, 20, 22, 44]). Deciding when rounds take place and which nodes can send messages in each round to meet real-time deadlines at minimum energy costs is a complex scheduling problem; for example, the allocation of messages to rounds resembles combinatorial NP-hard bin packing [33].
2. *Coupling of communication and application tasks.* What ultimately matters in a CPS are the real-time constraints among application tasks executing on distributed devices. Hence, task executions and message exchanges over the wireless network must be coupled to guarantee end-to-end deadlines. A common approach for wired fieldbuses is to statically co-schedule all tasks and messages [2, 5, 12] to minimize delays by solving a satisfiability modulo theories (SMT) [11, 19, 41] or a mixed integer linear programming (MILP) [6]

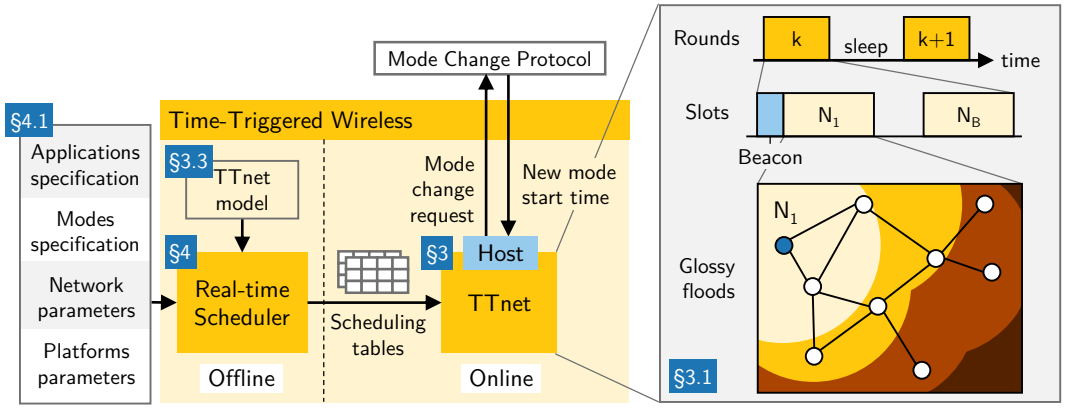


Figure 1 High-level overview of the Time-Triggered Wireless (TTW) architecture.

problem. To apply this approach to a wireless bus, we must embed the above-mentioned bin-packing problem into the schedule synthesis. This, however, introduces non-linear constraints that cannot be easily handled in a SMT or MILP formulation (see § 4.3).

3. *Dependencies across modes.* Static co-scheduling lacks runtime adaptability. This is often mitigated by enabling the system to switch at runtime between multiple operation modes, each having a pre-computed scheduling table [16]. The real-time guarantees that can be provided across mode changes depend on the mode-change protocol [9] and how the modes are scheduled. For example, if the periodicity of a task should be preserved across a mode change, this task must have the same schedule in both modes. A naïve approach to handle such dependencies across modes is a single MILP formulation with a global objective function. However, the poor scalability of the scheduling problem (NP-hard [27]) becomes a bottleneck for any realistic CPS scenario with many modes, while holding no guarantee that the computed schedules perform efficiently in terms of energy.
4. *Worst-case communication time.* The synthesis of scheduling tables requires an accurate model of the timing of all operations, including the worst-case execution time of tasks *and* the worst-case communication time of messages. While staple methods exist for tasks [45], a predictable protocol implementation is needed that yields accurate upper bounds on the time required for exchanging messages across a multi-hop network.

Contributions. This paper presents Time-Triggered Wireless (TTW), a wireless architecture for multi-mode CPS. By addressing all of the above challenges, TTW provides: (*i*) highly reliable and efficient communication across dynamic low-power wireless multi-hop networks; (*ii*) formal guarantees on end-to-end delays among distributed application tasks; (*iii*) runtime adaptability through mode changes that respect the tasks' periodicity constraints. With this, TTW does not only significantly advance the state of the art in real-time wireless systems, but also represents a solid foundation for the design of dependable wireless CPS.

As shown in Figure 1, TTW consists of two main components: a system-wide real-time scheduler that executes offline and a communication stack called TTnet that runs online on distributed low-power wireless devices. Based on the application specification (e.g., tasks, messages, modes) and system parameters (e.g., number of nodes, message sizes), the scheduler synthesizes optimized scheduling tables for the entire system. These tables are loaded onto the devices, and at runtime each device follows the table that corresponds to the current mode. TTnet provides a generic interface to implement different mode-change protocols [9].

Our design of the real-time scheduler uses concepts from network calculus [34] and novel heuristics to addresses challenges **1.**–**3.** Further, our design of TTnet addresses challenge **4.** by exploiting synchronous transmissions, in particular the deterministic behavior of Glossy floods [14]. As indicated for node N_1 in the bottom right corner of Figure 1, a Glossy flood sends a message to all nodes in a multi-hop network with a reliability that can exceed 99.9 % in certain networks and challenging conditions [13, 40], yet the time this process takes can be confined to a time slot of known length. TTnet groups a series of such time slots into rounds to save energy. This yields a highly timing-predictable protocol execution for which we devise timing and energy models as input for the TTW real-time scheduler.

We implement TTW on physical platforms and open-source our implementation [24]. Using this implementation, we perform real-world experiments on 27 low-power wireless nodes of the FlockLab testbed [35]. The results demonstrate that, for the settings we tested, our models are highly accurate and provide accurate upper bounds for the schedule synthesis; for example, our timing model overestimate the length of a round in TTnet by at most 0.7 ms. The results also show that our scheduling techniques can effectively reduce the energy cost of wireless communication in TTW and make the scheduling problem tractable: the solving time for one mode on a standard laptop PC ranges from a few seconds to a few minutes depending on the complexity of the mode (e.g., number of tasks and messages).

In addition, TTW already proved its utility for practical wireless CPS representative of emerging applications, such as remote control in chemical plants and cooperative robotics in smart manufacturing. Specifically, TTW was essential in enabling fast and reliable feedback control and coordination of multiple physical systems over low-power wireless multi-hop networks with closed-loop stability guarantees despite mode changes and mobile nodes [7, 36].[1] In these works, we integrated TTW with a mode-change protocol and analyzed the worst-case end-to-end jitter; thanks to the predictability of TTW, this jitter can be made negligible ($\leq 50\,\mu$s) for typical CPS applications, which we empirically validated (see [7, 36] for details).

Difference to prior paper. This paper significantly extends a prior 4-page publication [25] by (*i*) incorporating mode changes in the system model and the formulation of the scheduling problem, (*ii*) providing an implementation of TTW on physical platforms, and (*iii*) using this implementation to evaluate TTW and validate our models through real-world experiments.

2 Related Work

TTW is most closely related to prior work on reliable and predictable solutions for wireless sensor-actuator networks and real-ime distributed control systems based on fieldbuses.

In the wireless domain, numerous standards and protocols have been proposed for low-power multi-hop wireless networks, including WirelessHART, ISA100, and TSCH [44] from industry and several proposals from academia (e.g., [10, 39, 18]). Closest to TTW is Blink [49], which also builds on the concept of synchronous transmissions, in particular the Low-Power Wireless Bus (LWB) [13], to achieve adaptive real-time communication with guarantees on packet deadlines. While certainly important and useful, the key difference to TTW is that all these solutions consider only the network resources. They do not take into account the scheduling of application tasks on the distributed devices, and can therefore not influence the end-to-end delays and jitter that ultimately matter from an application's perspective. The

[1] Public demo: `https://youtu.be/AtULmfGkVCE`.
Mobility experiment: `https://youtu.be/19xPHjnobkY`.

only prior work that has looked at this end-to-end problem is DRP [26]. To provide end-to-end guarantees on packet deadlines, DRP decouples tasks and messages as much as possible, aiming for efficient support of sporadic or event-triggered applications. Compared with TTW, which tightly couples tasks and messages by co-scheduling them, DRP incurs significantly higher worst-case delays, and is thus not suitable for demanding CPS applications [3].

In the wired domain, a lot of work has been done on time-triggered architectures, such as the Time-Triggered Protocol (TTP) [32], the static-segment of FlexRay [15], and Time-Triggered Ethernet [30]. Like TTW, many recent works in this area also use SMT or MILP based methods to synthesize and/or analyze static (co-)schedules for those architectures [5, 12, 41, 43, 46]. The key difference to TTW, however, is that these approaches assume that a message can be scheduled at any point in time. While this is a perfectly valid assumption for a wired system, it is not compatible with the use of communication rounds in a wireless setting. § 5 shows that using rounds greatly reduces the energy consumed for communication, but it makes the schedule synthesis more complex, as detailed in § 4.

3 Communication Stack

We introduce TTW, a time-triggered architecture that supports the design and implementation of dependable wireless CPS. This section presents the design, implementation, and analytical models of TTW's communication stack called TTnet. TTW's real-time scheduler, described in § 4, uses these models to synthesize optimized scheduling tables.

3.1 TTnet Design

To support a wide spectrum of CPS applications possibly involving control and coordination of multiple physical systems over large distances, TTW requires a low-power wireless stack that provides reliable many-to-all communication over multiple hops. Further, the communication delays must be as short as possible and tightly bounded to support fast physical systems (e.g., mechanical systems with dynamics of tens of milliseconds). Finally, network-wide time synchronization is essential to reduce delays and achieve high performance and efficiency.

TTnet addresses these requirements by taking inspiration from the Low-Power Wireless Bus (LWB) [13] and improving latency and efficiency by using a different scheduling strategy. The basic communication scheme is illustrated on the right side of Figure 1, where distributed nodes are wirelessly interconnected and communicate using a sequence of Glossy floods [14]. As shown for when node N_1 sends its message, the flood spreads like a wave through the multi-hop network (note the different colors) so all nodes in the network can receive the message. During the flood, nodes that receive the message at the same time also retransmit the message at the same time. This technique, known as synchronous transmissions, is both fast (it achieves the theoretical minimum latency for one-to-all flooding) and extremely reliable [50]. Theoretical and empirical studies have shown that the few messages losses that do occur due to the vagaries of wireless communication are largely decorrelated, which greatly simplifies analytical modeling and control design [29, 48]. During a Glossy flood, the nodes also time-synchronize themselves with sub-microsecond accuracy [14]. Finally, since the protocol logic is independent of the network topology, it is highly robust to changes inside the network (e.g., due to mobile or failing devices) and external interference.

As shown in Figure 1, TTnet performs multiple Glossy floods in consecutive slots within a round; between two rounds all nodes have their wireless radio turned off to conserve energy. The duration of the slots and rounds as well as their associated energy costs can be accurately modeled (§ 3.3). Which nodes get to send their messages in each slot is determined by

TTW's real-time scheduler. The scheduler can abstract the entire multi-hop network with all its complexities and dynamics as a single shared resource with a known bandwidth, just like a wired fieldbus or a uniprocessor. This is because TTnet nodes appear to share a common clock and every messages is distributed to all nodes irrespective of the network topology.

TTnet exploits the fact that the schedules are computed offline to minimize communication delays. Instead of letting a dedicated host node compute the schedules between rounds and distribute them at the beginning of each round as in LWB, the host in TTnet only needs to send the ID of the current round in a short beacon (see Figure 1). The nodes can then use the ID to look up the corresponding scheduling table in their local memory. As a result, the rounds in TTnet are "more compact," which minimizes delays and improves efficiency.

The beacons also contain all information required to reliably switch between different operation modes at runtime. In addition to the round ID, each beacon carries a mode ID and a trigger bit. Modes and rounds have unique IDs with a known mapping of rounds to modes. By default, a beacon includes the current mode ID and the trigger bit is set to 0. A mode change happens in two phases. First, the mode change is announced by a beacon with the new mode ID. In a later round, the trigger bit is set to 1, which triggers the mode change; the new mode starts at the end of that round. This two-step procedure lets the nodes prepare for an upcoming mode change; for example, by stopping the execution of application tasks that are not present in the new mode. Hence, TTnet provides a generic interface to implement different mode-change protocols. The mode-change protocol itself is independent of TTnet and can be freely designed or chosen among existing protocols [9].

Sending beacons in every round ensures a safe operation of TTnet: If a node fails to receive one, it does not participate in the communication until it receives again a beacon. Thus, it is guaranteed that all participating nodes know the current round and mode IDs.

3.2 TTnet Implementation

We implement TTnet using Baloo [22], a design framework for communication stacks based on synchronous transmissions. Baloo provides a customizable round structure and allows the user to implement the high-level logic of a communication stack via a programming interface based on callback functions. Low-level operations such as time synchronization and radio control are handled by a middleware layer integrated in Baloo.

Our TTnet implementation uses the static configuration mode of Baloo. The beacons are implemented using the user-bytes field of Baloo's control packets with a payload size of 2 bytes. The implementation runs on any physical platform supported by Baloo. Because of the need for timing predictability in TTW, we focus on a platform that is based on the dual-processor platform (DPP) architecture [8]. The DPP combines two arbitrary processors with an interconnect called Bolt [42]. Bolt provides predictable asynchronous message passing between two processors using message queues with first-in-first-out (FIFO) semantics, one for each direction. This allows to dedicate each processor to a specific task, execute these tasks in parallel, and compared to traditional dual- or multi-core platforms the DPP offers formally verified bounds on the interference between the processors [42]. The specific platform we use is composed of a 32-bit ARM Cortex-M4 based MSP432P401R running at 48 MHz and a 16-bit CC430F5147 running at 13 MHz. The former is dedicated to the execution of application tasks (e.g., sensing, actuation, and control), while the latter is dedicated to TTnet using a low-power wireless radio operating at 250 kbps in the 868 MHz frequency band. Our implementation is open source and freely available; refer to [24] for details.

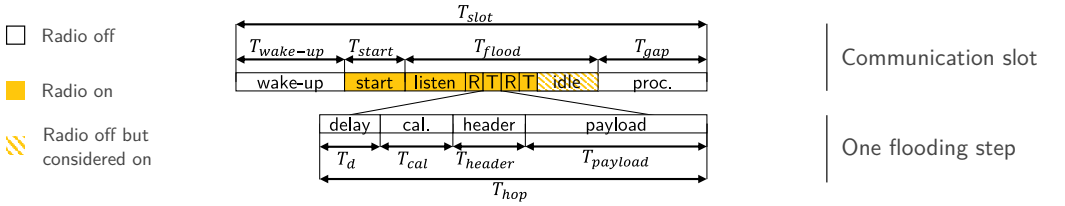


Figure 2 Detailed timings during a slot in TTnet. *At the slot level, the colored boxes identify phases where the radio is on. In the idle phase, the radio is off, but the duration depends on a node's distance to the initiator of the Glossy flood. To estimate the energy savings of rounds, we make the pessimistic assumption that the radio is on for T_{flood}, as specified in Equation* (7).

3.3 TTnet Model

The real-time scheduler in TTW requires timing and energy models of TTnet's operation to synthesize the scheduling tables. These models must be tight bounds to avoid delays and energy costs.

Let T_r be the length of a TTnet round. A round consists of a beacon slot plus up to B_{max} regular slots, in which Glossy floods are executed (see Figure 1). $T_r(L, B)$ denotes the length of a round with B regular slots having the same payload size L. The structure of a slot is illustrated in Figure 2. Accordingly, the length of a slot T_{slot} can be decomposed as follows

$$T_{slot} = T_{wake-up} + T_{start} + T_{flood} + T_{gap} \tag{1}$$

First, the nodes wake up ($T_{wake-up}$) and switch on their radio (T_{start}). Then, the Glossy flood executes (T_{flood}) followed by a small gap time (T_{gap}) in which the nodes can process the received packet. As explained in [47], the total length of a flood T_{flood} can be expressed by

$$T_{flood} = (H + 2N - 1) * T_{hop} \tag{2}$$

where H is the network diameter and N is the maximum number of times a node transmits during a Glossy flood. Let T_{hop} be the time needed for one protocol step, that is, the duration of one (synchronous) transmission during a flood. This quantity can be decomposed as

$$T_{hop} = T_d + T_{cal} + T_{header} + T_{payload} \tag{3}$$

where T_d is an initial radio delay, T_{cal} is the time needed to calibrate the radio clock (taking time equal to the transmission of L_{cal} bytes), and T_{header} and $T_{payload}$ are the times needed to transmit the packet header consisting of L_{header} bytes and the message payload consisting of L bytes. Using a radio with a transmit bitrate of R_{bit}, the transmission of L bytes takes

$$T(L) = 8L/R_{bit} \tag{4}$$

We divide the length of a slot T_{slot} into T^{on} and T^{off}, the times spent with the radio on and off, respectively. To this end, we make the conservative assumption that the radio stays on for the entirety of T_{flood}, as illustrated in Figure 2 and explained in the caption.

$$T_{slot}(L) = T^{off} + T^{on}(L) \tag{5}$$

$$T^{on}(L) = T_{start} + (H + 2N - 1) * (T_d + 8(L_{cal} + L_{header} + L)/R_{bit}) \tag{6}$$

$$T^{off} = T_{wake-up} + T_{gap} \tag{7}$$

With this, the length of a round with B regular slots and a common payload size L is

$$T_r(L, B) = T_{slot}(L_{beacon}) + B * T_{slot}(L) + T_{preprocess} \tag{8}$$

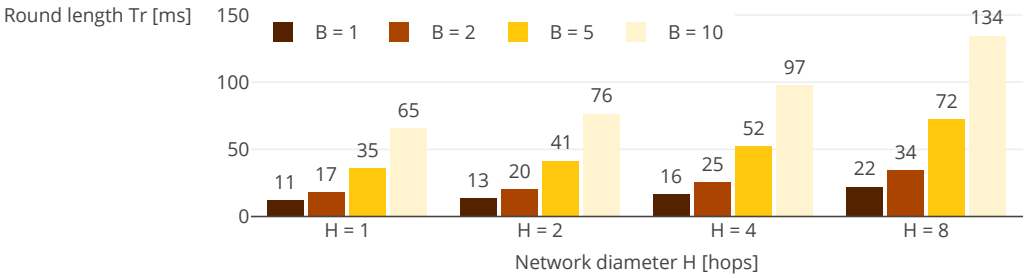


Figure 3 Example values of round length computed using the TTnet model for a payload size of 16 bytes and $N = 2$ transmissions during a Glossy flood. *Fewer slots lead to shorter rounds (e.g., 52 ms for $B = 5$ slots across a 4-hop network), which ultimately allows for shorter end-to-end delays.*

where L_{beacon} is the payload size of a beacon and $T_{preprocess}$ is the time needed by our TTnet implementation to prepare for an upcoming round (e.g., retrieve messages from send queue).

In addition to the timings of a TTnet round, we derive a model for the relative energy savings obtained by grouping multiple slots into the same round. As discussed in § 3.1, sending beacons is necessary to ensure a safe protocol operation. Without rounds, each regular message must still be preceded by a beacon to provide the same guarantee. In this case, the transmission time $T_{wo/r}$ for B regular messages of size L is given by

$$T_{wo/r}(L, B) = B * (T_{slot}(L_{beacon}) + T_{slot}(L)) \tag{9}$$

The relative energy savings obtained by a round-based design thus amount to

$$E = (T^{on}_{wo/r} - T^{on}_{r})/T^{on}_{wo/r} \tag{10}$$

with $T^{on}_{wo/r} = B * (T^{on}(L_{beacon}) + T^{on}(L))$ and $T^{on}_{r} = T^{on}(L_{beacon}) + B * T^{on}(L)$.

Using these expressions and the parameters measured for or used by our TTnet implementation (detailed in [24]), we can determine the round length T_r and the energy savings E for different number of slots per round B, message payload sizes L, network diameters H, and number of transmissions during a Glossy flood (N). For $L = 16$ byte and $N = 2$, Figure 3 plots the resulting round length T_r for different network diameters and slots per round. We can see, for example, that it takes less than 100 ms to send 10 messages in a 4-hop network. Real-world experiments using 27 distributed wireless nodes on a public testbed demonstrate that these models are very accurate (§ 5.1).

4 Real-Time Scheduler

Equipped with a timing-predictable implementation of the TTnet communication stack and a corresponding model that provides accurate upper bounds, we now turn to the second main component of TTW: the real-time scheduler. This section first defines the system model and scheduling problem, then describes the single-mode and multi-mode schedule synthesis.

4.1 System Model

Before we can formulate and solve the scheduling problem we face in TTW, we provide a formal system model of all relevant hardware and software components and their interactions.

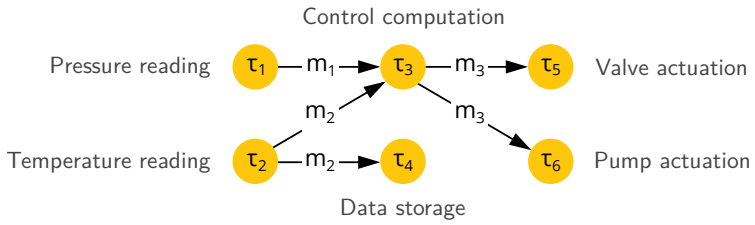


Figure 4 An example application and its precedence graph $\mathbb{P}$. *The application execution starts with sensing tasks τ_1 and τ_2. After messages m_1 and m_2 with the sensor readings are received by the controller, actuation values are computed (τ_3), sent to the actuators (m_3), and applied (τ_5 and τ_6).*

Platform. Each node in the system is a *platform* that executes tasks and exchanges messages using TTnet. All tasks have a known worst-case execution time (WCET) on that platform. Targeting state-of-the-art wireless CPS platforms with two processors (e.g., LPC541XX, VF3xxR, DPP [8]), each node can simultaneously execute tasks and exchange messages.

Applications. Let $\mathcal{A}$ denote the set of distributed *applications* in the system. An application is composed of *tasks* and *messages*, each with a unique task or message *id*, that are coupled by precedence constraints as described by a directed acyclic graph, where vertices and edges represent tasks and messages, respectively. We denote by $a.\mathbb{P}$ the *precedence graph* of application a (see Figure 4 for an example). Each application executes at a periodic interval $a.p$ called the *period*. An application execution is completed when all tasks in $\mathbb{P}$ have been executed. All tasks and messages in $a.\mathbb{P}$ share the same period $a.p$. If the same task belongs to applications with different periods, it can be equivalently modeled as two different tasks. Applications are subject to real-time constraints: an application *relative deadline*, denoted by $a.d$, represents the maximum tolerable *end-to-end delay* to complete the application execution. The deadline can be arbitrary, without a specific relation to the period $a.p$. Some applications may require to keep the same schedule (e.g., the same task offsets) when switching between different operation modes, for example, to guarantee the stability of control loops under arbitrary mode switches [7]. We call these *persistent applications* and denote them by $\mathcal{A}_P \subset \mathcal{A}$.

Tasks. We denote by $\mathcal{T}$ the set of all *tasks* in the system. A node executes at most one task at any point in time. Since interactions with the physical world like sensing and actuation should not be interrupted, we consider non-preemptive task scheduling. Each task τ is mapped to a given node $\tau.map$ on which it executes with WCET $\tau.e$. The *task offset* $\tau.o$ represents the start of the task execution relative to the beginning of the application execution. A task can have an arbitrary number of preceding messages, which must all be received before the task can start to execute. $\tau.prec$ denotes the set of preceding message *id*s. Within one application, each task is unique; however, the same task may belong to multiple applications (e.g., the same sensing task may source different feedback loops). If so, we consider that these applications have the same period.

Messages. Let $\mathcal{M}$ be the set of all *messages*. Every message m has at least one preceding task, that is, a task that needs to finish before the message can be transmitted. The set of preceding task *id*s is denoted by $m.prec$. The *message offset* $m.o$ relative to the beginning of the application execution represents the earliest time message m can be allocated to a TTnet round for transmission (i.e., after all preceding tasks are completed). The *message deadline*

Table 1 Inputs and outputs of the scheduling problem we solve in TTW.

Inputs	$\mathcal{N}$	Set of nodes in the system
	$\mathcal{A}$, $\mathcal{A}_P$	Set of applications and persistent applications (including periods, deadlines, and precedence graphs)
	$\mathcal{O}$	Set of operation modes (including mode priorities)
	$\mathbb{M}$	Mode graph
	$\tau.p$, $m.p$	Task and message periods, inherited from the application
	$\tau.map$	Mappings of tasks to nodes
	$\tau.e$	Task worst-case execution time (WCET) given $\tau.map$
	B_{max}	Maximum number of slots per round
	L_{max}	Maximum message payload size
	H	Network diameter (in number of hops)
	N	Number of transmissions in a Glossy flood [14]
Outputs	$\tau.o$	Task offsets
	$m.o$, $m.d$	Message offsets and deadlines
	$r.t$, $r.[B_{max}]$	Round starting times and allocation vectors

$m.d$ relative to the message offset represents the latest time the message transmission must be completed (i.e., the earliest offset of subsequent tasks). The payload of all messages is upper-bounded by L_{max}. Messages are not necessarily unique: multiple edges of $a.\mathbb{P}$ can be labeled with the same message m, which captures the case of multicast or broadcast transmissions (see Figure 4). If a message belongs to multiple applications, we consider that these applications have the same period.

Modes. We denote with $\mathcal{O}$ the set of operation *modes*. These modes represent mutually exclusive phases in the system execution (e.g., init, normal, and failure modes), each having its own schedule. A mode has a unique *priority* $M.prio$. We write $a \in M$ to denote that application a executes in mode M. When unambiguous, we use M to denote the set of all applications that execute in mode M. The *hyperperiod* LCM of mode M is the least common multiple of the periods of all applications in M. Possible transitions between modes at runtime are specified by the *mode graph* $\mathbb{M}$ (see Figure 6). The mode graph is undirectional, that is, a transition from M_i to M_j implies that it is also possible to switch from M_j to M_i.

Network and rounds. We denote with $\mathcal{N}$ the set of all *nodes* in the network. The following four network parameters must be specified by the user of the TTW architecture:

- L the payload size of regular messages, in bytes;
- B_{max} the maximum number of slots in a TTnet round;
- N the maximum number of transmissions of a node during a Glossy flood;
- H the estimated diameter of the network, in number of hops.

As explained in § 3, communication over the network is scheduled in *rounds* r. The schedule of mode M has R_M rounds. We consider TTnet rounds as atomic, that is, they cannot be interrupted. Thus, the ordering of messages in a round is irrelevant.

Round r contains B_r slots (at most B_{max}), each of which is allocated to a unique message m. The *allocation vector* $r.[B_r]$ contains the *id*s of the messages that are allocated to the slots in round r, where $r.B_s$ denotes the allocation of the s-th slot. The *starting time* $r.t$ is the start of the round relative to the beginning of the mode's hyperperiod. Using the models from § 3.3, the length of a round and its energy cost can be determined.

Algorithm 1 Pseudo-code of the single-mode schedule synthesis.

Input: mode M, $a \in M$, $\tau.map$, $\tau.e$, B_{max}, T_o
Output: $Sched(M)$
  $LCM = hyperperiod(M)$
  $R_{max} = floor(LCM/T_o)$
  $R_M = 0$
  while $R_M \leq R_{max}$ **do**
    formulate the MILP for mode M using R_M rounds
    ($Sched(M)$, *feasible*) = *solve*(MILP)
    if *feasible* **then return** $Sched(M)$
    end if
    $R_M = R_M + 1$
  end while
  return 'Problem infeasible'

4.2 Scheduling Problem

Based on our system model, we are now in the position to formally state the scheduling problem we have to solve in TTW. Given all modes, applications, task-to-node mappings, and WCETs, for each mode M the remaining variables define the *mode schedule* $Sched(M)$

$$Sched(M) = \left\{ \begin{array}{l|l} \tau.o,\ m.o,\ m.d & \forall\ a \in M,\ (\tau, m) \in a.\mathbb{P} \\ r_k.t,\ r_k.[B_{max}] & \forall\ k \in [1,\ R_M] \end{array} \right\}$$

Table 1 lists all inputs and outputs of the scheduling problem in TTW. A schedule for mode M is said to be *valid* if all applications executing in mode M meet their end-to-end deadlines. The scheduling problem to solve thus consists of finding valid schedules for all modes $M \in \mathcal{O}$ such that the following two objectives are met:

(O1) The number of communication rounds is minimized, which results in minimizing the energy required for wireless communication.

(O2) All persistent applications $\mathcal{A}_P \subset \mathcal{A}$ seamlessly switch between modes, that is, their schedules remain the same across all possible mode changes.

4.3 Single-Mode Schedule Synthesis

TTW statically synthesizes the schedule of all tasks, messages, and communication rounds by solving a MILP. We first look at the single-mode case, effectively showing how to achieve objective **(O1)**, whereas § 4.4 considers the multi-mode case and thus objective **(O2)**.

The schedule of a mode M is computed for one hyperperiod, after which it repeats itself. To minimize the number of rounds used while taming computational complexity, we solve the problem sequentially as described in Algorithm 1. Each MILP formulation considers a fixed number of rounds R_M to be scheduled, starting with $R_M = 0$. The number of rounds is incremented until a feasible solution is found or the maximum number of rounds R_{max} that fit into one hyperperiod is reached. So, even without an explicit objective function, Algorithm 1 guarantees by construction that if the problem is feasible, the synthesized schedule uses the minimum number of rounds, thereby minimizing the communication energy costs.

Each MILP formulation contains a set of classical scheduling constraints such as: precedence constraints between tasks and messages must be respected, the applications' end-to-end deadlines must be satisfied, nodes process at most one task simultaneously, communication rounds must not overlap, and rounds must not be allocated more then B_{max} messages. These constraints can be easily formulated based on our system model. However, we must also guarantee that the allocation of messages to rounds is valid. Specifically,

(C1) messages must be served in rounds that start after their release time;

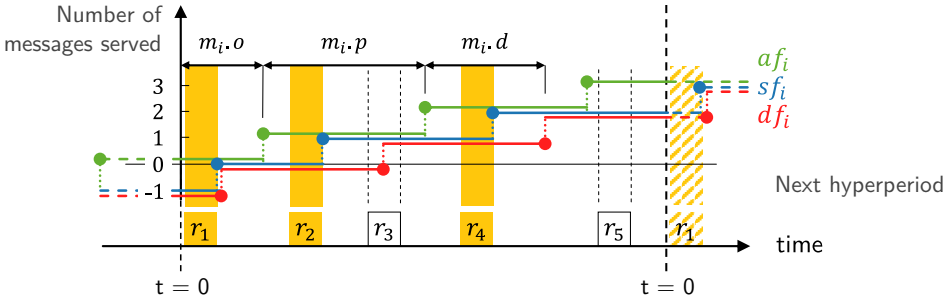


Figure 5 Example showing arrival (af), demand (df), and service (sf) functions for message m_i. *The lower part of the chart shows the five round, r_1 to r_5, that are scheduled within the hyperperiod. Message m_i is allocated a slot in the colored rounds, that is, in rounds r_1, r_2, and r_4. Allocating m_i to r_3 instead of r_2 would be invalid because r_3 does not finish before m_i's deadline, thus violating constraint* **(C2)**. *By contrast, allocating m_i to r_5 instead of r_1 would be valid and result in $r_0.B_i = 0$.*

(C2) messages must be served in rounds that finish before their deadline.

In other words, we must integrate the bin-packing problem that arises when allocating messages to rounds within the MILP formulation. This is a non-trivial challenge and a major difference compared with the existing approaches for wired architectures (e.g., [12]).

To address this challenge, we first formulate the constraints **(C1)** and **(C2)** using *arrival*, *demand*, and *service* functions – denoted by af, df, and sf– inspired by network calculus [34] (see Figure 5 for an illustration). These three functions count the number of message instances that are released, have passed deadlines, and have been served since the beginning of the hyperperiod. It must hold that

$$\forall\, m_i \in \mathcal{M},\ \forall\, t, \qquad df_i(t) \leq sf_i(t) \leq af_i(t) \tag{11}$$

$$\text{with} \qquad af_i :\ t \longmapsto \left\lfloor \frac{t - m_i.o}{m_i.p} \right\rfloor + 1 \tag{12}$$

$$\text{and} \qquad df_i :\ t \longmapsto \left\lceil \frac{t - m_i.o - m_i.d}{m_i.p} \right\rceil \tag{13}$$

However, as the service function sf stays constant between the rounds, we can formulate constraints **(C1)** and **(C2)** as follows: $\forall\, m_i \in \mathcal{M},\ \forall\, j \in [1..R_M]$,

$$\textbf{(C1)} \qquad sf_i(r_j.t + T_r) \ \leq\ af_i(r_j.t) \tag{14}$$

$$\textbf{(C2)} \qquad sf_i(r_j.t) \ \geq\ df_i(r_j.t + T_r) \tag{15}$$

The arrival and demand functions are step functions, which cannot be directly used in a MILP formulation. However, we observe that

$$\forall\, k \in \mathbb{N}, \qquad af_i(t) = k \quad \Leftrightarrow \quad 0 \leq t - m_i.o - (k-1) * m_i.p \ <\ m_i.p \tag{16}$$

$$\text{and} \qquad df_i(t) = k \quad \Leftrightarrow \quad 0 \ < \ t - m_i.o - m_i.d - (k-1) * m_i.p \leq\ m_i.p \tag{17}$$

This allows us to introduce, for each message $m_i \in \mathcal{M}$ and each round r_j, $j \in [1..R_M]$, two integer variables k^a_{ij} and k^d_{ij} that we constrain to take the values of af and df at the time points of interest, namely $r_j.t$ and $r_j.t + T_{r_j}$. More formally, we can write

$$0 \leq r_j.t - m_i.o - (k^a_{ij} - 1) * m_i.p \ <\ m_i.p \tag{18}$$

$$0 \ < \ r_j.t + T_{r_j} - m_i.o - m_i.d - (k^d_{ij} - 1) * m_i.p \leq\ m_i.p \tag{19}$$

$$\text{Thus,} \quad (18) \quad \Leftrightarrow \quad af_i(r_j.t) = k^a_{ij}$$

$$(19) \quad \Leftrightarrow \quad df_i(r_j.t + T_{r_j}) = k^d_{ij}$$

Finally, we must express the service function sf, which counts the number of message instances served by *the end* of each round. Recalling that $r_k.B_s$ denotes the *id* of the message that is allocated to the s-th slot of round r_k, we have that for any time $t \in [\, r_j.t + T_{r_j} \,;\, r_{j+1}.t + T_{r_j} \,[$ the number of instances of message m_i served is

$$\sum_{k=1}^{j} \sum_{s=1}^{B} r_k.B_s \quad s.t. \quad B_s = i$$

As visible in Figure 5, it can happen that $m.o + m.d > m.p$, resulting in $df(0) = -1$ according to Equation (13). This situation arises when a message is released at the end of a hyperperiod and thus has its deadline in the *next* hyperperiod. To account for this, we introduce for each message m_i a variable $r_0.B_i$ that is set to the number of such "leftover" message instances at $t = 0$. With this, for each $m_i \in \mathcal{M}$ and $t \in [\, r_j.t + T_{r_j} \,;\, r_{j+1}.t + T_{r_j} \,[$,

$$sf_i : t \longmapsto \sum_{\substack{k=1 \\ s.t.\ r_k.t + T_{r_k} < t}}^{j} \sum_{\substack{s=1 \\ s.t.\ B_s = i}}^{B} r_k.B_s - r_0.B_i \tag{20}$$

Based on the above reasoning, we can express constraints **(C1)** and **(C2)** in the MILP formulation using Equations (18) and (19) and the following two equations

$$(14) \quad \Leftrightarrow \quad \sum_{k=1}^{j} \sum_{\substack{s=1 \\ s.t.\ B_s = i}}^{B} r_k.B_s - r_0.B_i \;\leq\; k_{ij}^{a} \tag{21}$$

$$(15) \quad \Leftrightarrow \quad \sum_{k=1}^{j-1} \sum_{\substack{s=1 \\ s.t.\ B_s = i}}^{B} r_k.B_s - r_0.B_i \;\geq\; k_{ij}^{d} \tag{22}$$

4.4 Multi-Mode Schedule Synthesis

The multi-mode schedule synthesis is a multi-objective problem: As stated in § 4.2, the number of communication rounds in each mode should be minimized **(O1)**, while the schedule of all persistent applications should remain unchanged across mode changes **(O2)**. Objective **(O2)** induces dependencies between the different modes, which cannot be handled by solving a set of independent single-mode schedule synthesis problems.

A straightforward approach would be to synthesize the schedules of all modes at once based on a single MILP formulation. This approach, however, has two caveats. First, the resulting schedule synthesis problem is NP-hard [27] and thus scales poorly as the number of modes increases, which becomes a bottleneck for realistic CPS applications requiring a high degree of system adaptability. Second, a global objective function must be defined, such as minimizing the total number of rounds across all modes, which still gives no guarantee that the corresponding communication energy costs are effectively minimized: at runtime, if the system remains almost always in a certain mode, it may be better to minimize the number of rounds in that particular mode even if this implies a larger number of rounds overall.

To address these caveats, we solve the multi-mode schedule synthesis problem sequentially using heuristics, as commonly done in related approaches [28, 37, 41]. Algorithm 2 summarizes our approach. The modes are scheduled based on their priority $M.prio$. The mode with the highest priority (i.e., $M.prio = 1$) is scheduled first according to the single-mode synthesis outlined in Algorithm 1. Then, the mode with the second-highest priority (i.e., $M.prio = 2$)

Algorithm 2 Pseudo-code of the multi-mode schedule synthesis.

Input: Applications specification; modes specification; network parameters; system parameters
Output: { $Sched(M_i)$ for $M_i \in \mathcal{O}$ }

```
inheritance_constraints = ∅
for all M_i ∈ O in order of decreasing mode priority M_i.prio do
    add inheritance_constraints to mode M_i
    ( Sched(M_i) , feasible ) = single_mode_synthesis(M_i)
    if feasible then
        add Sched(M_i) to inheritance_constraints
    else
        return 'Problem infeasible'
    end if
end for
return { Sched(M_i) for M_i ∈ O }
```

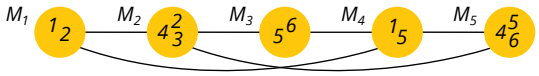


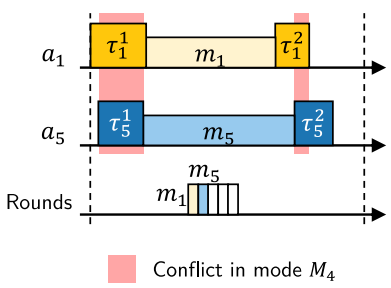
Figure 6 Mode graph $\mathbb{M}$ for Examples 1 and 6. *Five modes are depictd by circles and possible transitions between them by edges. Numbers inside each circle indicate which of the six applications, a_1 to a_6, execute in the corresponding mode, for example, $M_1 = \{a_1, a_2\}$. Mode M_i has priority i.*

is considered; however, its specification is first extended with *inheritance constraints* to guarantee that all persistent applications can seamlessly switch between all modes considered so far. The process repeats until all modes are scheduled.

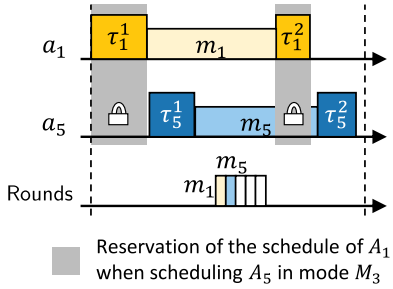
This sequential approach addresses the scalability of the multi-mode problem and provides a reasonable heuristic for minimizing the number of executed rounds and thus the communication energy costs. Indeed, these costs ultimately depend on how much time the system spends in each mode. It is reasonable to assume that an application domain expert knows in which modes the system likely operates most of the time. These modes are assigned a higher priority and therefore scheduled first by Algorithm 2. A mode with a lower priority is subject to more inheritance constraints, so it may schedule more than the minimal number of rounds to achieve **(O2)**. This may lead to a sub-optimal energy cost of lower-priority modes, which is nevertheless acceptable as the system likely spends less time in these modes.

In the following, we address the problem of deriving the set of inheritance constraints that are *necessary* and *sufficient* to guarantee objective **(O2)**. We first formalize the continuity constraints necessary to satisfy **(O2)** and characterize how these constraints may lead to conflicts between the mode schedules. Then we derive the minimally restrictive inheritance constraints that are necessary and sufficient to prevent such conflicts while satisfying **(O2)**.

▶ **Example 1.** Let us consider the mode graph in Figure 6 and assume that all applications are persistent. The modes are scheduled sequentially, starting with the highest-priority mode M_1, which is freely scheduled. When mode M_2 is scheduled, the schedule for application a_2 is inherited from mode M_1 to guarantee **(O2)** and the schedules for applications a_3 and a_4 are synthesized without constraints. In M_3, the specified applications a_5 and a_6 are new and can be scheduled without constraints. Then, in mode M_4, the specified applications a_1 and a_5 have been previously scheduled and thus must be inherited **(O2)**. However, as mode M_3 has been scheduled without constraint, the schedule synthesized for a_5 may be incompatible with that of a_1 from mode M_1. This leads to a conflict in M_4 and thus renders the sequential synthesis of the multi-mode problem infeasible, as illustrated in Figure 7.



(a) a_5 is scheduled in mode M_3 without considering the previously computed schedule of a_1, which leads to a conflict in mode M_4.



(b) a_5 is scheduled in mode M_3 considering the schedule of a_1 as reserved. A compatible schedule for a_5 is found, preventing a conflict in M_4.

Figure 7 Representations of the schedule of applications a_1 and a_5 from Example 1. For sake of illustration, we consider that all tasks are mapped to the same node. a_1 and a_5 are scheduled in modes M_1 and M_3, and must both be inherited in mode M_4. In Figure 7a, overlapping task schedules result in a conflict, while in Figure 7b this was prevented by reserving a_1's schedule.

4.4.1 Continuity Constraints

The schedule synthesis returns the application schedules (i.e., task and message offsets, and message deadlines) and the round schedules (i.e., starting times and allocation vectors). We represent an application schedule through a scheduling function s as defined below.

▶ **Definition 2** (Scheduling function). *The scheduling function s, defined over the set of applications $\mathcal{A}$, returns for a given application a all parameters characterizing the schedule of a. The schedule of application a is denoted by $s(a)$. $s_M(a)$ denotes the schedule of application a in mode M. The scheduling function is extended to sets of applications as follows*

$$\forall\, S \subset \mathcal{A}\,, \quad s(S) = \bigcup_{a \in S} s(a)$$

All persistent applications $a \in \mathcal{A}_P$ must keep the same schedule across mode changes, which leads us to the definition of continuity constraints.

▶ **Definition 3** (Continuity constraint). *The set of all continuity constraints is given by*

$$\forall\, a \in \mathcal{A}_P, \forall\, (M_i, M_j) \in \mathcal{O}^2,\ a \in M_i \wedge a \in M_j \ \wedge \mathbb{M}(M_i, M_j) = 1 \ \Rightarrow\ s_{M_i}(a) = s_{M_j}(a) \quad (23)$$

▶ **Definition 4** (Schedule domains). *The schedule domains of an application are the possibly multiple subsets of modes in which the application schedule must remain the same.*

▶ **Corollary 5.** *Two modes M_i and M_j belong to the same schedule domain of an application $a \in \mathcal{A}_P$ if and only if (i) a is scheduled in both modes, that is, $a \in M_i \ \wedge \ a \in M_j$, and (ii) there is a possible transition between the two modes, that is, $\mathbb{M}(M_i, M_j) = 1$.*

Proof. Multiple modes belong to the same schedule domain because of a continuity constraint. The formalization of the schedule domains directly follows from Definition 3 ◀

The schedule domains of an application can be extracted from the mode graph $\mathbb{M}$. One approach entails considering the sub-graph $\mathbb{G}_A$ of $\mathbb{M}$ that contains only the modes in which application a is specified – every connected component of $\mathbb{G}_A$ is a schedule domain of a.

Any non-persistent application that is present in multiple modes can be replaced by a distinct application with the same parameters in the respective modes. Similarly, any persistent application with multiple schedule domains can be replicated into distinct applications

having one scheduling domain each (illustrated by Example 6). This leads us to consider in the following that all applications are persistent and have a single schedule domain.

▶ **Example 6.** Consider again the mode graph in Figure 6. Application a_6 has two schedule domains, $\{M_3\}$ and $\{M_5\}$. This can also be modeled by two distinct applications $a_{6.3}$ and $a_{6.6}$ executing in M_3 and M_6. Application a_1 has only one schedule domain, $\{M_1, M_4\}$. If a_1 was not persistent, the continuity constraint would not apply and a_1 could be equivalently modeled by two distinct applications $a_{1.1}$ and $a_{1.4}$ executing in M_1 and M_4.

Continuity constraints can cause conflicts leading to the failure of the multi-mode synthesis although a solution may exist. In particular, if a given mode M belongs to the schedule domains of two different applications, which have been independently scheduled in higher-priority modes, there is a risk of conflict as the inherited schedules may be incompatible. We now formalize the notions of (virtual) legacy applications and conflicting modes. $\overline{X} = \mathcal{A} \setminus X$ denotes the complement of X. For each mode M_i, we define four application sets.

- **Known applications** are the applications previously scheduled in higher-priority modes. The set of known applications of mode M_i is denoted by $K_i = \cup_{j=1}^{i-1} M_j$.
- **Free applications** are the newly scheduled applications in mode M_i, that is, no higher-priority mode belongs to the schedule domain of these applications. The set of free applications of mode M_i is denoted by $F_i = M_i \cap (\mathcal{A} \setminus K_i) = M_i \cap \overline{K_i}$.
- **Legacy applications** are the applications previously scheduled in higher-priority modes that must be scheduled in mode M_i. Since applications have a single schedule domain, M_i necessarily belongs to the same schedule domain as these higher-priority modes and the legacy application schedules must be inherited. The set of legacy applications of mode M_i is denoted by $L_i = M_i \cap K_i$.
- **Virtual legacy applications** are the applications previously scheduled in higher-priority modes that are not scheduled in mode M_i. The set of virtual legacy applications of mode M_i is denoted by $VL_i = (\mathcal{A} \setminus M_i) \cap K_i = \overline{M_i} \cap K_i$. The virtual legacy applications of M_i are not executed in M_i; they simply have been scheduled in higher-priority modes. As illustrated in Example 1, it may be necessary to reserve the space for some of these virtual legacy applications to avoid future inheritance conflicts.

The schedules of two applications A and B are said to be in conflict if two tasks, one from A and one from B, are mapped to the same node and are scheduled in overlapping time intervals. We denote by $s(A) \cap s(B) \neq \emptyset$ the property that A and B are in conflict.

▶ **Definition 7** (Conflict-free). *A set of applications S is said to be conflict-free when there is no conflict between the schedules of the applications in S. We denote by $CF(S)$ the property that S is conflict-free, and $\overline{CF(S)}$ denotes that the set S is in conflict. Formally,*

$$CF(S) \quad \Leftrightarrow \quad \bigcap_{A \in S} s(A) = \emptyset$$

A mode is conflict-free if its legacy applications are conflict-free. In other words, $\forall M_i \in \mathcal{O}$,

$$CF(M_i) \quad \Leftrightarrow \quad CF(L_i)$$

The schedule $Sched(M_i)$ of mode M_i is valid only if $CF(M_i)$.

▶ **Corollary 8.** *A valid schedule for mode $M_i \in \mathcal{O}$ can only exist if the legacy applications of M_i are conflict-free, that is,*

$$CF(L_i) \Leftarrow \text{“Sched}(M_i)\text{ is feasible”}$$

Proof. Using Example 1 as a counter-example, $\overline{CF(L_4)}$ makes it impossible to derive a valid schedule for M_4. ◀

4.4.2 Minimal Inheritance Constraints

The single-mode schedule synthesis is complete: If the problem is feasible for mode M_i, then Algorithm 1 finds a valid schedule. In particular, the scheduled mode is conflict-free, $CF(M_i)$. In the multi-mode case, certain applications are subject to continuity constraints, which are satisfied by fixing the schedules of legacy applications L_i in the MILP formulation for mode M_i. However, by Corollary 8, this leads to a feasible schedule only if $CF(L_i)$.

Example 1 shows that inheriting legacy applications is not sufficient to prevent conflicts. Thus, we now derive the subset of virtual legacy applications VL_i of mode M_i that is necessary and sufficient to reserve in order to guarantee the absence of a conflict due to continuity constraints. More formally, our goal is

$$\forall\, k \in [1..i-1],\ \text{“}Sched(M_k) \text{ is feasible”} \quad \Rightarrow \quad CF(L_i) \tag{24}$$

To this end, we first formalize the constraints on the $Sched()$ function such that continuity constraints are enforced and conflicts are prevented as follows

$$\begin{aligned} &Sched : \mathcal{O} \longmapsto Sched(M) \\ &\quad s.t. \quad CF(M_i) \\ &\qquad \forall\, a \in L_i \cap M_j,\ j < i,\ s_{M_i}(a) = s_{M_j}(a) \\ &\qquad \forall\, a \in F_i,\ s(a) \cap s(\widetilde{VL_i}^a) = \emptyset \end{aligned} \tag{25}$$

The first constraint in Equation (25) ensures that the schedule is valid. The second one enforces the continuity constraints of applications. The third constraint enforces Equation (24) based on the idea that the free applications F_i in mode M_i must be compatible with the schedules of some virtual legacy applications. Theorem 9 derives for any free application the minimal set of such virtual legacy applications.

▶ **Theorem 9** (Minimal virtual legacy sets). *For mode M_i and free application $a \in F_i$, the minimal set of virtual legacy applications $\widetilde{VL_i}^a$ necessary and sufficient to satisfy* (24) *is*

$$\widetilde{VL_i}^a = \{X \in VL_i \mid \exists\, j > i, \quad a \in L_j \ \wedge \ X \in L_j\} \tag{26}$$

Proof. Please refer to Appendix A for the proof of this theorem. ◀

Finally, we set the sum of all message deadlines, $\sum_{m_i \in \mathcal{M}} m_i.d$, as objective function of the MILP solver. Maximizing this allows to limit the constraints inherited between the different modes and improves the schedulability of the multi-mode schedule synthesis problem.

5 Experimental Evaluation

Our experimental evaluation of TTW answers the following questions:

- Are our timing and energy models of the TTnet stack accurate (§ 5.1)?
- How big are the energy savings due to TTnet's round-based communication (§ 5.1)?
- Can the minimal inheritance constraints of TTW's real-time scheduler effectively reduce the number of rounds while respecting the persistency of applications (§ 5.2)?
- How long does it take to solve the multi-mode schedule synthesis problem (§ 5.3)?

All the evaluation artifacts (implementation, raw data, and scripts) are openly available [24].

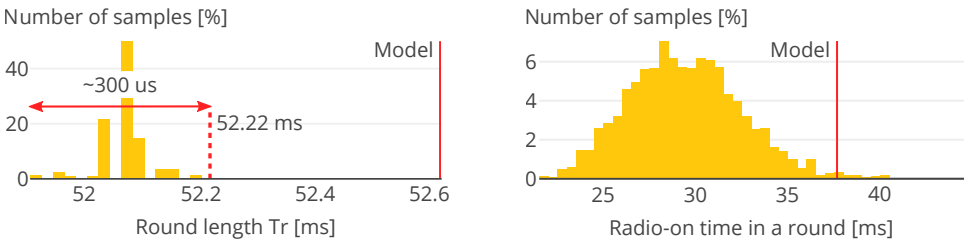


Figure 8 Distributions of round length (left) and radio-on time (right) measurements from all 27 nodes on the FlockLab testbed, collected in one series of 60 runs with 5 slots per round and a payload size of 16 bytes. *Our TTnet models are accurate; the timing model is empirically safe.*

5.1 TTnet Model Validation and Efficiency of Communication Rounds

We begin by validating our TTnet model and investigating the efficiency of rounds. Before discussing our results, we detail the evaluation scenario and the design of our experiments.

Evaluation scenario. Using 27 DPP nodes on the FlockLab testbed [1], we program TTnet to execute one round of B regular slots, followed by B rounds with one regular slot each. For each of these rounds, we collect the round length and the radio-on time. Both values are measured in software using a 32 kHz timer, which gives a measurement resolution of about 30 μs. For $H = 4$ and $N = 2$, we test different number of slots per round B and payload sizes L. For each setting, we measure the round length and radio-on time experienced by each individual node in the network, and we compare the results with our TTnet model.

Experimental design. We design our real-world experiments using TriScale [4], a framework that facilitates the reproducibility of networking evaluations by allowing to make performance claims with quantifiable confidence. TriScale distinguishes *metrics*, which are computed over one run of the evaluation scenario, and *key performance indicators* (KPIs), which capture the expected performance across a series of runs. Concretely, KPIs are percentiles of the underlying distribution of the metric estimated with a certain level of confidence [4].

We consider two performance dimensions: the empirical worst-case length of a round and the average per-node energy savings due to the use of rounds. As we are interested in the worst-case, we take the maximum measurement across all nodes as metric for one run. The true worst-case across any run is the 100th percentile of the metric distribution, which cannot be estimated with a finite number of runs. We thus take as KPI the 95th percentile of our round time metric with a desired confidence level of 95%. For the average energy savings, we use the median values across nodes as metric, and estimate the 5th percentile of that metric with 95% confidence. TriScale indicates that a minimum of 59 runs are needed for these estimations. It has been shown that the experimental conditions on FlockLab exhibit seasonal components [23]. Therefore, to avoid any bias in our results, we perform the series of runs over a span of one week during which we randomly schedule 60 runs for each setting. To investigate the reproducibility of the results, we perform 3 series of tests over the course of six months. We test our TTnet implementation with 5, 10, and 30 slots per round, and a payload of 8, 16, and 64 bytes. This results in a total of 1620 individual runs.

Results – round length. The experimental results in Table 2 confirm that our TTnet model provides a highly accurate estimate of the length of a round. The results are extremely stable across series: the largest difference in the KPI values corresponds to our measurement

Table 2 KPIs from the TTnet model validation for different series and the corresponding model estimates of the round length T_r and the energy savings E. †*marks series in which, due to construction work taking place in the building where FlockLab is deployed, the number of collected metric values is insufficient to compute the KPIs; the reported values are then the maximum round length and the minimum energy saving across all runs in the series.*

Payload	Slots per round	Round length T_r [ms]				Energy savings E [%]			
L [bytes]	B [.]	Series 1	Series 2	Series 3	Model	S. 1	S. 2	S. 3	Model
8	5	42.3	42.3	42.3†	42.52	25	31	29†	34
	10	77.21	77.24	77.24†	77.52	34	36	34†	38
	30	217.01	217.01	217.01	217.52	38	39	39	41
16	5	52.22	52.22	52.22†	52.52	25	24	22†	28
	10	97.04	97.08	97.08†	97.52	28	29	30†	32
	30	276.52	276.52	276.49†	277.52	32	33	32†	34
64	5	104.77	104.74	104.74†	105.02	8	8	9†	14
	10	202.09	202.12	202.09	202.52	8	12	12	16
	30	591.49	591.52	591.49†	592.52	13	11	14†	17

resolution of about 30 μs. Concretely, this means that the largest round length measured by any node is essentially the same. Furthermore, the measured KPI values are very close to and consistently lower than the model estimates. By definition of the KPI, we can claim that, with 95% probability, at least 95% of the runs of the evaluation scenario would yield a maximum round length that is less than or equal to the KPI value [4], and thus smaller than the model estimate. The left plot in Figure 8 shows the distribution of the round length measurements from all the nodes collected during one series of 60 runs. We observe that the distribution is very narrow (less than 300 μs of spread); this is because all operations in a TTnet round are time-triggered, so the measurement differences between nodes only come from the small differences in execution time of Baloo's end-of-round processing.

▶ Remark 10. The first series of test revealed a bug in the time synchronization software which led to an erratic behavior of certain nodes in some corner cases. This bug was fixed before the second series and the erratic data filtered out. In a previous version of this work [21], we presented a single case where one node measured a round time *larger* than the model value. After closer investigation, it appears that this too was a consequence of the time synchronization bug, which we failed to detect and filter. The analysis script has been adapted accordingly; please refer to the TTW artifacts for more details [24].

Results – energy savings. The results in Table 2 also show that the energy savings from the use of rounds in TTnet are significant: the savings are around 30% for a payload size of 16 bytes, which is representative of real-world CPS scenarios [17, 36, 38]. In general, the more slots are packed into a round (increasing B) and the smaller the payload size (decreasing L), the higher the energy savings. Since our energy KPI estimates the 5th percentile of the energy savings, we can claim that, with 95% probability, at least 95% of the runs in our evaluation scenario yield an average energy saving larger than or equal to the KPI value. The right plot in Figure 8 shows the distribution of radio-on time measurements from all the nodes, collected in one series of 60 runs. We see that the nodes experience significant differences in radio-on time during a round. This is expected since the nodes turn off their radio as soon as they have transmitted N times during a Glossy flood, which happens earlier for nodes that are closer to the initiator of the flood.

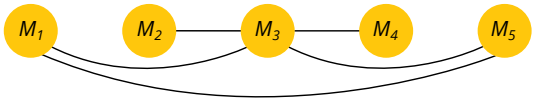


Figure 9 Mode graph used in the evaluation scenario of § 5.2.

Table 3 Solving times for each mode for three inheritance approaches (in seconds).

	M_1	M_2	M_3	M_4	M_5
No inheritance	5	≈ 0	7	294	≈ 0
Minimal inheritance	8	≈ 0	≈ 0	216	≈ 0
Full inheritance	3	33	2	43	578

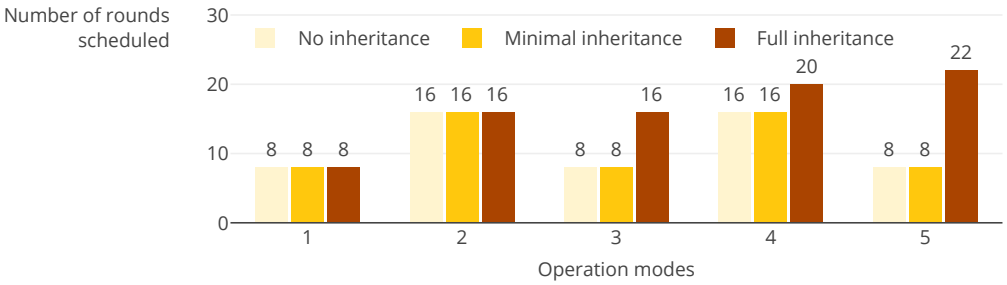


Figure 10 Number of rounds scheduled in each mode for three inheritance approaches. *We consider the number of rounds scheduled over 80 s, the least common multiple of the modes' hyperperiod.*

5.2 Effectiveness of TTW's Minimal Inheritance Approach

We now focus on TTW's real-time scheduler and investigate whether the minimal inheritance constraints from § 4.4.2 help keep the total number of scheduled rounds low, while guaranteeing that persistent applications keep the same schedule across mode changes.

Scenario. We consider a scenario with 13 nodes running 15 different persistent applications. There are in total 45 tasks and 30 messages. The periods and deadlines vary between 10 s and 80 s. The applications execute in 5 different modes; Figure 9 shows the mode graph with all possible transitions. We synthesize schedules for the 5 modes on a standard laptop PC. Besides our *minimal inheritance* approach, we consider two baselines for comparison: (*i*) *no inheritance*, which yields the minimum number of rounds under the (false) assumption that all applications are non-persistent; and (*ii*) *full inheritance*, which makes the (pessimistic) assumption that all applications executing in mode M_i are also executing in M_j. In contrast to *no inheritance*, the *full inheritance* baseline guarantees continuity across mode changes but may find problems to be non-schedulable although a feasible solution exists.

Results. Figure 10 shows for all approaches the number of rounds scheduled in each mode. We can see that with *full inheritance* the number of rounds steadily increases as it assumes that all previously scheduled application are still executing. This not only wastes energy, it also limits scalability as the number of modes grows. Our *minimal inheritance* approach performs and scales significantly better as it considers only the relevant constraints. In fact, in this particular scenario, *minimal inheritance* performs optimally: it does not schedule more rounds than the absolute minimum, which is captured by the *no inheritance* approach.

5.3 Offline Solving Time of TTW's Real-Time Scheduler

Although TTW targets CPS scenarios in which the scheduling tables are synthesized before the system operation starts (e.g., to a priori check that closed-loop stability can be guaranteed under given schedules [36, 7]), the solving time of the real-time scheduler is a relevant factor.

Scenario. Using the scenario from the previous experiment, we measure the solving time for the three different inheritance approaches on a standard laptop PC.

Results. We observe from Table 3 that the per-mode solving time ranges between less than a second and ten minutes, depending on the complexity of the mode. For example, modes M_3 and M_4 contain the most applications, leading to more constraints in the formulation. We also see that *minimal inheritance* does not increase the overall solving time compared to *no inheritance*. This is because, by reserving some applications' schedule, certain problem variables are fixed, which reduces the computational load. However, if too many variables are fixed, as shown by *full inheritance*, the resulting problem may become harder to solve: more rounds are required, which may increases the number of variables and thus the solving time.

6 Conclusions

This paper presented TTW, a time-triggered architecture for wireless CPS. TTW provides guarantees on end-to-end deadlines by statically co-scheduling all tasks and messages in the system, while supporting runtime adaptability via mode changes that respect the schedules of persistent applications. Our design of TTW's real-time scheduler addressed key challenges concerning the formulation and tractability of the scheduling problem. We leveraged synchronous transmissions to design a highly reliable, timing-predictable, and efficient low-power wireless communication stack that is robust to network dynamics. We believe that TTW takes wireless systems a major step closer the wired systems, which opens up several exciting opportunities for future CPS applications that seemed so far out of reach.

References

1 –. FlockLab. `http://flocklab.ethz.ch/`. Last accessed: 2020-04-14.

2 Tarek Abdelzaher and Kang Shin. Combined task and message scheduling in distributed real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 1999. `doi:10.1109/71.809575`.

3 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*, 2011. `doi:10.1109/INDIN.2011.6034912`.

4 Anonymous. TriScale: A Framework Supporting Reproducible Performance Evaluations in Networking. In *Zenodo*, 2020. `doi:10.5281/zenodo.3656819`.

5 Mohammad Ashjaei, Nima Khalilzad, Saad Mubeen, Moris Behnam, Ingo Sander, Luis Almeida, and Thomas Nolte. Designing end-to-end resource reservations in predictable distributed embedded systems. *Real-Time Systems*, 2017. `doi:10.1007/s11241-017-9283-6`.

6 Akramul Azim. *Scheduling of Overload-Tolerant Computation and Multi-Mode Communication in Real-Time Systems*. Doctoral Thesis, University of Waterloo, 2014. URL: `http://hdl.handle.net/10012/8973`.

7 Dominik Baumann, Fabian Mager, Romain Jacob, Lothar Thiele, Marco Zimmerling, and Sebastian Trimpe. Fast Feedback Control over Multi-hop Wireless Networks with Mode Changes and Stability Guarantees. *ACM Transactions on Cyber-Physical Systems*, 2019. `doi:10.1145/3361846`.

8 Jan Beutel, Roman Trüb, Reto Da Forno, Markus Wegmann, Tonio Gsell, Romain Jacob, Michael Keller, Felix Sutton, and Lothar Thiele. The Dual Processor Platform Architecture: Demo Abstract. In *Proceedings of the 18th International Conference on Information Processing in Sensor Networks*, IPSN '19, Montreal, Quebec, Canada, 2019. ACM. doi:10.1145/3302506.3312481.

9 Tianyang Chen and Linh T. X. Phan. SafeMC: A System for the Design and Evaluation of Mode-Change Protocols. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018. doi:10.1109/RTAS.2018.00021.

10 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*, 2011. doi:10.1109/ECRTS.2011.15.

11 Silviu S. Craciunas and Ramon Serna Oliver. SMT-based Task- and Network-level Static Schedule Generation for Time-Triggered Networked Systems. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, RTNS '14, Versaille, France, 2014. ACM. doi:10.1145/2659787.2659812.

12 Silviu S. Craciunas and Ramon Serna Oliver. Combined Task- and Network-level Scheduling for Distributed Time-triggered Systems. *Real-Time Systems*, 2016. doi:10.1007/s11241-015-9244-x.

13 Federico Ferrari, Marco Zimmerling, Luca Mottola, and Lothar Thiele. Low-power Wireless Bus. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, SenSys '12, New York, NY, USA, 2012. ACM. doi:10.1145/2426656.2426658.

14 Federico Ferrari, Marco Zimmerling, Lothar Thiele, and Olga Saukh. Efficient network flooding and time synchronization with Glossy. In *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*, 2011. URL: https://ieeexplore.ieee.org/document/5779066.

15 FlexRay. ISO 17458-1:2013–Road vehicles–FlexRay communications system–Part 1: General information and use case definition. Standard, International Organization for Standardization (ISO), Geneva, Switzerland, 2013. URL: https://www.iso.org/standard/59804.html.

16 Gerhard Fohler. Changing operational modes in the context of pre run-time scheduling. *IEICE transactions on information and systems*, 1993. URL: https://pdfs.semanticscholar.org/272b/615266e763369e903dcb0b966e22077f127c.pdf.

17 Samira Hayat, Evşen Yanmaz, and Raheeb Muzaffar. Survey on Unmanned Aerial Vehicle Networks for Civil Applications: A Communications Viewpoint. *IEEE Communications Surveys Tutorials*, Fourthquarter 2016. doi:10.1109/COMST.2016.2560343.

18 Tian He, John A. Stankovic, Chenyang Lu, and Tarek Abdelzaher. SPEED: A stateless protocol for real-time communication in sensor networks. In *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.*, 2003. doi:10.1109/ICDCS.2003.1203451.

19 Jia Huang, Jan Olaf Blech, Andreas Raabe, Christian Buckl, and Alois Knoll. Static scheduling of a Time-Triggered Network-on-Chip based on SMT solving. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2012. doi:10.1109/DATE.2012.6176522.

20 International Electrotechnical Commission (IEC). Industrial networks - Wireless communication network and communication profiles - WirelessHART. https://webstore.iec.ch/publication/24433. Last accessed: 2020-04-14.

21 Romain Jacob. *Leveraging Synchronous Transmissions for the Design of Real-Time Wireless Cyber-Physical Systems.* Doctoral Thesis, ETH Zurich, 2020. Accepted: 2020-02-26T08:48:57Z. doi:10.3929/ethz-b-000401717.

22 Romain Jacob, Jonas Bächli, Reto Da Forno, and Lothar Thiele. Synchronous Transmissions Made Easy: Design Your Network Stack with Baloo. In *Proceedings of the 2019 International Conference on Embedded Wireless Systems and Networks*, 2019. doi:10.3929/ethz-b-000324254.

23 Romain Jacob, Reto Da Forno, Roman Trüb, Andreas Biri, and Lothar Thiele. Wireless Link Quality Estimation on FlockLab - and Beyond, 2019. doi:10.5281/zenodo.3354717.

24 Romain Jacob and Licong Zhang. TTW Artifacts - Initial release, 2020. doi:10.5281/zenodo.3759222.

25 Romain Jacob, Licong Zhang, Marco Zimmerling, Jan Beutel, Samarjit Chakraborty, and Loth Thiele. TTW: A Time-Triggered Wireless design for CPS. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2018. doi:10.23919/DATE.2018.8342127.

26 Romain Jacob, Marco Zimmerling, Pengcheng Huang, Jan Beutel, and Lothar Thiele. End-to-End Real-Time Guarantees in Wireless Cyber-Physical Systems. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, 2016. doi:10.1109/RTSS.2016.025.

27 Kevin Jeffay, Donald F. Stanat, and Charles U. Martel. On non-preemptive scheduling of period and sporadic tasks. In *Proceedings Twelfth Real-Time Systems Symposium*, 1991. doi:10.1109/REAL.1991.160366.

28 Prachi Joshi, Haibo Zeng, Unmesh D. Bordoloi, Soheil Samii, S. S. Ravi, and Sandeep K. Shukla. The Multi-Domain Frame Packing Problem for CAN-FD. In Marko Bertogna, editor, *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECRTS.2017.12.

29 Jens Karschau, Marco Zimmerling, and Benjamin M. Friedrich. Renormalization group theory for percolation in time-varying networks. *Scientific Reports*, 2018. doi:10.1038/s41598-018-25363-2.

30 Hermann Kopetz, Astrit Ademaj, Petr Grillinger, and Klaus Steinhammer. The time-triggered Ethernet (TTE) design. In *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, 2005. doi:10.1109/ISORC.2005.56.

31 Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 2003. doi:10.1109/JPROC.2002.805821.

32 Hermann Kopetz and G. Grunsteidl. TTP - A time-triggered protocol for fault-tolerant real-time systems. In *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*, 1993. doi:10.1109/FTCS.1993.627355.

33 Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Algorithms and Combinatorics. Springer-Verlag, Berlin Heidelberg, third edition, 2006. doi:10.1007/3-540-29297-7.

34 Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Lecture Notes in Computer Science, Lect.Notes Computer. Tutorial. Springer-Verlag, Berlin Heidelberg, 2001. doi:10.1007/3-540-45318-0.

35 Roman Lim, Federico Ferrari, Marco Zimmerling, Christoph Walser, Philipp Sommer, and Jan Beutel. FlockLab: A Testbed for Distributed, Synchronized Tracing and Profiling of Wireless Embedded Systems. In *Proceedings of the 12th International Conference on Information Processing in Sensor Networks*, IPSN '13, New York, NY, USA, 2013. ACM. doi:10.1145/2461381.2461402.

36 Fabian Mager, Dominik Baumann, Romain Jacob, Lothar Thiele, Sebastian Trimpe, and Marco Zimmerling. Feedback Control Goes Wireless: Guaranteed Stability over Low-power Multi-hop Networks. In *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '19, Montreal, Quebec, Canada, 2019. ACM. doi:10.1145/3302509.3311046.

37 Francisco Pozo, Guillermo Rodriguez-Navas, Hans Hansson, and Wilfried Steiner. SMT-based synthesis of TTEthernet schedules: A performance study. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2015. doi:10.1109/SIES.2015.7185055.

38 James A. Preiss, Wolfgang Honig, Gaurav S. Sukhatme, and Nora Ayanian. Crazyswarm: A large nano-quadcopter swarm. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2017. doi:10.1109/ICRA.2017.7989376.

39 Abusayeed Saifullah, You Xu, Chenyang Lu, and Yixin Chen. Real-Time Scheduling for WirelessHART Networks. In *2010 31st IEEE Real-Time Systems Symposium*, 2010. doi:10.1109/RTSS.2010.41.

40 Markus Schuß, Carlo Alberto Boano, Manuel Weber, and Kay Römer. A Competition to Push the Dependability of Low-Power Wireless Protocols to the Edge. In *Proceedings of the 2017 International Conference on Embedded Wireless Systems and Networks*, EWSN '17, USA, 2017. Junction Publishing. doi:10.5555/3108009.3108018.

41 Wilfried Steiner. An Evaluation of SMT-Based Schedule Synthesis for Time-Triggered Multi-hop Networks. In *2010 31st IEEE Real-Time Systems Symposium*, 2010. doi:10.1109/RTSS.2010.25.

42 Felix Sutton, Marco Zimmerling, Reto Da Forno, Roman Lim, Tonio Gsell, Georgia Giannopoulou, Federico Ferrari, Jan Beutel, and Lothar Thiele. Bolt: A Stateful Processor Interconnect. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, SenSys '15, New York, NY, USA, 2015. ACM. doi:10.1145/2809695.2809706.

43 Domitian Tamas-Selicean, Paul Pop, and Wilfried Steiner. Synthesis of Communication Schedules for TTEthernet-based Mixed-criticality Systems. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, Tampere, Finland, 2012. ACM. doi:10.1145/2380445.2380518.

44 Watteyne Watteyne, Thomas, Pere Tuset-Peiro, Xavier Vilajosana, Sofie Pollin, and Bhaskar Krishnamachari. Teaching Communication Technologies and Standards for the Industrial IoT? Use 6TiSCH! *IEEE Communications Magazine*, 2017. doi:10.1109/MCOM.2017.1700013.

45 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 on Embedded Computing Systems (TECS)*, 2008. doi:10.1145/1347375.1347389.

46 Licong Zhang, Dip Goswami, Reinhard Schneider, and Samarjit Chakraborty. Task- and network-level schedule co-synthesis of Ethernet-based time-triggered systems. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2014. doi:10.1109/ASPDAC.2014.6742876.

47 Marco Zimmerling. *End-to-End Predictability and Efficiency in Low-Power Wireless Networks*. Doctoral Thesis, ETH Zurich, Zürich, 2015. doi:10.3929/ethz-a-010531577.

48 Marco Zimmerling, Federico Ferrari, Luca Mottola, and Lothar Thiele. On Modeling Low-Power Wireless Protocols Based on Synchronous Packet Transmissions. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems(MASCOTS)*, 2013. doi:10.1109/MASCOTS.2013.76.

49 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*, 2017. doi:10.1145/3012005.

50 Marco Zimmerling, Luca Mottola, and Silvia Santini. Synchronous Transmissions in Low-Power Wireless: A Survey of Communication Protocols and Network Services. *arXiv:2001.08557 [cs, eess]*, 2020. arXiv:2001.08557.

A Proof of Theorem 9

We first prove by recurrence that virtual legacy sets defined in (26) are sufficient to satisfy (24).

For the highest priority mode M_1, by definition, $L_1 = \emptyset$, thus $CF(L_1)$. Let us assume that for any $k \in [1..i]$, $Sched(M_k)$ is feasible in the sense of (25). This induces that $CF(S_k)$, hence $CF(L_k)$ for any $k \in [1..i]$. Let us finally assume that L_{i+1} is *not* conflict-free; that is,

$$\overline{CF(L_{i+1})} \quad \Leftrightarrow \quad \bigcap_{A \in L_{i+1}} s(A) \neq \emptyset \quad \Rightarrow \quad \exists\, (A,B) \in {L_{i+1}}^2,\ s(A) \cap s(B) \neq \emptyset \tag{27}$$

$$\Rightarrow \quad \begin{cases} \exists!\, M_a,\ A \in F_a \ \wedge\ a < i+1 \\ \exists!\, M_b,\ B \in F_b \ \wedge\ b < i+1 \end{cases} \tag{28}$$

where ∃! means "there exists a unique." Without loss of generality, we consider $a \leq b$. If $a = b$, then $S_a = S_b$ and $CF(S_a) \equiv CF(S_b)$. Therefore,

$$\bigcap_{A \in S_a = S_b} s(A) = \emptyset \quad \Rightarrow \quad s(A) \cap s(B) = \emptyset \tag{29}$$

This contradicts (27), thus necessarily $a < b$; in other words, mode M_a has higher priority than mode M_b. Therefore, a belongs either to L_b or VL_b by definition of those sets. By hypothesis, $CF(S_b)$ and $(28) : B \in F_b$, thus

$$A \in L_b \quad \Rightarrow \quad s(A) \cap s(B) = \emptyset$$

which again contradicts (27). Hence necessarily, $A \in VL_b$. Furthermore,

$$\left.\begin{array}{l}(28) : i+1 > b \\ (27) : A \in L_{i+1} \\ (27) : B \in L_{i+1}\end{array}\right\} \text{ Taking } b = i \text{ and } j = i+1, \quad (26) \ : \ A \in \widetilde{VL_b{}^B} \tag{30}$$

By hypothesis, $Sched(M_b)$ is feasible, thus $s(B) \cap s(\widetilde{VL_b{}^B}) = \emptyset$, which yields $s(B) \cap s(A) = \emptyset$ and contradicts (27) again. Therefore, the recurrence hypothesis is necessarily false. Hence, if for any $k \in [1..i]$, $Sched(M_k)$ is feasible in the sense of (25), then $CF(L_{i+1})$. By recurrence, we can conclude that the virtual legacy sets as defined by (26) are sufficient to satisfy (24).

We now prove that the virtual legacy sets are also necessary. Let us consider smaller virtual legacy sets than defined by (26), that is, $\exists\, i \in [1..M],\ a \in F_i,\ \widehat{VL_i{}^A} \not\subseteq \widetilde{VL_i{}^A}$. Let us further assume that $Sched()$ is redefined to replace $\widetilde{VL}$ by $\widehat{VL}$. By hypothesis,

$$\exists\, X \in \mathcal{A},\ X \in \widehat{VL_i{}^A} \ \wedge \ X \notin \widetilde{VL_i{}^A} \tag{31}$$

$$\text{Furthermore,} \quad X \in \widetilde{VL_i{}^A} \quad \Rightarrow \quad X \in VL_i \quad \Rightarrow \quad X \notin S_i \tag{32}$$

$$X \in \widetilde{VL_i{}^A} \quad \Rightarrow \quad \exists\, j > i, \quad a \in L_j \ \wedge \ X \in L_j \tag{33}$$

Assuming that $Sched(M_i)$ is feasible, the resulting schedule guarantees that $CF(M_i)$ and $\forall\, a \in F_i,\ s(a) \cap s(\widehat{VL_i{}^A}) = \emptyset$. However, $(31) : X \notin \widehat{VL_i{}^A}$ and $(32) : X \notin S_i$. Hence, schedule $s(a)$ may be synthesized such that $s(A) \cap s(X) \neq \emptyset$. According to $(33) : (A, X) \in {L_j}^2$, which induces a conflict in mode M_j. Hence, we can conclude that no sets $\widehat{VL}$ smaller than $\widetilde{VL}$ are sufficient to satisfy (24).

Overall, this shows that the virtual legacy sets $\widetilde{VL}$ as defined in (26) are both necessary and sufficient for the schedule synthesis method to satisfy (24), that is, to guarantee that inheritance of schedules from legacy applications does not lead to conflicts in lower-priority modes. In other words, $\widetilde{VL}$ from (26) defines the sets of minimally restrictive constraints such that $Sched()$ as defined in (25) satisfies (24). This completes the proof of Theorem 9.

Evaluation of the Age Latency of a Real-Time Communicating System Using the LET Paradigm

Alix Munier Kordon
Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
Alix.Munier@lip6.fr

Ning Tang
Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
Ning.Tang@lip6.fr

Abstract

Automotive and avionics embedded systems are usually composed of several tasks that are subject to complex timing constraints. In this context, the LET paradigm was introduced to improve the determinism of a system of tasks that communicate data through shared variables. The age latency corresponds to the maximum time for the propagation of data in these systems. Its precise evaluation is an important and challenging question for the design of these systems.

We consider in this paper a set of multi-periodic tasks that communicate data following the LET paradigm. Our main contribution is the development of mathematical and algorithmic tools to model precisely the dependency between tasks executions to experiment with an original methodology for computing the age latency of the system. These tools allow to handle the whole graph instead of particular chains and to extract automatically the critical parts of the graph. Experiments on randomly generated graphs indicate that systems with up to 90 periodic tasks and a hyperperiod bounded by 100 can be handled within a reasonable amount of time.

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

Keywords and phrases Real-Time Systems, Logical Execution Time, Age Latency

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.20

1 Introduction

A real-time system is a system that responds in a timely fashion to external events created by its environment [18]. In various contexts such as avionics or automotive, these systems must verify hard timing constraints. Their design and analysis are usually complex processes that require efficient methods.

We consider in this paper a set $\mathcal{T}$ of periodic tasks with different periods that are executed following the model of Liu and Layland [19]. A directed acyclic graph $\mathcal{G} = (\mathcal{T}, E)$ defines communication links between task executions. Each arc $(t_i, t_j) \in E$ between the two tasks t_i and t_j is associated to a shared memory variable that is modified by t_i and read by t_j. We assume that each execution of t_i updates the variable at its completion time, while each execution of t_j reads it at its starting time. This communication scheme, usually known as "implicit communication" follows the AUTOSAR requirements [1] and is commonly used for the design of automotive real-time systems.

However, the instants of the exchanges between tasks depend on the successive starting and completion times of the tasks, and are thus not predictable. The Logical Execution Time (LET) paradigm [15] delays writes to the periodic deadlines of the tasks and advances reads to their periodic release dates. The communication instants are then fixed before the execution of the tasks and the system is deterministic. This communication scheme was implemented by the time-triggered language Giotto [12]. This timing predictability makes it particularly suitable for safety-critical applications. This model was thus considered in

industrial domains like automotive [4, 10] and avionics [13, 23]. We suppose in this paper that tasks are periodic with different periods and that all communications follow the LET paradigm.

A real-time system usually communicates with its environment through sensors that detect events and actuators that transduce its reactions. Paths from a sensor to an actuator are usually referred to as event chains (see, for example, [10]). The time needed to propagate data from a sensor to an actuator is closely related to the reaction delay of the system. Several measures can be defined to capture these delays, as presented by Feiertag et al. [8]. We limit our study to the age latency, also called the end-to-end latency, which is the maximum time interval from a specific input value on a sensor to the last corresponding output value. It can be interpreted as the maximum delay that a specific data element spends in the system. This value measures the freshness of data producing a response of the system, and ensures that the actions of actuators are not too old.

The main contribution of the paper is to develop a general framework to model communications on successive task executions using LET communications for a general task dependency graph. The computation of the age latency of the application can then be seen as an example of a concrete application. This value cannot be defined in the presence of cycles in the dependency graph, thus graphs are assumed to be cycle-free. However, the transformations presented in this paper can be considered for general graphs. Observe that most of authors limit their methods to a single event chain [2, 8, 20].

Indeed, we first prove that dependencies induced by a LET communication $e = (t_i, t_j) \in E$ between the successive executions of t_i and t_j can be modelled by an original simple inequality involving parameters of the tasks t_i and t_j and the execution numbers considered.

Then, it can be observed that, if T_i denotes the period of task t_i, these dependency relations between task executions are repeated within the hyperperiod $T = lcm_{t_i \in \mathcal{T}}(T_i)$. An expanded valued graph $P_N(\mathcal{G})$ can then be built by duplicating each task $N_i = \frac{T}{T_i}$ times. We prove in this paper that setting any vector K with $K_i \in \mathbb{N} - \{0\}$ for any $t_i \in \mathcal{T}$, a partial expanded graph $P_K(\mathcal{G})$ can be built by duplicating each task K_i times. Each arc of this graph includes the modelling of the dependency relation between the corresponding executions of its adjacent task duplicates. This partial expanded graph is inspired from Bodin et al. [5] and de Groote [7] for Synchronous DataFlow Graphs [17], for which the initial inequality modelling dependency is slightly different.

Subsequently, we show that upper bounds on the latency between adjacent duplicates of $P_K(\mathcal{G})$ can be derived and considered as a valuation of the arcs. The longest paths of $P_K(\mathcal{G})$ then provide an upper bound on the latency. However, the computation of these paths has a time complexity proportional to $\sum_{e=(t_i,t_j)\in E} K_i \times K_j$. The main problem is then to find the value of K that minimises this function with an exact evaluation of the age latency.

We first prove that our study can be limited to vectors K such that, for any task t_i, K_i divides N_i. We then develop a greedy algorithm that converges to a vector $K^\star$ that provides the exact value of the age latency. This algorithm can be seen as an adaptation of the K-iter algorithm [6] for the determination of the maximum throughput of a Synchonous DataFlow Graph, which is up to now one of the best algorithms to solve this latter problem. Our algorithm was experimentally tested on randomly generated graphs with periods inspired from automotive real-life benchmarks [11, 16].

Our paper is organised as follows. Section 2 presents related work. The problem and our characterisation of the dependencies between tasks executions are presented in Section 3. Section 4 is devoted to the construction of the partial expanded graph $P_K(\mathcal{G})$ for any fixed vector K. It is shown in Section 5 that exploration can be limited to K vectors such that,

for any task $t_i \in \mathcal{T}$, K_i is a divisor of N_i. Section 6 presents our greedy algorithm for the computation of a vector $K^\star$ leading to the exact value of the age latency. In section 7, we experiment with this algorithm on the ROSACE case study. Section 8 presents experiments on randomly generated graphs. Section 9 is our conclusion.

2 Related work

The evaluation of the age latency of an event chain is a challenging question tackled by several authors. Feiertag et al. [8] first introduced the definition of dependency between tasks of an event chain and four metrics to evaluate the delay between a sensor and an actuator. Becker et al. [2] developed a general framework to evaluate the age latency of an event chain using feasible intervals. They built an expanded graph by evaluating the possible propagation of input data by the successive executions of tasks. They tested in [3] their approach against the evaluation of the latency of a fixed schedule or under the LET hypothesis. They concluded that if there is no information on the communications or on the schedule, a pessimistic value of the age latency will be obtained, which is very similar to the value obtained using the LET paradigm. However, the computation time grows exponentially with the number of tasks if an enumeration is needed, while it remains constant for the LET paradigm.

Under the LET assumption, the times of the communications between tasks are known before the executions of the tasks. This strong assumption allows to characterise the dependencies between tasks if their parameters are fixed. Martinez et al. [20] gave a formal characterisation of the dependencies between tasks in an event chain using time instants. They then derived the age latency by enumerating all the possible paths of the corresponding expanded graph. They also proved that the release times influence the age latency and they developed a heuristic to fix them in order to minimise it.

Many practical applications are composed of graphs with no particular assumption on their structure [16, 22]. None of these previous approaches can be easily extended to these graphs. Indeed, the number of paths between two vertices is potentially exponential. The complexity of a method that enumerates all the paths for evaluating their age latency will thus grow exponentially following the parameters of the graph. Anyway, mainly two frameworks referenced below are capable of tackling such applications.

Pagetti et al. [21] have developed a language to express the constraints and a multi-periodic synchronous model to represent the whole system for a general graph. The size of the description of the communications is then equivalent to the one of the expanded graph $P_N(\mathcal{G})$. Forget et al. [9] showed that this approach supports several metrics.

Khatib et al. [14] proved that constraints between the successive executions of two adjacent tasks can be modelled using a Synchronous DataFlow Graph [17]. Our equation is slightly different since for any arc $e = (t_i, t_j)$, they did not not consider the successive constraints between two adjacent tasks if $T_i > T_j$, dealing only with precedence constraints. They then computed the age latency using the expansion of the Synchronous DataFlow Graph which is equivalent to $P_N(\mathcal{G})$. They also proposed the computation of a polynomial upper bound on the age latency equivalent to the determination of the longest paths of $P_{\mathbb{1}^n}(\mathcal{G})$ with $n = |\mathcal{T}|$. Lastly, they showed that the difference between this bound and the age latency is on average between 10 and 15 percent. This result motivates the development of efficient methods to evaluate more precisely the age latency of a graph $\mathcal{G}$.

3 Modelling of the system

This section formally presents the problem tackled in this paper. Subsection 3.1 defines the periodic tasks model considered according to LET restrictions. Subsection 3.2 is dedicated to the definition of the dependency relation between the successive executions of two adjacent tasks. Subsection 3.3 formally defines the age latency of a graph. Subsection 3.4 is devoted to the definition of the problem and the presentation of a small pedagogical example.

3.1 Periodic tasks model considering LET communications

Let us consider a set $\mathcal{T} = \{t_1, \ldots, t_n\}$ of real-time periodic tasks following the model of Liu and Layland [19]. Each task $t_i \in \mathcal{T}$ is characterised by a quadruple (r_i, C_i, D_i, T_i) such that:

- r_i is the release date (the offset) of the first execution of t_i;
- C_i is the worst-case execution time of t_i;
- D_i is the relative deadline of t_i;
- T_i is the period of t_i.

For any value $n \in \mathbb{N} - \{0\}$, we denote by $\langle t_i, n\rangle$ the nth execution of t_i and by $s(t_i, n)$ its starting time. The execution of $\langle t_i, n\rangle$ must be scheduled in its time window, that is $r_i + (n-1) \times T_i \leq s(t_i, n)$ and $s(t_i, n) + C_i \leq D_i + (n-1) \times T_i$.

The LET communication model separates task executions from communications. In this model, data are read at the release dates of reading tasks and written at the deadlines of writing tasks. Moreover, reading tasks always get the last emitted data. The main advantage of this model is to define a deterministic communications system even if tasks are delayed inside their time windows.

In this paper, we only consider LET communications and we limit the characterization of tasks to their successive time windows. The execution time associated to the nth execution of t_i is then set to its release date, that is, $\mathcal{S}(t_i, n) = r_i + (n-1) \times T_i$. Similarly, the completion time is fixed to $\mathcal{S}(t_i, n) + D_i$. Each task t_i is then given by the triple (r_i, D_i, T_i).

3.2 LET dependencies

Communications are expressed by a directed graph $\mathcal{G} = (\mathcal{T}, E)$. Each arc $e = (t_i, t_j) \in E$ induces dependencies between executions of t_i and t_j, defined as follows:

▶ **Definition 1.** *Let us suppose that $e = (t_i, t_j) \in E$ and that ν_i and ν_j are two positive integers. There exists a dependency relation from $\langle t_i, \nu_i\rangle$ to $\langle t_j, \nu_j\rangle$ if $\langle t_j, \nu_j\rangle$ receives data from $\langle t_i, \nu_i\rangle$ that is if:*

1. *The execution time of $\langle t_j, \nu_j\rangle$ is greater than or equal to the completion time of $\langle t_i, \nu_i\rangle$ and*
2. *the execution time of $\langle t_i, \nu_i + 1\rangle$ is greater than the completion time of $\langle t_j, \nu_j\rangle$ (since the data element from $\langle t_i, \nu_i + 1\rangle$ is not available for $\langle t_j, \nu_j\rangle$).*

Figure 1 presents successive time windows of the first executions of two periodic tasks t_1 and t_2 with a LET communication $e = (t_1, t_2) \in E$. Since $T_1 > T_2$ a single write from t_1 can be read by several executions of t_2. As an example, there is a dependency from $\langle t_1, 2\rangle$ to $\langle t_2, 4\rangle$ since $\langle t_1, 2\rangle$ ends before the beginning of $\langle t_2, 4\rangle$ and the data written by $\langle t_1, 3\rangle$ is not available at the beginning of $\langle t_2, 4\rangle$.

The next theorem characterises the dependency relation between the executions of two communicating tasks using the parameters of the executions:

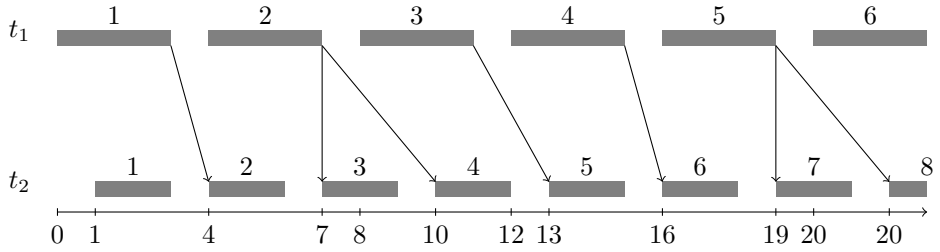


Figure 1 Time windows associated to two periodic tasks t_1 and t_2 with a LET dependency $e = (t_1, t_2)$. Parameters of tasks are respectively $(r_1, D_1, T_1) = (0, 3, 4)$ and $(r_2, D_2, T_2) = (1, 2, 3)$.

▶ **Theorem 2.** *Let $e = (t_i, t_j) \in E$, $gcd_T^e = \gcd(T_i, T_j)$ and the delay of e, $M^e = T_j + \left\lceil \frac{r_i - r_j + D_i}{gcd_T^e} \right\rceil \times gcd_T^e$. For any pair $(\nu_i, \nu_j) \in \mathbb{N} - \{0\} \times \mathbb{N} - \{0\}$, there exists a dependency from $\langle t_i, \nu_i \rangle$ to $\langle t_j, \nu_j \rangle$ iff $T_i \geq M^e + T_i\nu_i - T_j\nu_j > 0$.*

Proof. Following Definition 1, there exists a dependency from $\langle t_i, \nu_i \rangle$ to $\langle t_j, \nu_j \rangle$ if:

1. $\langle t_j, \nu_j \rangle$ begins after the completion of $\langle t_i, \nu_i \rangle$, thus $\mathcal{S}(t_i, \nu_i) + D_i \leq \mathcal{S}(t_j, \nu_j)$. Since $\mathcal{S}(t_i, \nu_i) = r_i + (\nu_i - 1) \times T_i$ and $\mathcal{S}(t_j, \nu_j) = r_j + (\nu_j - 1) \times T_j$, we get

   $$r_i + (\nu_i - 1) \times T_i + D_i \leq r_j + (\nu_j - 1) \times T_j,$$

   thus,

   $$T_i \geq T_j + (r_i - r_j + D_i) + T_i\nu_i - T_j\nu_j,$$

   and since in the inequality above only $r_i - r_j + D_i$ cannot be divided by gcd_T^e, we obtain that $T_i \geq M^e + T_i\nu_i - T_j\nu_j$.
2. The completion time of $\langle t_i, \nu_i + 1 \rangle$ is strictly greater than the execution time of $\langle t_j, \nu_j \rangle$, thus $\mathcal{S}(t_i, \nu_i + 1) + D_i > \mathcal{S}(t_j, \nu_j)$ and then

   $$r_i + \nu_i T_i + D_i > r_j + (\nu_j - 1) \times T_j,$$

   thus,

   $$T_j + (r_i - r_j + D_i) + T_i\nu_i - T_j\nu_j > 0.$$

   Since $M^e \geq T_j + (r_i - r_j + D_i)$, $M^e + T_i\nu_i - T_j\nu_j > 0$.

Merging the two inequalities gives the theorem. ◀

Let us consider, for example, the two tasks t_1 and t_2 with the LET communication $e = (t_1, t_2)$ presented in Figure 1. We get $\gcd_T^e = \gcd(3, 4) = 1$ and $M^e = 3 + (0 - 1 + 3) = 5$. The inequality of Theorem 2 is $4 \geq 5 + 4\nu_1 - 3\nu_2 \geq 0$. One can observe that the first executions of t_1 and t_2 with a dependency relation correspond to the pairs that verify this inequality. For $(\nu_1, \nu_2) = (1, 2)$, we get $5 + 4\nu_1 - 3\nu_2 = 5 + 4 - 6 = 3 \in \{1, \ldots, 4\}$. Similarly, for $(\nu_1, \nu_2) = (2, 3)$, we get $5 + 4\nu_1 - 3\nu_2 = 5 + 8 - 9 = 4 \in \{1, \ldots, 4\}$. Now, if we consider $(\nu_1, \nu_2) = (2, 5)$, $5 + 4\nu_1 - 3\nu_2 = 5 + 8 - 15 = -2 \notin \{1, \ldots, 4\}$ and there is no dependency from $\langle t_1, 2 \rangle$ to $\langle t_2, 5 \rangle$.

3.3 Age latency

Let us suppose that $e = (t_i, t_j) \in E$ and let $\mathcal{R}(e)$ be the set of pairs $(\nu_i, \nu_j) \in (\mathbb{N} - \{0\})^2$ such that e induces a dependency from $\langle t_i, \nu_i \rangle$ to $\langle t_j, \nu_j \rangle$. Then, for any pair $(\nu_i, \nu_j) \in \mathcal{R}(e)$, we define the latency of e between the executions $\langle t_i, \nu_i \rangle$ and $\langle t_j, \nu_j \rangle$ as

$$\mathcal{L}_{\nu_i,\nu_j}(e) = \mathcal{S}(t_j, \nu_j) - \mathcal{S}(t_i, \nu_i) = r_j - r_i + T_i - T_j - (T_i\nu_i - T_j\nu_j). \tag{1}$$

Now, for any path $p = t_1 t_2 \dots t_k$ of $\mathcal{G}$, we set $e_\ell = (t_\ell, t_{\ell+1})$ for the corresponding arcs with $\ell \in \{1, \dots, k-1\}$. We define $\mathcal{R}(p)$ as the set of k-tuples $(\nu_1, \dots, \nu_k) \in (\mathbb{N} - \{0\})^k$ such that $\forall \ell \in \{1, \dots, k-1\}$, $(\nu_\ell, \nu_{\ell+1}) \in \mathcal{R}(e_\ell)$. Then, for any k-tuple $(\nu_1, \dots, \nu_k) \in \mathcal{R}(p)$, we have

$$\mathcal{L}_{\nu_1,\dots,\nu_k}(p) = \sum_{\ell=1}^{k-1} \mathcal{L}_{\nu_\ell,\nu_{\ell+1}}(e_\ell) + D_k.$$

The age latency of a path p of $\mathcal{G}$ is then defined as the maximum time interval from a specific input value $\langle t_1, \nu_1 \rangle$ to the end of the output value $\langle t_1, \nu_1 \rangle$, thus

$$\mathcal{L}^\star(p) = \max\{\mathcal{L}_{\nu_1,\dots,\nu_k}(p), (\nu_1, \dots, \nu_k) \in \mathcal{R}(p)\}$$

and the maximum latency of a directed graph $\mathcal{G}$ corresponds to

$$\mathcal{L}^\star(\mathcal{G}) = \max\{\mathcal{L}^\star(p), p \text{ path of } \mathcal{G}\}.$$

Let us observe that, if the initial graph $\mathcal{G}$ contains cycles, its latency may not be bounded. Indeed, infinite paths p can be built in this case by looping in the cycles and the latency cannot be defined. So, we suppose in the following that $\mathcal{G}$ is acyclic. Moreover, since the latency between two executions is positive, $\mathcal{L}^\star(\mathcal{G})$ is reached for a path p such that t_1 has no predecessor and t_k no successor.

If $\mathcal{G}$ contains cycles, other definitions of the latency could be considered as "last-to-first" or "first-to-first", following Feiertag et al.'s definition [8]. The methodology and the algorithms presented in this paper can clearly be extended to tackle these cases and the existence of cycles does not complicate most of the reasoning.

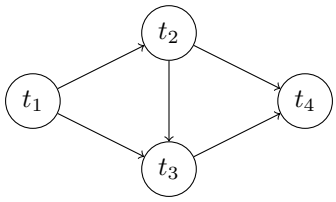
3.4 Problem definition and example

The problem tackled in this paper can be formalised as follows: let us consider a directed acyclic graph $\mathcal{G} = (\mathcal{T}, E)$, each arc modelling a LET communication. Each periodic task $t_i \in \mathcal{T}$ is associated to a triple (r_i, D_i, T_i). The problem is to compute the maximal age latency $\mathcal{L}^\star(\mathcal{G})$.

Figure 2 presents an instance of our problem comprising four periodic tasks and the associated directed acyclic graph $\mathcal{G}$. Dependency relations between the first executions of tasks t_1, t_2 and t_4 are shown in Figure 3, following the path $p = t_1 t_2 t_4$ of $\mathcal{G}$. The latency of the path from $\langle t_1, 1 \rangle$ to $\langle t_4, 1 \rangle$ is $\mathcal{L}_{1,2,1}(p) = \mathcal{S}(t_4, 1) - \mathcal{S}(t_1, 1) + 3 = 3 - 0 + 3 = 6$. In the same way, the latency of the path p from $\langle t_1, 3 \rangle$ to $\langle t_4, 2 \rangle$ is $\mathcal{L}_{3,5,2}(p) = \mathcal{S}(t_4, 2) - \mathcal{S}(t_1, 3) + 3 = 6 - 4 + 3 = 5$. We deduce that $\mathcal{L}^\star(p) = 6$.

4 Construction of a partial expanded graph

The aim of this section is to detail and prove the construction of a partial expanded graph $P_K(\mathcal{G})$ associated to a fixed vector $K \in (\mathbb{N} - \{0\})^n$. The main idea is to duplicate each task t_i, K_i times and to express the dependencies directly on duplicates.



t_i	t_1	t_2	t_3	t_4
r_i	0	1	2	3
D_i	1	0.5	4	3
T_i	2	1	6	3

Figure 2 An instance of four periodic tasks and the associated DAG $\mathcal{G}$.

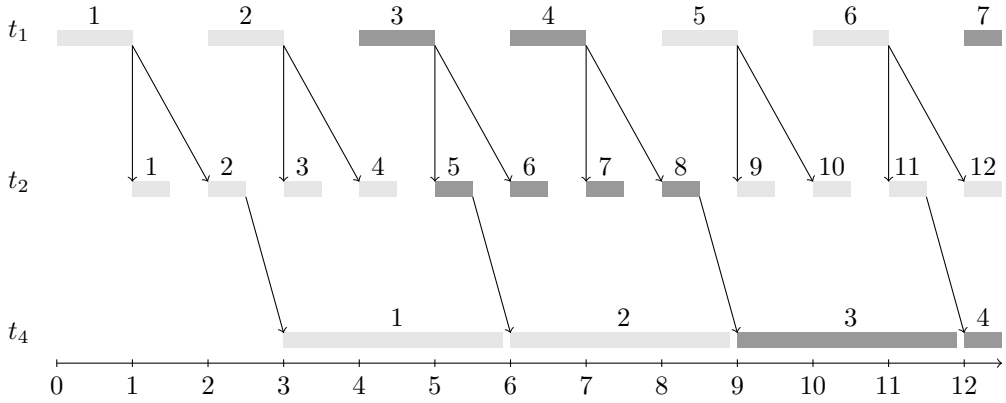


Figure 3 A path $p = t_1t_2t_4$ from the graph $\mathcal{G}$ shown in Figure 2. Time windows are colored following blocks of $K_1 = 2$ executions of t_1, $K_2 = 4$ executions of t_2 and $K_4 = 2$ executions of t_4.

Subsection 4.1 is devoted to the proof of Theorem 5 that characterises the dependency relations between the duplicates of two adjacent tasks. An upper bound on the latency between two duplicates corresponding to dependant executions is then evaluated in Subsection 4.2. Subsection 4.3 formally defines the partial expanded graph $P_K(\mathcal{G})$ associated with a vector K, while subsection 4.4 evaluates the complexity of its computation.

4.1 Characterisation of the dependencies between duplicates of the partial expanded graph

Let us suppose that for any task t_i, a number of duplicates $K_i \in \mathbb{N} - \{0\}$ is fixed. Then, for any $a_i \in \{1, \ldots, K_i\}$, the a_ith duplicate of t_i is simply associated to the executions $a_i + pK_i$ for $p \in \mathbb{N}$. For example, let us suppose that the task t_2 has a fixed number of duplicates $K_2 = 4$. For any value $a_2 \in \{1, 2, 3, 4\}$, we merge into a unique duplicate all the executions $\langle t_2, a_2 + pK_2\rangle$ for $p \in \mathbb{N}$. For $a_2 = 1$, it corresponds to executions $\langle t_2, 1\rangle, \langle t_2, 5\rangle, \langle t_2, 9\rangle \ldots \langle t_2, 1 + 4p\rangle$.

Now, suppose that $K_2 = 4$, $K_4 = 2$. We aim to characterize the dependencies from duplicates of t_2 to duplicates of t_4 due to the LET communication $e = (t_2, t_4)$. We observe in Figure 3 that there exists a dependency from $\langle t_2, 11\rangle$ to $\langle t_4, 4\rangle$. Moreover, $11 = 3 + 2 \times 4$ and $4 = 2 + 1 \times 2$. So, we set $a_2 = 3$, $a_4 = 2$ and we look to characterize dependencies from executions $\nu_2 = a_2 + p_2K_2 = 3 + 4p_2$ of t_2 to executions $\nu_4 = a_4 + p_4K_4 = 2 + 2p_4$ of t_4.

Following Theorem 2, the delay associated to e is $M^e = 3 + \left\lceil \frac{1-3+0.5}{1} \right\rceil = 2$. Moreover, there exists a dependency from $\langle t_2, \nu_2\rangle$ to $\langle t_4, \nu_4\rangle$ if and only if $T_2 \geq M^e + T_2\nu_2 - T_4\nu_4 > 0$.

Now, with these previous assumptions, $T_2\nu_2 - T_4\nu_4 = (3+4p_2) - 3(2+2p_4) = (4p_2 - 6p_4) - 3$. This difference is composed by a linear function of p_2 and p_4 and a constant term equal to 3.

These two terms are characterized in next lemma. Moreover, since $T_2 = 1$ we observe that, $M^e + T_2\nu_2 - T_4 = (4p_2 - 6p_4) - 1 = 1$, and thus $4p_2 - 6p_4 = 2$.

The conclusion is that there exists a dependency from $\langle t_2, 3 + 4p_2\rangle$ to $\langle t_4, 2 + 2p_4\rangle$ if and only if $4p_2 - 6p_4 = 2$. Theorem 5 generalizes this characterization to any LET communication between two communicating tasks.

▶ **Lemma 3.** *Consider $e = (t_i, t_j) \in E$ and let gcd_T^e (resp., gcd_K^e) be the greatest common divisor between T_i and T_j (resp., K_iT_i and K_jT_j). Let $\nu_i = a_i + p_iK_i$ and $\nu_j = a_j + p_jK_j$ with $(a_i, a_j) \in \{1, \ldots, K_i\} \times \{1, \ldots, K_j\}$ and $(p_i, p_j) \in \mathbb{N} \times \mathbb{N}$. Let us define the four values*

- $\alpha_e(a_i, a_j) = \dfrac{T_ia_i - T_ja_j}{gcd_T^e}$,
- $\pi_e(p_i, p_j) = \dfrac{T_ip_iK_i - T_jp_jK_j}{gcd_K^e}$,
- $\pi_e^{max}(a_i, a_j) = \left\lfloor \dfrac{-M^e + T_i - \alpha_e(a_i, a_j) \cdot gcd_T^e}{gcd_K^e} \right\rfloor$ *and*
- $\pi_e^{min}(a_i, a_j) = \left\lceil \dfrac{-M^e + gcd_T^e - \alpha_e(a_i, a_j) gcd_T^e}{gcd_K^e} \right\rceil$.

If e induces a dependency from $\langle t_i, \nu_i\rangle$ to $\langle t_j, \nu_j\rangle$, then

$$T_i\nu_i - T_j\nu_j = \pi_e(p_i, p_j) \cdot gcd_K^e + \alpha(a_i, a_j) \cdot gcd_T^e$$

with $\pi_e(p_i, p_j) \in \{\pi_e^{min}(a_i, a_j), \ldots, \pi_e^{max}(a_i, a_j)\}$.

Proof. By definition of ν_i and ν_j,

$$\begin{aligned} T_i\nu_i - T_j\nu_j &= T_i \times (a_i + K_ip_i) - T_j \times (a_j + K_jp_j) = (T_iK_ip_i - T_jK_jp_j) + (T_ia_i - T_ja_j) \\ &= \pi_e(p_i, p_j) \cdot gcd_K^e + \alpha_e(a_i, a_j) \cdot gcd_T^e. \end{aligned}$$

By Theorem 2, $T_i - M^e \geq T_i\,\nu_i - T_j\nu_j > -M^e$. Thus, since all the terms of this inequality are divisible by gcd_T^e, it is equivalent to $T_i - M^e \geq T_i\,\nu_i - T_j\nu_j \geq -M^e + gcd_T^e$ and we get

$$T_i - M^e \geq \pi_e(p_i, p_j) \cdot gcd_K^e + \alpha_e(a_i, a_j) \cdot gcd_T^e \geq -M^e + gcd_T^e.$$

From the right part of the inequality,

$$\pi_e(p_i, p_j) \geq \frac{-M^e + gcd_T^e - \alpha_e(a_i, a_j) \cdot gcd_T^e}{gcd_K^e}.$$

Since $\pi_e(p_i, p_j)$ is an integer, we can tighten the lower bound of $\pi_e(p_i, p_j)$ by

$$\pi_e(p_i, p_j) \geq \left\lceil \frac{-M^e + gcd_T^e - \alpha_e(a_i, a_j) \cdot gcd_T^e}{gcd_K^e} \right\rceil = \pi_e^{min}(a_i, a_j).$$

In the same way, the left part of the previous inequality is

$$\frac{T_i - M^e - \alpha_e(a_i, a_j) \cdot gcd_T^e}{gcd_K^e} \geq \pi_e(p_i, p_j).$$

Since $\pi_e(p_i, p_j)$ is an integer, we can tighten the upper bound on $\pi_e(p_i, p_j)$ by

$$\left\lfloor \frac{T_i - M^e - \alpha_e(a_i, a_j) \cdot gcd_T^e}{gcd_K^e} \right\rfloor \geq \pi_e(p_i, p_j)$$

So we get $\pi_e^{max}(a_i, a_j) \geq \pi_e(p_i, p_j)$ and the lemma is proved. ◀

Table 1 Values $\alpha_e(a_2, a_4)$, $\pi_e^{max}(a_2, a_4)$ and $\pi_e^{min}(a_2, a_4)$ for $a_2 \in \{1, 2, 3, 4\}$ and $a_4 \in \{1, 2\}$.

a_2 \ a_4	1	2
1	−2	−5
2	−1	−4
3	0	−3
4	1	−2

$\alpha_e(a_2, a_4)$

a_2 \ a_4	1	2
1	0	2
2	0	1
3	−1	1
4	−1	0

$\pi_e^{max}(a_2, a_4)$

a_2 \ a_4	1	2
1	1	2
2	0	2
3	0	1
4	−1	1

$\pi_e^{min}(a_2, a_4)$

Consider as an example, the arc $e = (t_2, t_4)$ of the example shown in Figure 2 with fixed values $K_2 = 4$ and $K_4 = 2$. We get $gcd_T^e = \gcd(1, 3) = 1$, $gcd_K^e = \gcd(4, 6) = 2$ and $M^e = 2$. The corresponding values of $\alpha_e(a_i, a_j)$, $\pi_e^{max}(a_i, a_j)$ and $\pi_e^{min}(a_i, a_j)$ are shown in Table 1.

For the pair $(a_2, a_4) = (3, 2)$, suppose that there exists a dependency from $\langle t_2, \nu_2 \rangle$ to $\langle t_4, \nu_4 \rangle$ with $\nu_2 = a_2 + p_2 K_2 = 3 + 4p_2$ and $\nu_4 = a_4 + p_4 K_4 = 2 + 2p_4$.

$$T_2\nu_2 - T_4\nu_4 = \nu_2 - 3\nu_4 = (3+4p_2) - 3(2+2p_4) = 2(2p_2 - 3p_4) - 3 = gcd_K^e \cdot \pi_e(p_2, p_4) - \alpha_e(3, 2).$$

As $\pi_e^{max}(3, 2) = \pi_e^{min}(3, 2) = 1$, the only possible value for $\pi_e(p_2, p_4)$ is 1, thus $\pi_e(p_2, p_4) = 2p_2 - 3p_4 = 1$.

Consider now the pair $(a_2, a_4) = (1, 1)$. Then, since $\pi_e^{max}(1, 1) < \pi_e^{min}(1, 1)$, such a decomposition of the difference $T_2\nu_2 - T_4\nu_4$ with $\nu_2 = 1 + p_2 K_2$ and $\nu_4 = 1 + p_4 K_4$ is not possible; a simple consequence of Lemma 3 is that there is no dependency relation between executions $\langle t_2, 1 + p_2 K_2 \rangle$ and $\langle t_4, 1 + p_4 K_4 \rangle$.

We observe in Figure 3 that there exist dependencies $\langle t_2, 2 \rangle \to \langle t_4, 1 \rangle$, $\langle t_2, 5 \rangle \to \langle t_4, 2 \rangle$, $\langle t_2, 8 \rangle \to \langle t_4, 3 \rangle$ and $\langle t_2, 11 \rangle \to \langle t_4, 4 \rangle$. They correspond respectively to the pairs $(a_2, a_4) = (2, 1)$, $(a_2, a_4) = (1, 2)$, $(a_2, a_4) = (4, 1)$ and $(a_2, a_4) = (3, 2)$. For all these pairs, one can check that $\pi_e^{max}(a_2, a_4) \geq \pi_e^{min}(a_2, a_4)$.

For the general case, a consequence of Lemma 3 is that there is no dependency between executions $\langle t_i, a_i + p_i K_i \rangle$ and $\langle t_j, a_j + p_j K_j \rangle$ if $\pi_e^{max}(a_i, a_j) < \pi_e^{min}(a_i, a_j)$. Thus, let us define

$$\mathbb{A}(e) = \left\{ (a_i, a_j) \in \{1, \ldots, K_i\} \times \{1, \ldots, K_j\} \mid \pi_e^{max}(a_i, a_j) \geq \pi_e^{min}(a_i, a_j) \right\}.$$

For our particular case, $\mathbb{A}(e) = \{(2, 1), (1, 2), (4, 1), (3, 2)\}$.

The next lemma is the converse of Lemma 3.

▶ **Lemma 4.** *Let $e = (t_i, t_j) \in E$ and $(a_i, a_j) \in \mathbb{A}(e)$. For any integer value $\pi \in \{\pi_e^{min}(a_i, a_j), \ldots, \pi_e^{max}(a_i, a_j)\}$, there exists an infinite number of pairs $(p_i, p_j) \in \mathbb{N}^2$ such that $\pi = \pi_e(p_i, p_j)$. Moreover, setting $\nu_i = a_i + p_i K_i$ and $\nu_j = a_j + p_j K_j$, e induces a dependency from $\langle t_i, \nu_i \rangle$ to $\langle t_j, \nu_j \rangle$.*

Proof. By Bezout's identity, there exists $(x, y) \in \mathbb{Z}^2$ such that $xK_iT_i + yK_jT_j = gcd_K^e$ and thus $\pi x K_i T_i + \pi y K_j T_j = \pi \cdot gcd_K^e$.

For $z \in \mathbb{N}$, let us define $p_i = \pi x + z K_j T_j$ and $p_j = -\pi y + z K_i T_i$. Let us also consider values ν_i and ν_j such that $\nu_i = a_i + K_i p_i$ and $\nu_j = a_j + K_j p_j$. For z sufficiently large ($z \geq z_0$), $p_i \geq 1$ and $p_j \geq 1$, and thus ν_i and ν_j are both greater than 1. Then,

$$\begin{aligned} T_i p_i K_i - T_j p_j K_j &= K_i T_i (\pi x + z K_j T_j) - K_j T_j (-\pi y + z K_i T_i) \\ &= x\pi K_i T_i + y\pi K_j T_j = \pi \cdot gcd_K^e, \end{aligned}$$

thus $\pi = \pi_e(p_i, p_j)$. Now,

$$T_i\nu_i - T_j\nu_j = a_iT_i - a_jT_j + K_iT_ip_i - K_jZ_jp_j = a_iT_i - a_jT_j + \pi \cdot gcd_K^e$$

and thus, by definition of α_e, $T_i\nu_i - T_j\nu_j = \alpha_e(a_i, a_j) \cdot gcd_T^e + \pi \cdot gcd_K^e$. Recall now that $\pi \in \{\pi_e^{min}(a_i, a_j), \ldots, \pi_e^{max}(a_i, a_j)\}$, thus

$$T_i\nu_i - T_j\nu_j \leq \alpha_e(a_i, a_j) \cdot gcd_T^e + \pi_e^{max}(a_i, a_j) \cdot gcd_K^e,$$

and, since $\pi_e^{max}(a_i, a_j) \cdot gcd_K^e \leq -M^e + T_i - \alpha_e(a_i, a_j) \cdot gcd_T^e$,

$$T_i\nu_i - T_j\nu_j \leq -M^e + T_i. \tag{2}$$

Similarly, since $\pi_e^{min}(a_i, a_j) \cdot gcd_K^e \geq -M^e + gcd_T^e - \alpha_e(a_i, a_j) \cdot gcd_T^e$,

$$\begin{aligned} T_i\nu_i - T_j\nu_j &\geq \pi_e^{min}(a_i, a_j)gcd_K^e + \alpha_e(a_i, a_j)gcd_T^e \\ &\geq -M^e + gcd_T^e > -M^e. \end{aligned} \tag{3}$$

From equations (2) and (3), we have $T_i \geq M^e + T_i\nu_i - T_j\nu_j > 0$ and by Theorem 2 there is a dependency from $\langle t_i, \nu_i\rangle$ to $\langle t_j, \nu_j\rangle$. The lemma is proved. ◀

From Lemmas 3 and 4, we deduce the following main theorem:

▶ **Theorem 5.** *Let t_i and t_j be two tasks such that t_i (resp.t_j) is duplicated K_i (resp.K_j) times. Let $e = (t_i, t_j) \in E$ and $(a_i, a_j) \in \{1, \ldots, K_i\} \times \{1, \ldots, K_j\}$. There exists a dependency relation from $\langle t_i, a_i + p_iK_i\rangle$ to $\langle t_j, a_j + p_jK_j\rangle$ for $(p_i, p_j) \in \mathbb{N}^2$ iff $\pi_e^{min}(a_i, a_j) \leq \pi_e(p_i, p_j) \leq \pi_e^{max}(a_i, a_j)$.*

4.2 Upper bound on the latency

For any arc $e = (t_i, t_j) \in E$ and any pair $(a_i, a_j) \in \mathbb{A}(e)$, Theorem 5 gives the existence of a dependency from some executions $\langle t_i, \nu_i\rangle$ to $\langle t_j, \nu_j\rangle$ with $\nu_i = a_i + p_iK_i$ and $\nu_j = a_j + p_jK_i$. In order to evaluate the age latency of the whole graph $\mathcal{G}$, the next theorem evaluates the maximum latency associated to these executions of t_i and t_j.

▶ **Theorem 6** (Upper bound on the latency between two tasks)**.** *Let t_i and t_j be two tasks such that t_i (resp.t_j) is duplicated K_i (resp.K_j) times. Let also $e = (t_i, t_j) \in E$ and $(a_i, a_j) \in \mathbb{A}(e)$. Then*

$$\mathcal{L}^{max}_{(a_i,a_j)}(e) = r_j - r_i + T_i - T_j - (\pi_e^{min}(a_i, a_j) \cdot gcd_K^e + \alpha_e(a_i, a_j) \cdot gcd_T^e)$$

is the maximal value of the latency $\mathcal{L}_{\nu_i,\nu_j}(e)$ for $(\nu_i, \nu_j) \in \mathcal{R}(e)$ with $\nu_i = a_i \bmod K_i$ and $\nu_j = a_j \bmod K_j$.

Proof. By Equation (1), the latency between executions $\langle t_i, \nu_i\rangle$ and $\langle t_j, \nu_j\rangle$ for $(\nu_i, \nu_j) \in \mathcal{R}(e)$ is $\mathcal{L}_{\nu_i,\nu_j}(e) = r_j - r_i + T_i - T_j - (T_i\nu_i - T_j\nu_j)$. Assuming that $\nu_i = a_i + p_iK_i$ and $\nu_j = a_j + p_jK_j$ with $(p_i, p_j) \in \mathbb{N}^2$ we have by Lemma 3 that

$$\mathcal{L}_{\nu_i,\nu_j}(e) = r_j - r_i + T_i - T_j - (\pi_e(p_i, p_j) \cdot gcd_K^b + \alpha_b(a_i, a_j) \cdot gcd_T^b) \tag{4}$$

By Theorem 5, $\pi_e(p_i, p_j) \in \{\pi_e^{min}(a_i, a_j), \ldots, \pi_e^{max}(a_i, a_j)\}$. We conclude that $\mathcal{L}_{\nu_i,\nu_j}(e)$ is maximum for $\pi_e(p_i, p_j) = \pi_e^{min}(a_i, a_j)$ and the theorem is proved. ◀

4.3 Definition of the partial expanded graph

We suppose that the vector $K \in (\mathbb{N} - \{0\})^n$ is fixed. The associated expanded graph $P_K(\mathcal{G}) = (V, B, \mathcal{L}^{max})$ is a valued directed acyclic graph defined as follows:

1. Each task t_i is duplicated K_i times. For any value $a \in \{1, \ldots, K_i\}$, the ath duplicate of t_i is denoted by t_i^a and is associated to the executions $\langle t_i, a + pK_i \rangle$ for $p \in \mathbb{N}$.
2. For any arc $e = (t_i, t_j) \in E$, we build an arc (t_i^a, t_j^b) for every pair $(a, b) \in \{1, \ldots, K_i\} \times \{1, \ldots, K_j\}$ if $\pi_e^{max}(a, b) \geq \pi_e^{min}(a, b)$.
3. For every arc $\beta = (t_i^a, t_j^b) \in B$, $\mathcal{L}^{max}(\beta) = \mathcal{L}^{max}_{(a,b)}(e)$ following Theorem 6.
4. Lastly, two additional fictitious tasks s and f are considered with the arcs defined as:
   - For any duplicate t_i^a with no predecessors, add the arc $\beta = (s, t_i^a)$ with $\mathcal{L}^{max}(\beta) = 0$;
   - For any duplicate t_i^a with no successors, add the arc $\beta = (t_i^a, f)$ with $\mathcal{L}^{max}(\beta) = D_i$.

Let us denote by $LP^{max}(P_K(\mathcal{G}))$ the length of the longest path of the associated partial expanded graph $P_K(\mathcal{G})$ considering the arcs values $\mathcal{L}^{max}(\beta)$, $\beta \in B$. By Theorem 6, values on the arcs of $P_K(\mathcal{G})$ are upper bounds of the age latency, thus $LP^{max}(P_K(\mathcal{G}))$ is an upper bound of the maximum latency of $\mathcal{G}$.

Figure 4 presents the expanded graph $P_K(\mathcal{G})$ associated with the vector $K = (2, 4, 1, 2)$ for the instance shown in Figure 2. A longest path is given by $p = s, t_1^1, t_3^1, t_4^1, f$ with a corresponding length equal to 12, i.e., $LP^{max}(P_K(\mathcal{G})) = 12$. We conclude that $\mathcal{L}^{\star}(\mathcal{G}) \leq LP^{max}(P_K(\mathcal{G})) = 12$.

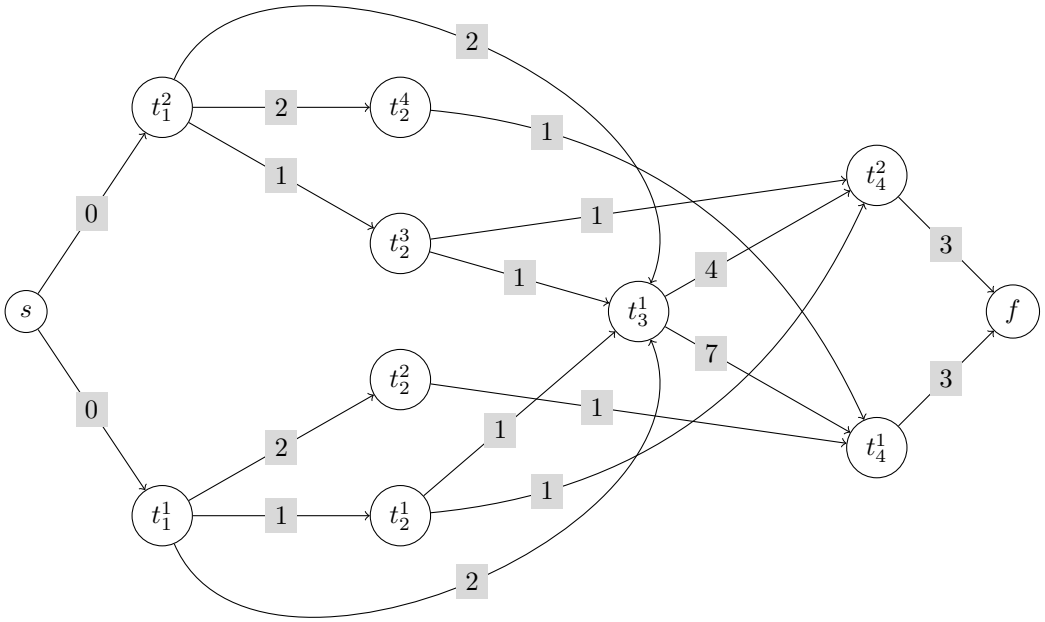


Figure 4 Expanded graph $P_K(\mathcal{G}) = (V, B, \mathcal{L}^{max})$ for the instance shown in Figure 2 associated with the vector $K = (2, 4, 1, 2)$. Arcs $\beta \in B$ are weighted by $\mathcal{L}^{max}(\beta)$ in gray.

4.4 Complexity of the computation of $P_K(\mathcal{G})$ and its longest paths

$P_K(\mathcal{G})$ is a graph without cycles. Thus, the computation of the longest paths can be done in time complexity $\Theta(|V| + |B|)$ by simply sorting the vertices following a topological order used in the next step to explore the vertices.

Note that the total number of vertices of $P_K(\mathcal{G})$ is $|V| = \sum_{i=1}^{n} K_i + 2$, while the number of arcs $|B|$ is bounded by $\mathcal{O}(\sum_{e=(t_i,t_j)\in E} K_i \times K_j)$. These two values may be huge for large values of K. The main problem consists then in the determination of the vector K of small values such that the bound $LP^{max}(P_K(\mathcal{G}))$ is as close as possible to the age latency $\mathcal{L}^{\star}(\mathcal{G})$.

5 Dominant set for the expansion vector K

This section is devoted to the study of dominance properties on K w.r.t the age latency to reduce the set of vectors K. In Subsection 5.1 we prove that the value of the longest paths of the expanded graph $P_N(\mathcal{G})$ associated with the hyperperiod N of $\mathcal{G}$ is the age latency $\mathcal{L}^{\star}(\mathcal{G})$. We prove in Subsection 5.2 that we can reduce our study to the set of the partial expansions $P_K(\mathcal{G})$ such that each component K_i divises N_i and we provide a partial order relation between these vectors that will be exploited in the following section for the computation of the age latency of $\mathcal{G}$.

5.1 Maximal value of the age latency for $K = N$

Consider $T = lcm_{t_i\in\mathcal{T}}(T_i)$ and the repetition vector $N \in \mathbb{N}^{*n}$ defined as $N_i = \dfrac{T}{T_i}$ for any task $t_i \in \mathcal{T}$. For our example shown in Figure 2, we get $T = lcm(2, 1, 6, 3) = 6$ and thus $N = (3, 6, 1, 2)$. Lemma 7 is a simple technical lemma.

▶ **Lemma 7.** *Let $P_N(\mathcal{G}) = (V, B, \mathcal{L}^{max})$ be the expanded graph with $K = N$, $e = (t_i, t_j)$ be an arc of $\mathcal{G}$. For any arc $\beta = (t_i^{a_i}, t_j^{a_j}) \in B$ associated with e and any pair $(q_i, q_j) \in \mathbb{N}^2$, $\pi_e(q_i, q_j) = q_i - q_j$.*

Proof. By definition of π_e, $\pi_e(q_i, q_j) = \dfrac{T_i q_i K_i - T_j q_j K_j}{gcd_K^e}$. As $T_i K_i = T_j K_j = T = gcd_K^e$, we have $\pi_e(q_i, q_j) = q_i - q_j$ and the lemma is proved. ◀

We prove formally in the following that the value of the longest path of the expanded graph $P_N(\mathcal{G})$ is the age latency of $\mathcal{G}$, i.e., $\mathcal{L}^{\star}(\mathcal{G})$:

▶ **Theorem 8.** *For any acyclic directed graph $\mathcal{G}$, $LP^{max}(P_N(\mathcal{G})) = \mathcal{L}^{\star}(\mathcal{G})$.*

Proof. By Theorem 6 and the definition of the partial expanded graphs, $LP^{max}(P_N(\mathcal{G})) \geq \mathcal{L}^{\star}(\mathcal{G})$. We prove that $LP^{max}(P_N(\mathcal{G})) \leq \mathcal{L}^{\star}(\mathcal{G})$.

Consider a path $p_N = t_1^{a_1}, t_2^{a_2} \dots t_k^{a_k}$ of $P_N(\mathcal{G})$ and the corresponding path $p = t_1, t_2 \dots t_k$ of $\mathcal{G}$. We also set $e_\ell = (t_\ell, t_{\ell+1})$ for $\ell \in \{1, \dots, k-1\}$. By Lemma 7, we have for any vector $(q_1, \dots, q_k) \in \mathbb{N}^k$ and $\ell \in \{1, \dots, k-1\}$, $\pi_{e_\ell}(q_\ell, q_{\ell+1}) = q_\ell - q_{\ell+1}$. Let us consider the sequence of integers $\tilde{q}_1, \dots, \tilde{q}_k$ defined as follows:

- $\tilde{q}_{\ell+1} = \tilde{q}_\ell + \pi_{e_\ell}^{max}(a_\ell, a_{\ell+1})$
- $\tilde{q}_1$ is fixed sufficiently large such that, $\forall \ell \in \{1, \dots, k\}$, $\tilde{q}_\ell \geq 0$.

This sequence satisfies $\forall \ell \in \{1, \dots, k-1\}$, $\pi_{e_\ell}(\tilde{q}_\ell, \tilde{q}_{\ell+1}) = \pi_{e_\ell}^{max}(a_\ell, a_{\ell+1})$, thus by Theorem 5, there is a dependency relation from $\langle t_\ell, a_\ell + \tilde{q}_\ell K_\ell\rangle$ to $\langle t_{\ell+1}, a_{\ell+1} + \tilde{q}_{\ell+1} K_{\ell+1}\rangle$. Moreover, by the definition of the sequence of arcs β_ℓ, $\mathcal{L}^{max}(\beta_\ell) = \mathcal{L}_{\tilde{q}_\ell, \tilde{q}_{\ell+1}}(e_\ell)$ and then $\mathcal{L}_{\tilde{q}_1,\dots,\tilde{q}_k}(p) = LP^{max}(p_N)$. If p_N is the longest path $P_N(\mathcal{G})$, $LP^{max}(P_N(\mathcal{G})) = LP^{max}(p_N) = \mathcal{L}_{\tilde{q}_1,\dots,\tilde{q}_k}(p) \leq \mathcal{L}^{\star}(\mathcal{G})$, which proves the theorem. ◀

5.2 Order relation between the divisors of the repetition vector N

The next theorem introduces an order relation between vectors $K \in (\mathbb{N} - \{0\})^n$.

▶ **Theorem 9.** *For any acyclic directed graph $\mathcal{G}$, suppose that K and K' are two different vectors such that $\forall t_i \in \mathcal{T}$, K'_i is a divisor of K_i, then $LP^{max}(P_{K'}(\mathcal{G})) \geq LP^{max}(P_K(\mathcal{G}))$.*

Proof. Let us consider the arc $e = (t_i, t_j)$ of $\mathcal{G}$. By the hypothesis, there exists $(x_i, x_j) \in (\mathbb{N} - \{0\})^2$, such that $K_i = x_i K'_i$ and $K_j = x_j K'_j$. Let $\beta = (t_i^{a_i}, t_j^{a_j})$ be an arc of $P_K(\mathcal{G})$ with $(a_i, a_j) \in \{1, \ldots, K_i\} \times \{1, \ldots, K_j\}$. Then, following Theorem 6 and the definition of the partial expanded graph, there exists $(\nu_i, \nu_j) \in (\mathbb{N} - \{0\})^2$ such that $\nu_i = a_i + p_i K_i$, $\nu_j = a_j + p_j K_j$ and $\mathcal{L}_{\nu_i,\nu_j}(t_i, t_j) = \mathcal{L}^{max}(\beta)$.

Let us consider now integer values $a'_i \in \{1, 2, \ldots, K'_i\}$, $a'_j \in \{1, 2, \ldots, K'_j\}$, y_i and y_j such that $a_i = a'_i + y_i K'_i$ and $a_j = a'_j + y_j K'_j$. Thus, $\nu_i = a'_i + (y_i + x_i p_i) K'_i$ and $\nu_j = a'_j + (y_j + x_j p_j) K'_j$. Since there is a dependency relation between $\langle t_i, \nu_i \rangle$ and $\langle t_j, \nu_j \rangle$, $\beta' = (t_i^{a'_i}, t_j^{a'_j})$ belongs to $P_{K'}(G)$ and $\mathcal{L}_{\nu_i,\nu_j}(t_i, t_j) \leq \mathcal{L}^{max}(\beta')$, thus we get $\mathcal{L}^{max}(\beta) \leq \mathcal{L}^{max}(\beta')$.

For any path $p = t_1^{a_1}, t_2^{a_2}, \ldots t_q^{a_q}$ in $P_K(\mathcal{G})$, there is a corresponding path $p' = t_1^{a'_1}, t_2^{a'_2}, \ldots t_q^{a'_q}$ in $P_{K'}(G)$ that includes all executions represented by path p. Therefore, $LP^{max}(P_{K'}(\mathcal{G})) \geq LP^{max}(P_K(\mathcal{G}))$. ◀

For any pair of vectors $(K, K') \in (\mathbb{N} - \{0\})^n \times (\mathbb{N} - \{0\})^n$, we set $K' \preceq K$ if, for any $t_i \in \mathcal{T}$, K'_i divides K_i. By Theorem 8, the exact value of the latency is reached for $K = N$. The consequence of this last theorem is that we can limit our study to the set $\mathcal{K}$ of vectors $K \preceq N$. Let us consider the graph $H = (\mathcal{K}, \preceq)$. The evaluation of the age latency is improved following paths from $K = \mathbb{1}^n$ to $K = N$. A vector $K \in \mathcal{K}$ is said to be optimum if $LP^{max}(P_K(\mathcal{G})) = \mathcal{L}^{\star}(\mathcal{G})$.

Figure 5 shows the graph H associated with the example from Figure 2. We observe that the exact value $\mathcal{L}^{\star}(\mathcal{G})$ of the age latency can be reached for vectors K smaller than N, i.e., there are several optimum vectors. The next section presents an algorithm to compute an optimum vector.

6 Determination of an optimum vector $K^{\star}$

The problem considered in this section is to compute an optimum vector $K^{\star}$, i.e., such that $LP^{max}(P_{K^{\star}}(\mathcal{G})) = \mathcal{L}^{\star}(\mathcal{G})$. Our algorithm computes iteratively a vector $K \in \mathcal{K}$ until the optimality test expressed by the next lemma is true.

▶ **Lemma 10** (Optimality test). *Consider a vector $K \in \mathcal{K}$, a longest path p_K of $P_K(\mathcal{G})$ and its corresponding path p of $\mathcal{G}$. If, for every task $t_i \in p$, K_i is a multiple of $N_i(p) = \dfrac{lcm_{t_j \in p}\{T_j\}}{T_i}$, then $LP^{max}(p_K) = \mathcal{L}^{\star}(\mathcal{G})$.*

Proof. Consider a vector K and the path p of $\mathcal{G}$ following the assumptions of the theorem. By definition of p_K, $LP^{max}(P_K(\mathcal{G})) = LP^{max}(p_K)$. We first prove that $\mathcal{L}^{\star}(p) = LP^{max}(p_K)$.

- Since p is a path of $\mathcal{G}$, $\mathcal{L}^{\star}(\mathcal{G}) \geq \mathcal{L}^{\star}(p)$. Now, by Theorem 6, $LP^{max}(P_K(\mathcal{G})) \geq \mathcal{L}^{\star}(\mathcal{G})$ and by definition of p_K, $LP^{max}(p_K) = LP^{max}(P_K(\mathcal{G}))$, thus $\mathcal{L}^{\star}(p) \leq LP^{max}(p_K)$.
- Now, since for any task t_i of p, $N_i(p)$ is a divisor of K_i, we have by Theorem 9 that $LP^{max}(P_{N(p)}(p)) \geq LP^{max}(p_K)$. Moreover, by Theorem 8, $LP^{max}(P_{N(p)}(p)) = \mathcal{L}^{\star}(p)$, thus $\mathcal{L}^{\star}(p) \geq LP^{max}(p_K)$.

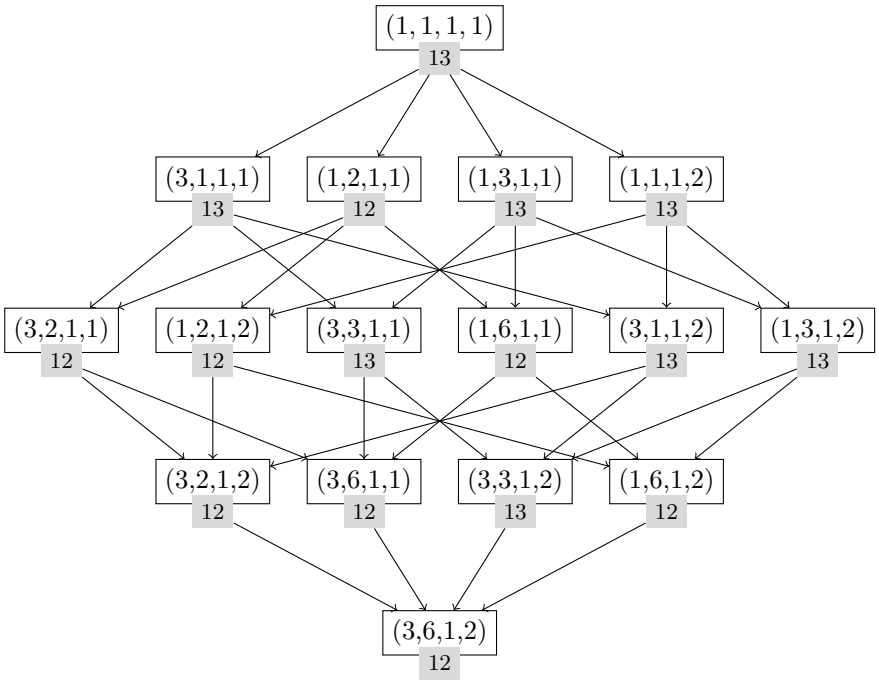


Figure 5 Graph $H = (\mathcal{K}, \preceq)$ associated with the example shown in Figure 2. Values $LP^{max}(P_K(\mathcal{G}))$ are given in gray for each vertex $K \in \mathcal{K}$.

So, we proved that $\mathcal{L}^\star(p) = LP^{max}(p_K) = LP^{max}(P_K(\mathcal{G}))$. Now, $\mathcal{L}^\star(\mathcal{G}) \geq \mathcal{L}^\star(p) = LP^{max}(p_K)$. Since $K \preceq N$, $\mathcal{L}^\star(\mathcal{G}) \leq LP^{max}(P_K(\mathcal{G})) = LP^{max}(p_K)$ by Theorem 9, and thus $LP^{max}(P_K(\mathcal{G})) = \mathcal{L}^\star(\mathcal{G}) = LP^{max}(p_K)$, which completes the proof. ◀

Algorithm 1 is inspired from the K-iter algorithm [6] which computes an expansion vector K for the determination of the optimum throughput of a Synchronous DataFlow Graph. For the initialisation phase, $K = \mathbb{1}^n$. K is simply increased at each step for tasks from the longest path of $P_K(\mathcal{G})$ until the maximality test is met.

Algorithm 1 Compute an optimum vector $K^\star$ and the age latency $\mathcal{L}(\mathcal{G})$.

Require: A DAG $\mathcal{G} = (\mathcal{T}, E)$, (r_i, D_i, T_i) for every $t_i \in \mathcal{T}$
Ensure: An optimum vector $K^\star$ and the age latency $\mathcal{L}^\star(\mathcal{G})$
  Set $K = \mathbb{1}^n$
  repeat
    Compute $P_K(\mathcal{G})$ and a longest path p_K of $P_K(\mathcal{G})$
    Set $p = s, t_1 \dots t_k, f$ to the corresponding path of $\mathcal{G}$
    Set $T(p) \leftarrow lcm(T_1, \dots, T_k)$ and $\forall i \in \{1, \dots, k\}$, $N_i(p) \leftarrow \frac{T(p)}{T_i}$
    OptPathFound$\leftarrow \forall t_i \in p, N_i(p) | K_i$
    if not OptPathFound **then**
      $\forall i \in \{1, \dots, k\}$, $K_i \leftarrow lcm(K_i, N_i(p))$
    end if
  until OptPathFound

Theorem 11 shows the convergence of the algorithm.

▶ **Theorem 11.** *For any directed acyclic graph $\mathcal{G}$, Algorithm 1 converges to a vector $K^\star \in \mathcal{K}$ such that $LP^{max}(P_{K^\star}(\mathcal{G})) = \mathcal{L}^\star(\mathcal{G})$.*

Proof. For any $q > 0$, we denote by $K(q)$ the vector K at the end of the qth iteration: $q = 0$ corresponds to the initialisation phase. We show that for any integer $q \geq 0$, $K(q) \in \mathcal{K}$ and $K(q) \preceq K(q+1)$ with $K(q) \neq K(q+1)$.

- At the initialisation step, $K(0) = \mathbb{1}^n \in \mathcal{K}$.
- Now, suppose that at step q, the optimality test is not true and that $K(q) \in \mathcal{K}$. Consider a task $t_i \in \mathcal{T}$. If t_i does not belong to p, $K_i(q+1) = K_i(q)$. Otherwise, $K_i(q+1) = lcm(K_i(q), N_i(p))$ where $K_i(q)$ and $N_i(p)$ are both divisors of N_i. Thus, $K_i(q+1)$ is also a divisor of N_i, and we get that $K(q+1) \in \mathcal{K}$ with $K(q) \preceq K(q+1)$.
- Lastly, we prove by contradiction that $K(q) \neq K(q+1)$. Indeed, suppose that $K_i(q) = K_i(q+1)$ for any task $t_i \in \mathcal{T}$, then since $K_i(q+1) = lcm(K_i(q), N_i(p))$, we deduce that $N_i(p)$ is a divisor of $K_i(q)$. Thus, the optimality test is true, which is a contradiction.

We conclude that vectors $K(q)$ are strictly increasing while the optimality test is false. By Lemma 10, the vector $K(q)$ is optimum when the optimality test is true. Lastly, the optimality test is true for the repetition vector N; this insures the convergence of the algorithm. ◀

The number of iterations of Algorithm 1 is not bounded and can be theoretically proportional to the maximum length of a path of the graph $H = (\mathcal{K}, E_\preceq)$.

Let us consider the first step of Algorithm 1 for the example of Figure 2. At initialisation, $K = \mathbb{1}^4$. The corresponding partial expanded graph $P_K(\mathcal{G})$ is shown by Figure 6. Its longest path of $P_K(\mathcal{G})$ is $p_K = s, t_1^1, t_2^1, t_3^1, t_4^1, f$ valued by $LP^{max}(p_K) = 13$. The optimality test fails, and we get $N(p) = (3, 6, 1, 2)$ which is the repetition vector and thus $K^\star = K(1) = N$.

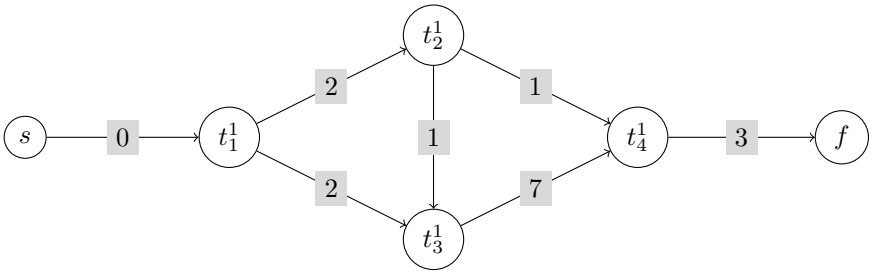


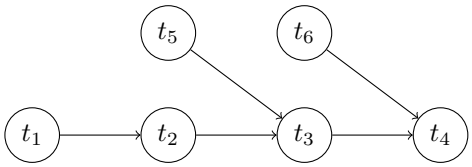
Figure 6 The partial expanded graph for the instance shown in Figure 2 and a unit vector $K = (1, 1, 1, 1)$. Arcs are weighted by $\mathcal{L}^{max}$ in gray.

7 ROSACE Case Study

ROSACE is the acronym for Research Open-Source Avionics and Control Engineering. This case study was developed by Pagetti et al. [22] to illustrate the implementation of a real-time system on a many-core architecture. Figure 7 presents an instance of the problem extracted from [9]. We arbitrarily set $r_i = 0$ and $D_i = T_i$ for any task $t_i \in \mathcal{T}$.

Figure 8 presents the partial expansion of the instance of Figure 7 for the unit expansion vector $K = \mathbb{1}^6$. A path of maximum length is $p_K = s, t_1^1, t_2^1, t_3^1, t_4^1, f$ with $LP^{max}(P_K(\mathcal{G})) = LP^{max}(p_K) = 260$ms.

At the first iteration of Algorithm 1, $p = s, t_1, t_2, t_3, t_4, f$ is expanded. We set $T(p) = \text{lcm}(60, 40, 30) = 120$, $N_1(p) = N_2(p) = 2$, $N_3(p) = 3$ and $N_4(p) = 4$. The next iteration, we set $K = (2, 2, 3, 4, 1, 1)$.



t_i	t_1	t_2	t_3	t_4	t_5	t_6
r_i	0	0	0	0	0	0
D_i	60	60	40	30	30	30
T_i	60	60	40	30	30	30

Figure 7 An instance of 6 periodic tasks and the associated DAG $\mathcal{G}$ extracted from the ROSACE case study [9].

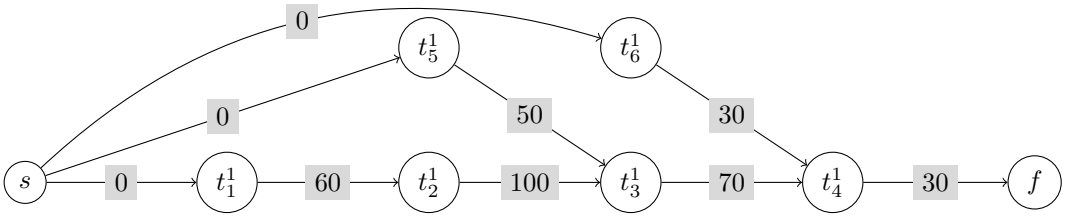


Figure 8 The partial expanded graph $P_K(\mathcal{G})$ for the instance shown in Figure 7 and a unit vector $K = \mathbb{1}^6$. Each arc β is weighted by $\mathcal{L}^{max}(\beta)$, shown in gray.

The partial expanded graph $P_K(\mathcal{G})$ built at the second iteration is shown in Figure 9. $p_K = s, t_1^1, t_2^2, t_3^2, t_4^4, f$ is a longest path of $P_K(\mathcal{G})$ with $LP^{max}(p_K) = LP^{max}(P_K(\mathcal{G})) =$ 240ms Moreover, the associated path $p = s, t_1, t_2, t_3, t_4, f$ verifies $T(p) = lcm(30, 40, 60)$, $N_1(p) = N_2(p) = 2$, $N_3(p) = 3$ and $N_4(p) = 4$. The optimality test is true and we get $K^\star = (2, 2, 3, 4, 1, 1)$. The maximum age latency of $\mathcal{G}$ is thus $\mathcal{L}^\star(\mathcal{G}) = LP^{max}(p_{K^\star}) = 240$ms.

We observe in this example that all the tasks of the critical path (i.e., the paths p of $\mathcal{G}$ such that $\mathcal{L}^\star(p) = \mathcal{L}^\star(\mathcal{G})$) were expanded at least following $N(p)$. Moreover, tasks from other paths are not necessarily duplicated: for example, $K_5^\star = K_6^\star = 1$ with $N_5 = N_6 = 4$. Thus, we can identify that paths s, t_5, t_3, t_4, f and s, t_6, t_4, f are not critical and tasks can be delayed without influence on the age latency.

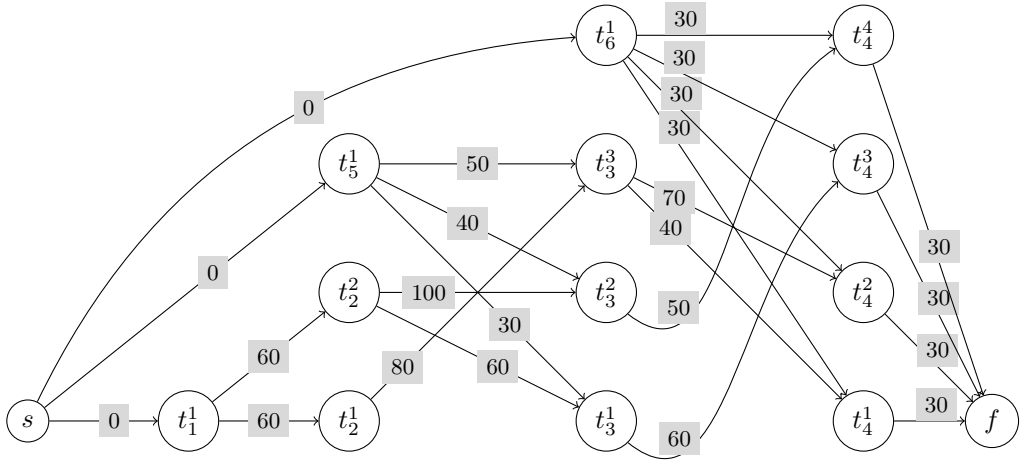


Figure 9 The partial expanded graph $P_K(\mathcal{G})$ for the instance shown in Figure 7 and the vector $K = (2, 2, 3, 4, 1, 1)$. Each arc β is weighted by $\mathcal{L}^{max}(\beta)$.

8 Experimental results

Our experiments aim at testing the performance of Algorithm 1. Following the experiments of Khatib et al. [14], the bound obtained from the longest paths of $P_{\mathbb{1}^n}(\mathcal{G})$ can be computed quickly, but its performance is on average between 10 and 15 percent from the maximal value $\mathcal{L}^\star(\mathcal{G})$. Moreover, their method does not precisely identify the real critical paths w.r.t the age latency of the initial graph.

Our Benchmarks were randomly generated: they are detailed is Subsection 8.1. The analysis of the computation time of our algorithm is presented in Subsection 8.2. Subsection 8.3 deals with the analysis of the critical vectors $K^\star$ obtained by our algorithm.

All our experiments were performed on an Intel(R) Core(TM) i5-8400 CPU (6 cores at 2.80GHz) and 15 GB of RAM. Our codes are written in Python. Functions dealing with graphs were implemented using the Python package NetworkX.

The goal is to experimentally analyse properties of Algorithm 1, like the number of iterations, space and time complexity. We used linear regression and curve fitting to map these properties to the size and density of initial graphs graphs.

8.1 Benchmarks

Random instances of n tasks were generated as follows. Periods of tasks are selected uniformly in $\mathcal{H} = \{1, 2, 5, 10, 20, 50, 100\}$. $\mathcal{H}$ is a subset of the values presented by Kramer et al. [16] for the 2015 WATERS challenge and several authors dealing with the age latency for automotive applications [10, 3].

Release times r_i are uniformly selected in $\{0, 1, 2, 3, 4, 5\}$, while we fix the relative deadline D_i equal to the period of the task, i.e., $D_i = T_i$ for any task $t_i \in \mathcal{T}$. Graphs are randomly generated using the Python NetworkX function `dense_gnm_random_graph`. Nodes are arbitrary numbered from 1 to n. A directed acyclic graph is then built by replacing each edge $e = \{i, j\}$ with $i < j$ by an arc $e = (i, j)$.

For any number n of tasks, we set the number of arcs to $m_\ell = \left\lfloor \frac{(n(n-1)}{4} \right\rfloor$ for *low density* graphs and $m_h = \left\lceil \frac{(n(n-1)}{3} \right\rceil$ for *high density*. We start with $n = 5$ tasks with a step of 5. For each data point, 150 random instances were generated and an average value of the functions considered are shown.

8.2 Analysis of the computation time of Algorithm 1

For sufficiently large n, the hyperperiod of an instance is exactly $T = lcm\{\alpha \in \mathcal{H}\} = 100$. The consequence is that the number of duplicates (*resp.*, the number of arcs) of the expanded graph $P_N(\mathcal{G})$ is bounded by $T \times n$ (*resp.*, $T^2 \times n^2$).

We measured the running time and the number of iterations of Algorithm 1. We stopped at $n = 90$ tasks, since the running time exceeded 15 minutes on average for instances with higher values of n. Figure 10 reports the average running times and Figure 11 the average number of iterations following the number of tasks.

We observed that the running time of Algorithm 1 is a quadratic function of the number of tasks, and thus is linear in the number of arcs of the graph $\mathcal{G}$. Unsurprisingly, these running times are longer for high-density graphs. This observation seems to contradict the experimental results of Becker et al. [3]: indeed, they remarked that the average running time for the computation of the age latency of a chain is linear w.r.t the number of tasks. In this case, the number of arcs equals $n - 1$: the running time is then also linear w.r.t the number of arcs, which is coherent with our result.

We also noticed that the whole number of iterations of Algorithm 1 grows logarithmically on average. Our first experimental conclusion is thus that the convergence of the algorithm to the exact value seems to be a logarithmic function of the number of tasks. The long running time is thus due to the time needed to build the successive partial expansions and not to the increase of the number of iterations of the algorithm.

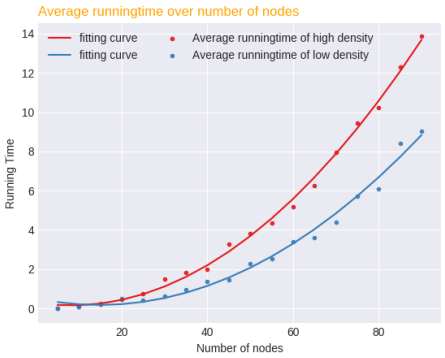


Figure 10 Average running times w.r.t the number of nodes. Fitting functions presented are $f_h(n) = (2.02\times10^{-3})n^2-0.03n+0.29$ and $f_\ell(n) = (1.53 \times 10^{-3})n^2 - 0.05n + 0.51$ for respectively high-density and low-density graphs.

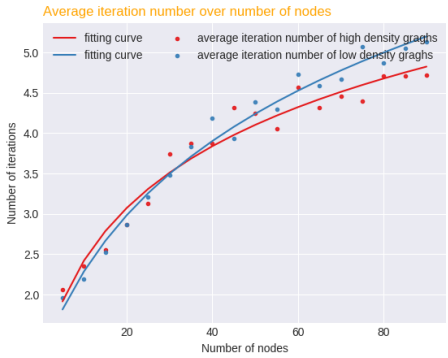


Figure 11 Average number of iterations w.r.t the number of nodes. Fitting functions presented are $g_h(n) = 1.34\ln(0.62(n + 5.89)) - 0.64$ and $g_\ell(n) = 1.96\ln(1.59(n + 13.42)) - 4.81$ for respectively high-density and low-density graphs.

8.3 Analysis of the partial expanded graph obtained

Figure 12 presents the evolution of the ratio $r(n) = \frac{\sum_{i=1}^{n} K_i^\star}{\sum_{i=1}^{n} N_i}$ following the number of tasks and the density of the graph. We observed that it is roughly a linear function that remains bounded by 0.8 for high-density graphs and 0.65 for low-density ones. The consequence is that in many cases we clearly do not need to completely expand the graph to get the exact value of the age latency and that good algorithms should be sought to identify the critical paths of a graph.

9 Conclusion

In this paper, we present a new definition of the dependency between the successive executions of two tasks that communicate following the LET paradigm. This definition was exploited to build a partial expanded graph $P_K(\mathcal{G})$ associated to any vector $K \in (\mathbb{N} - \{0\})^n$ for the computation of an upper bound of the age latency. A greedy algorithm to compute an accurate value $K^\star$ leading to the exact value of the age latency was developed and tested on random instances. This optimal partial expansion allows to identify the critical paths of the graph $\mathcal{G}$.

Many extensions of our study may be considered. The performance of our algorithm should be improved by building the successive partial expended graphs incrementally and optimizing data structures for graphs. Our methodology can surely be applied to evaluate accurate lower bounds of the age latency. Coupling the upper and the lower bounds will allow then to precisely measure the error between the longest paths of $P_K(\mathcal{G})$ and $\mathcal{L}^\star(\mathcal{G})$.

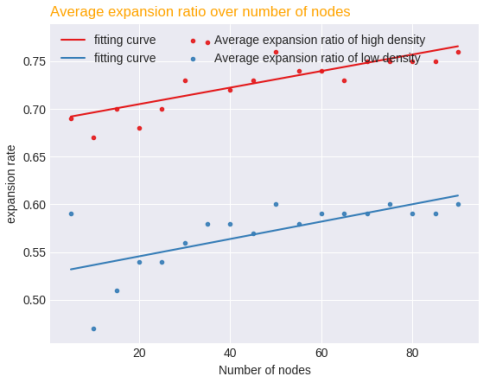


Figure 12 Average ratio $r(n) = \frac{\sum_{i=1}^{n} K_i^\star}{\sum_{i=1}^{n} N_i}$ for the partial expanded graph computed by Algorithm 1. Fitting functions presented are $r_h(n) = 8.67 \times 10^{-4} n + 0.69$ and $r_\ell(n) = 9.1 \times 10^{-4} n + 0.52$ for respectively high-density and low-density graphs.

Our general framework should also be extended to tackle other possible latencies [8]. Lastly, an implicit communication between two tasks of same period (which corresponds to two tasks in the same runnable for an AUTOSAR compatible system) could easily be considered in our model.

References

1 Autosar. URL: `https://www.autosar.org`.

2 Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. Synthesizing job-level dependencies for automotive multi-rate effect chains. In *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 159–169, August 2016. `doi:10.1109/RTCSA.2016.41`.

3 Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. End-to-end timing analysis of cause-effect chains in automotive embedded systems. *Journal of Systems Architecture*, 80:104–113, 2017.

4 Alessandro Biondi and Marco Di Natale. Achieving predictable multicore execution of automotive applications using the LET paradigm. In *IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2018, 11-13 April 2018, Porto, Portugal*, pages 240–250, 2018. `doi:10.1109/RTAS.2018.00032`.

5 Bruno Bodin, Alix Munier Kordon, and Benoît Dupont de Dinechin. K-periodic schedules for evaluating the maximum throughput of a synchronous dataflow graph. In *2012 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS XII, Samos, Greece, July 16-19, 2012*, pages 152–159, 2012.

6 Bruno Bodin, Alix Munier Kordon, and Benoît Dupont de Dinechin. Optimal and fast throughput evaluation of CSDF. In *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*, pages 160:1–160:6, 2016.

7 Robert de Groote. *On the analysis of synchronous dataflow graphs: a system-theoretic perspective*. PhD thesis, University of Twente, 2016.

8 Nico Feiertag, Kai Richter, Johan Nordlander, and Jan Jonsson. A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. In *IEEE Real-Time Systems Symposium, November 30-December 3*. IEEE Communications Society, 2009.

9 Julien Forget, Frédéric Boniol, and Claire Pagetti. Verifying end-to-end real-time constraints on multi-periodic models. In *22nd IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2017, Limassol, Cyprus, September 12-15, 2017*, pages 1–8, 2017.

10 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, June 27-30, 2017, Dubrovnik, Croatia*, pages 10:1–10:20, 2017.

11 Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, Falk Wurst, and Dirk Ziegenbein. Waters industrial challenge 2017. URL: `https://waters2017.inria.fr/challenge/#Challenge17`.

12 Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

13 Thomas A. Henzinger, Christoph M. Kirsch, Marco A.A Sanvido, and Wolfgang Pree. From control models to real-time code using Giotto. *IEEE Control Systems Magazine*, 23(1):50–64, February 2003.

14 Jad Khatib, Alix Munier Kordon, Enagnon Cédric Klikpo, and Kods Trabelsi-Colibet. Computing latency of a real-time system modeled by synchronous dataflow graph. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS 2016, Brest, France, October 19-21, 2016*, pages 87–96, 2016.

15 Christoph M. Kirsch and Ana Sokolova. The logical execution time paradigm. In Samarjit Chakraborty and Jörg Eberspächer, editors, *Advances in Real-Time Systems*, pages 103–120. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

16 Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free, 2015. URL: `https://www.ecrts.org/forum/viewtopic.php?f=20&t=23`.

17 Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceeding of the IEEE*, vol. 75(no. 9):pp. 1235–1245, 1987.

18 Qing Li and Caroline Yao. *Real-time concepts for embedded systems.* Taylor and Francis, Hoboken, NJ, 2014. URL: `http://cds.cern.ch/record/1990357`.

19 C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973. `doi:10.1145/321738.321743`.

20 Jorge Martinez, Ignacio Sañudo, and Marko Bertogna. Analytical characterization of end-to-end communication delays with logical execution time. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2244–2254, November 2018.

21 Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems*, 21(3):307–338, 2011.

22 Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard, and Pierre Siron. The ROSACE case study: from simulink specification to multi/many-core execution. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 309–318, April 2014.

23 Rémy Wyss, Frédéric Boniol, Claire Pagetti, and Julien Forget. End-to-end latency computation in a multi-periodic design. In *28th Symposium On Applied Computing (SAC'13)*, pages 1682–1687, Coimbra, Portugal, April 2013.

Control-System Stability Under Consecutive Deadline Misses Constraints

Martina Maggio
Saarland University, Department of Computer Science, Saarbrücken, Germany
Lund University, Department of Automatic Control, Sweden
Robert Bosch GmbH, Renningen, Germany
maggio@cs.uni-saarland.de

Arne Hamann
Robert Bosch GmbH, Renningen, Germany
arne.hamann@de.bosch.com

Eckart Mayer-John
Robert Bosch GmbH, Renningen, Germany
eckart.mayer@de.bosch.com

Dirk Ziegenbein
Robert Bosch GmbH, Renningen, Germany
dirk.ziegenbein@de.bosch.com

Abstract

This paper deals with the real-time implementation of feedback controllers. In particular, it provides an analysis of the stability property of closed-loop systems that include a controller that can sporadically miss deadlines. In this context, the weakly hard m-K computational model has been widely adopted and researchers used it to design and verify controllers that are robust to deadline misses. Rather than using the m-K model, we focus on another weakly-hard model, the number of consecutive deadline misses, showing a neat mathematical connection between real-time systems and control theory. We formalise this connection using the joint spectral radius and we discuss how to prove stability guarantees on the combination of a controller (that is unaware of deadline misses) and its system-level implementation. We apply the proposed verification procedure to a synthetic example and to an industrial case study.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Embedded and cyber-physical systems; Mathematics of computing → Mathematical analysis; Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases Real-Time Control, Deadline Misses, Weakly Hard Models

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.21

Funding This work was supported by: the ELLIIT Strategic Research Area, the project *ARAMiS II* of the German Federal Ministry for Education and Research with the funding ID 01IS16025. The project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871259 (ADMORPH). The responsibility for the content remains with the authors.

Acknowledgements This research was developed while Martina Maggio was on sabbatical at Robert Bosch GmbH.

1 Introduction

The contribution of this paper is a verification procedure to prove the robustness of controller implementations to deadline misses. For this task, literature contributions focus on the computation model where the control task can *miss* at most m deadlines in a window of K

activations (i.e., the m-K model). This was one of four proposed models [3] to analyse systems with deadline misses. In this paper we show that there is a natural analytical connection between another of these four models, the number of maximum consecutive deadline misses, and control-theoretical tools that can be used to prove properties of closed-loop systems, such as stability.

Historical Perspective. During the past couple of decades the real-time systems community made an effort in formalising requirements, models, and algorithms to handle systems that can (sporadically) miss deadlines. In this quest, Hamdaoui and Ramanathan analysed tasks that behave according to the (m, k) model [18]. With this model, tasks can be modelled as sequences of jobs. In every set of k consecutive jobs, at least m jobs must meet their deadline. This model was analysed, finding schedulability conditions and scheduling schemes, e.g., [35]. Building on the ideas from this research, the *weakly hard* model of computation was formalised [3]. A weakly hard real-time system is a system in which the distribution of deadline misses and hits during a window of time is precisely bounded.

Bernat, Burns and Liamosí [3] give four possible definitions for a weakly hard task τ.

1. $\tau \vdash \binom{n}{m}$, with $1 \leq n < m$: According to this definition, for each set of m consecutive deadlines, τ meets at least n of them. With a slight difference in notation, this model corresponds to the (m, k) model by Hamdaoui and Ramanathan [18].
2. $\tau \vdash \left\langle {n \atop m} \right\rangle$, with $1 \leq n < m$: Here, the system guarantees that for each set of m consecutive deadlines, τ meets consecutively n of them.
3. $\tau \vdash \overline{\binom{n}{m}}$, with $1 \leq n < m$: This is the dual definition with respect to the first one. In this case, the system guarantees that for each set of m consecutive deadlines, τ misses at most n of them.
4. $\tau \vdash \overline{\langle n \rangle}$, $n \geq 1$: According to this definition the maximum number of consecutive deadline misses that τ can experience is n.[1]

The third of these models gained traction in the research community, and the term weakly-hard task started to indicate a task that can experience a bounded number of misses in a window of jobs. In particular, with a slightly confusing terminology, this third model is also often called the m-K model.[2] This specifies that a task can experience at most m misses in a window of K consecutive jobs [1, 9, 10, 12, 14, 19, 20, 33, 34, 41–43].

The m-K Model for Control Tasks. In the attempt to achieve computing and control co-design, both schedulability results like [43] and analysis using model checking [12] have been investigated. The research community contributed with criteria to determine stability [5], determine convergence rates [14], and to design controllers in the presence of deadline misses [29]. Furthermore, the performance cost of deadline misses was investigated [34, 44], together with the role of the strategy used to handle the misses [33, 41], i.e., killing the task or allowing its continuation with different policies for the following iteration. The general consensus is that sequences of misses and hits (i.e., the m-K model) are to be considered and analysed to determine physical properties of the system.

---

[1] The original definition was $\tau \vdash \overline{\left\langle {n \atop m} \right\rangle}$, with $1 \leq n < m$. However, according to [3, Theorem 4] there is no need to specify the window size, i.e., the task τ can be equivalently defined using any window, hence we use $\tau \vdash \overline{\langle n \rangle}$.

[2] Notice the difference between the (m, k) model (that stated that there were m hits for every sequence of k consecutive jobs, with $1 \leq m < k$) and the m-K model (that imposes that there are at most m misses in every window of K jobs, with $1 \leq m < K$).

Contribution. In this paper we use the $\tau \vdash \overline{\langle n \rangle}$ model and cast the problem of verifying the stability of closed-loop systems into a neat mathematical framework. The solution ensures the stability of the combination of: (1) the physics of the plant under control and, (2) the execution of the controller (that may miss deadlines). We believe that the $\tau \vdash \overline{\langle n \rangle}$ model is as relevant as the m-K model from the industrial standpoint.

In fact, in industrial products, deadline misses are often caused by transient overload periods or faults. In many industrial applications, a system load of over 80% is targeted for cost reasons. Such a high system load can usually not be achieved with purely formal methods that are based on worst-case considerations, especially on multi-core platforms. For this reason, many industrial systems are designed for average runtimes plus a safety margin, in conjunction with rate monotonic scheduling. Our experience is that, following this practical approach in contrast to analytical ones, deadline violations may occur, e.g., due to a transient interrupt load. During these transient periods the miss ratio often is quite high, making the number of consecutive misses a very relevant indicator of the system performance. We argue that methods that evaluate and prove the robustness of controllers to deadline violations in this setup are of high industrial interest.

Outline. In the remainder of this paper we will recap the necessary control background and then provide our contribution. In particular, Section 2 explains how a plant is modelled and how a *state feedback controller* is applied to regulate the plant's behaviour. Section 3 describes the strategies that are typically used to handle deadline misses and provides some insights on what is the best choice from the system perspective. Section 4 shows how to guarantee properties of closed-loop systems (like stability) in the presence of deadline misses with the $\tau \vdash \overline{\langle n \rangle}$ model. Section 5 shows some experimental results validating our claims. Finally, Section 6 presents an overview of related work and Section 7 concludes the paper.

2 Control Background

In this section we recap the basic concepts of control theory that are used in the rest of the paper. We analyse linear time-invariant models and controllers implemented as periodic tasks with implicit deadlines.

Plant Model. The starting point for control design is always understanding the object that the controller should act upon. The control engineer obtains a model $\mathcal{P}_c$ of the plant to control. In most cases, this model is linear and time-invariant, and represents with ordinary differential equations the dynamics of the system in the following form.

$$\mathcal{P}_c : \begin{cases} \dot{x}(t) = A_c\, x(t) + B_c\, u(t) \\ y(t) = C_c\, x(t) + D_c\, u(t) \end{cases} \tag{1}$$

Here, the system state $x(t) = [x_1(t), \ldots, x_p(t)]^{\mathrm{T}}$ evolves depending on the current state and the input signal $u(t) = [u_1(t), \ldots, u_r(t)]^{\mathrm{T}}$, where the superscript $^{\mathrm{T}}$ indicates the transposition operator. We denote with p the number of state variables (i.e., the length of vector x) and with r the number of input variables (i.e., the length of vector u). The matrices A_c, B_c, C_c, and D_c encode the dynamics of the system. In the following, we will make two assumptions: D_c is a zero matrix of appropriate size, and C_c is the unit matrix of appropriate size.[3] The

[3] A_c is a $p \times p$ matrix, B_c is a $p \times r$ matrix, C_c is a $p \times p$ matrix, and D_c is a $p \times r$ matrix.

first assumption means that the system is *strictly proper* and holds for almost all the physical models used in control. The second assumption means that the state is measurable. This does not always hold for real systems, but state observers can be built whenever this is not true [26], to estimate the state $x(t)$.[4] This means that, without losing generality, we can represent $\mathcal{P}_c$ as

$$\dot{x}(t) = A_c\, x(t) + B_c\, u(t), \tag{2}$$

and describe the system dynamics using only A_c and B_c.

From this starting point, control systems are usually designed and realised in one of these two ways:

1. The plant model $\mathcal{P}_c$ is used to synthesise a controller (model) $\mathcal{C}_c$ in continuous-time. Closed-loop system properties, like stability, are proven on the feedback interconnection of $\mathcal{C}_c$ and $\mathcal{P}_c$. However, when the controller is implemented, digital hardware is used. This means that the controller model $\mathcal{C}_c$ has to be discretised, obtaining $\mathcal{C}_d$. $\mathcal{C}_d$ describes the behaviour of $\mathcal{C}_c$ at given sampling instants.
2. The model of the plant in continuous time $\mathcal{P}_c$ is discretised, obtaining a discrete-time plant model $\mathcal{P}_d$. $\mathcal{P}_d$ describes the behaviour of $\mathcal{P}_c$ at given sampling instants. A controller $\mathcal{C}_d$ is designed directly in the discrete-time framework, using the discrete-time plant model $\mathcal{P}_d$. Closed-loop properties are proven on the feedback interconnection of $\mathcal{C}_d$ and $\mathcal{P}_d$.

In both cases, when an object (being it the plant or the controller) is discretised, a sampling period π is chosen. With either design methods, we can obtain a discrete-time model of the plant $\mathcal{P}_d$ and of the controller $\mathcal{C}_d$. In control theory, usually it is possible to prove properties of the interconnection between these two models. In particular, we use $\mathcal{C}_d$ rather than $\mathcal{C}_c$ to prove properties using the controller that is closer to the real implementation. However, on top of what is done in classical control theory, we want to take into account deadline misses.

We discretise $\mathcal{P}_c$ from Equation (2). From the representation in terms of ordinary differential equations, we obtain the system of difference equations $\mathcal{P}_d$ as

$$\mathcal{P}_d : x_{[k+1]} = A_d\, x_{[k]} + B_d\, u_{[k]}. \tag{3}$$

Here, k counts the sampling instants (i.e., there is a distance of π [s] between the k-th and the $k+1$-th instant). The matrices A_d and B_d are the counterparts of A_c and B_c for the continuous-time system. They describe the evolution of the system in discrete-time, have the same dimensions of the corresponding continuous-time matrices, and their elements depend on the choice of the sampling period π.

Controller Model. Once a model of the plant is available, control design can be carried out with many different methods. In this paper we tackle periodic controllers expressed as state feedback controllers, i.e., controllers that execute periodically with period π and whose discrete-time form is

$$u_{[k]} = K_k\, x_{[k]}. \tag{4}$$

The control design problem is the problem of finding the matrix K_k that stabilises the system and obtains some desired properties. The state feedback formulation is more general than it may seem at a first glance. State feedback controllers are not purely proportional controllers,

[4] As a remark, if a state observer is present its dynamics should be taken into account in the analysis. This extension only requires to augment the system state with the rows and columns corresponding to the execution of the observer, but the analysis method remains the same.

although their update is proportional to the state vector. It is possible to augment the state vector of the system – for example introducing an error term and its integral – to achieve controllers that are not simply proportional but contain integral action.[5] Additionally, it is possible to use pole placement [26], or to compute optimal controllers using the Linear Quadratic Regulator [25] formulation.

In an industrial setting,[6] many controllers are still designed assuming zero latency and instantaneous computation [46], i.e., assuming that it takes zero time to retrieve the sensor measurement from the plant, compute the control signal, and apply it. When the dynamics of the plant are slow and the controller is able to sample and measure signals at a reasonable speed, this assumption does not significantly affect the behaviour of the system. However, in most cases, basic properties like stability can be violated because of the computational delays that are introduced in the loop. The controller job that is activated at time t_a completes its execution at time t_c, where t_c is in the controller period, i.e., $t_c \in (t_a, t_a + \pi]$, introducing a computational delay $t_c - t_a$.

Due to this computational delay, in industry, it became common practice to design control systems following the Logical Execution Time (LET) paradigm and to synchronise input and output exactly to the period boundary. In this case, the control signal is computed within a control period and applied at the beginning of the next period. This enhances the predictability of the system, allows the processor to execute other tasks without affecting the control properties, and ensures a consistent behaviour.

In control terms, this means that the controller actuates its control signal computation with a one-step delay. Assuming that the cycle of sampling, computing, and actuating can be always terminated within a control period, this allows the designer to synthesise an optimal controller regardless of the time-varying components of the computational delay such as activation jitter, unpredictable interrupts, uncertain computation times [28]. The equation for the state feedback controller then becomes

$$\mathcal{C}_d : u_{[k]} = K\, x_{[k-1]}, \tag{5}$$

where K is the designed controller. With very few exceptions, the vast literature on control design assumes that the deadlines to compute control signals are always met. Recently, Linsenmayer and Allgöwer started to connect the theory of m-K real-time systems (i.e., the $\tau \vdash \overline{\binom{m}{K}}$ model) with control design [29], showing that it is in some cases possible to design a state feedback controller that is robust (i.e., guarantees stability) to deadline misses. In this paper we will connect the amount of possible consecutive deadline misses (i.e., the $\tau \vdash \overline{\langle n \rangle}$ model) to the analysis of stability as a control design property.

[5] The most widely adopted controllers in industry are the Proportional and Integral (PI) or the Proportional, Integral and Derivative (PID) controllers. These controllers can be expressed in state-feedback form (as seen later in Section 5 for a specific example), by augmenting the system state $x(t)$ with the difference between the desired state values and the obtained ones, i.e., the error, and its integral, or sum, over time. There is a small difference between the controller expressed in state-feedback form and the controller expressed as a state-space system. In the first case, when the controller misses its deadline, the update function for the state is still executed (as it is now part of the system equation). It is however possible to generalise the findings in this paper to handle controllers in state-space form.

[6] In fact, a survey published in 2001 by Honeywell [11] states that 97% of the existing industrial controllers are PI controllers and use no delay compensation. This does not mean that the control community has not developed solutions to properly address delays in the control design. It simply means that in many industrial settings the design is still simple and limited to considering the computation instantaneous.

Feedback Interconnection. Assume there are no deadline misses. In this case, we can plug the value of $u_{[k]}$ obtained from Equation (5) into the plant Equation (3), obtaining

$$x_{[k+1]} = A_d\, x_{[k]} + B_d\, K\, x_{[k-1]}. \tag{6}$$

To analyse the closed-loop system, we define a new state variable $\tilde{x}_{[k]} = [x_{[k]}^{\mathrm{T}}, x_{[k-1]}^{\mathrm{T}}]^{\mathrm{T}}$ (the superscript $^{\mathrm{T}}$ indicates the result of the transposition operator). We recall that p denotes the order of the system (i.e., the number of state variables in vector $x_{[k]}$). Using the new state variable $\tilde{x}_{[k]}$, Equation (6) can be rewritten as

$$\tilde{x}_{[k+1]} = \begin{bmatrix} x_{[k+1]} \\ x_{[k]} \end{bmatrix} = \underbrace{\begin{bmatrix} A_d & B_d\, K \\ I_p & 0_{p\times p} \end{bmatrix}}_{A} \begin{bmatrix} x_{[k]} \\ x_{[k-1]} \end{bmatrix} = A\, \tilde{x}_{[k]}, \tag{7}$$

where I_p and $0_{p\times p}$ are respectively the identity matrix and the zero matrix of size of the number of state variables p.

Stability. A discrete-time linear time-invariant system is asymptotically stable if and only if all the eigenvalues of its state matrix are strictly inside the unit disk. For the system shown in Equation (7), this means that the eigenvalues of A should have magnitude strictly less than one.

Another way of formulating the stability requirement uses the concept of *spectral radius* $\rho(A)$. The spectral radius is defined as the maximum magnitude of the eigenvalues of A. If we denote with $\{\lambda_1, ..., \lambda_n\}$ the set of eigenvalues of A, this means

$$\rho(A) = \max\{|\lambda_1|, \ldots, |\lambda_n|\}. \tag{8}$$

Requiring that all the eigenvalues have magnitude strictly less than one is equivalent to stating that the spectral radius of the A matrix should be less than 1.

This only proves the stability of the system in absence of deadline misses. However, we are aware that sporadic misses can occur, either due to faults [16] or to the chosen period π not satisfying the requirement of worst-case response time for the controller task τ being less than the controller period [12, 33, 34].

3 Deadline Miss

In order to properly analyse the closed-loop system properties when deadlines can be missed, it is necessary to define a model of how the system reacts to deadline misses. There are two aspects of this reaction: (i) what is the chosen control signal when a miss occurs [29], and (ii) how is the operating system treating the job that missed the deadline [33]. In the remainder of this section, suppose that in the k-th iteration the controller task τ did not complete its execution before the deadline, i.e., it does not complete its computation before time $(k+1)\,\pi$ [s]. We denote time $(k+1)\,\pi$ [s] with t_m.

Control Signal. At time t_m, a control signal should be applied to the plant. Two alternatives have been identified for how to select the next control signal [29]: zero and hold.

1. *Zero:* The control signal $u_{[k+1]}$ is set to zero.
2. *Hold:* The control signal $u_{[k+1]}$ is unchanged, i.e., it is the previous value of the control signal $u_{[k]}$.

The choice of these two alternatives often depends on the control goal that should be achieved.

When a controller is designed for setpoint tracking (i.e., to ensure that the value of some physical quantity follows a desired profile – e.g., to have a robot follow a desired trajectory), the control signal is usually zero in case the measured physical quantity is equal to its setpoint. In this case, setting the control signal to zero means assuming that the model of the plant is correct and the computation does not need correction. When a controller is designed for disturbance rejection (i.e., to ensure that the effect of some physical disturbance is not visible in the measurements – e.g., to keep the altitude of a helicopter constant despite wind) then the control signal is usually a reflection of the effort needed to counteract the disturbance. In this case, holding the previous value of the control signal means making the assumption that the system is experiencing the same disturbance.

System-Level Action. The second decision to make is the choice of what to do with the job that missed the deadline. In this case three alternatives have been proposed [33]: kill, skip-next, and queue(1).

1. *Kill:* At time t_m the job that missed the deadline is killed and a new job is activated.
2. *Skip-next:* At time t_m the job that missed its deadline is allowed to continue with the same scheduling parameters (e.g., priority or budget) and carries on in the next period. The job that should have been activated at the deadline missed is not activated, and the next activation is set to $t_m + \pi$.
3. *Queue(1):* At time t_m the job that missed its deadline is continued. A new job is activated with deadline $t_m + \pi$. The two jobs share the scheduling parameters during the period interval $[t_m, t_m + \pi]$. At time $t_m + \pi$, the most recent update of the control value is applied. If both jobs finish their computation, the control variable is set to the value produced by the most recently activated job (i.e., the job that started at time t_m and was placed in the queue until the old job that missed its previous deadline finished). If only the first job finishes the computation the control variable is set to the value of the job that finished and the following one is continued in the subsequent period.

In Section 4 we will analyse the system in all possible configurations. However, we point out that, from an implementation perspective, killing the control job may not be feasible in many industrial settings. In fact, the system has reached an inconclusive intermediate state. The internal state of the controller could have been updated and the system should implement a clean rollback of these changes. Implementing a clean rollback procedure is risky. Furthermore, if the lengthy computation (and subsequent deadline miss) is due to the received input values, it is likely that the next iteration will start from state values that are fairly close to the previous ones, with higher than normal risk of missing a deadline.

We also notice that enqueuing the task could be beneficial from the control perspective, because a computation with most recent measurements of the state variables could be applied. However, the scheduling parameters for τ have most likely been tuned for one single control job to be executed in a period. For example, if the control task is executed using reservation-based scheduling, its budget is selected to match one execution. When using fixed-priority scheduling, the controller priority has been selected. Executing a second control task may create ripple effects and have a disruptive effect on lower priority tasks.

Finally, if the deadline is missed, this means that the system is likely experiencing a transient overload state, which would make *skip-next* the best option to relieve some pressure from the system.

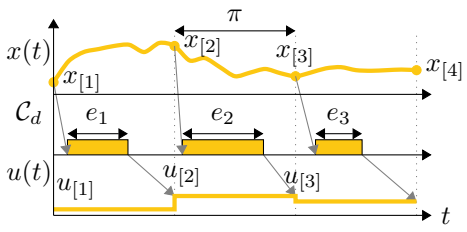


Figure 1 System evolution in case no deadline is missed. The state feedback controller $\mathcal{C}_d$ computes the control signal u based on measurements of the state x.

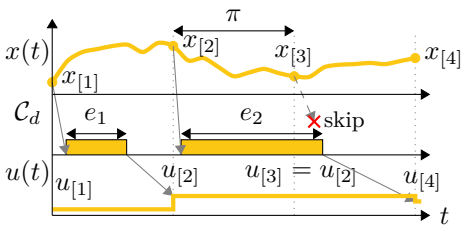


Figure 2 System evolution in case of a deadline miss with the hold policy and skip-next strategy. The controller misses the deadline and completes in the subsequent period.

4 System Analysis

In this section we present our analysis of the closed-loop system with deadline misses. We first discuss the fundamentals of what happens from the physical perspective when a deadline is missed and then discuss the combinations identified in Section 3 for how the system handles the miss.

Fundamentals. Here we present the general methodology that we apply to verify the stability of closed-loop systems with different strategies. We cast the problem into a switching-systems stability problem and show how real-world implementations behave.

Within one control period, there are two possible realisations. The controller job that was activated at time $k\,\pi$ can either hit or miss its deadline. Figure 1 shows the case in which no deadline is missed, while Figure 2 shows the behaviour of the system when a deadline miss occur with the *hold and skip-next* strategy. In the figures, we use e_i to indicate the execution time of the i-th job of the controller. The figure just provides a visual representation of a lengthy execution, but misses can occur due to other sources of interference, e.g., higher priority task being executed with a fixed-priority scheduling algorithm, interrupts being raised and served during the execution of the control task, or access to locked shared resources being requested. In Figure 2, the control signal $u_{[2]}$ is held as $u_{[3]}$. The next controller execution instance is skipped and the result of the completion of e_2 is applied as $u_{[4]}$.

The procedure that we follow to analyse the closed loop system is the following:

1. We express the dynamics of the closed-loop system in the cases of hit and of miss. Following a procedure similar to the one we used in Equation (7), we determine the state matrices for the closed-loop systems in case of deadline hit and deadline miss, respectively A_H and A_M. We then know that the system with (unconstrained) deadline misses can be expressed as a switching system [27] that arbitrarily switches between these two matrices. If the original system in Equation (3) was unstable, there is no hope that the switching system that arbitrarily switches between A_H and A_M is stable (as either an old or no control action is applied when a miss occurs). However, we still have not introduced any weakly hard constraint.
2. We determine the set of possible cases for the evolution of the system when $\tau \vdash \overline{\langle n \rangle}$ guarantees are provided, i.e., the possible realisations of the system behaviour. We denote with Σ the set of possible matrices that these realise. For $\tau \vdash \overline{\langle n \rangle}$ guarantees, the set of possible realisations is $\{H, MH, \ldots, M^n H\}$. The set contains either a single

hit, or a certain number of misses (up to n) followed by a hit.[7] This means that $\Sigma = \{A_H, A_H A_M, A_H A_M^2, ... A_H A_M^n\}$. This can be written in a compact form as $\Sigma = \{A_H A_M^i \mid i \in \mathbb{Z}^{\geq}, i \leq n\}$ where $\mathbb{Z}^{\geq}$ indicates the set of integers including zero. Notice that matrices are multiplied from the right to the left (denoting the standard evolution of the system from a mathematical standpoint). This step introduces the weakly hard constraint for which we investigate the system stability.

3. We compute a generalisation of the spectral radius concept, called *joint spectral radius* $\rho(\Sigma)$ [21, 36], that allows us to assess the stability of the closed-loop system that switches between the realisations (i.e., the valid scenarios including a number of misses between 0 and n followed by a hit) included in Σ. More precisely, the closed-loop system that can switch between the realisations included in Σ is asymptotically stable *if and only if* $\rho(\Sigma) < 1$ [21, Theorem 1.2].

In order to generalise the spectral radius to a set of matrices, we introduce some notation. The following paragraphs are using the notation and sequential treatise proposed in [21] to introduce the concept of the joint spectral radius. We recap only what is needed for the purpose of understanding our analysis.

Joint Spectral Radius [21, 36]. The first step for our definition is to determine what happens when some steps of evolution of the switching system occur. We then denote with $\rho_\mu(\Sigma)$ the spectral radius of the matrices that we find after μ-steps. Precisely,

$$\rho_\mu(\Sigma) = \sup\{\rho(A)^{1/\mu} : A \in \Sigma^\mu\}. \tag{9}$$

In this definition we quantify the average growth over μ time steps, as the supremum of the spectral radius (elevated to the power $1/\mu$) of all the matrices that can be evolutions of the system after μ matrix multiplications (i.e., after μ evolution steps, where an evolution step is either a hit or a set of constrained misses followed by a hit). Equation (9) denotes the supremum of all the possible combinations of products of μ matrices that are included in Σ.

Using $\rho_\mu(\Sigma)$ we can define the joint spectral radius of a bounded set of matrices Σ as

$$\rho(\Sigma) = \limsup_{\mu \to \infty} \rho_\mu(\Sigma). \tag{10}$$

We are then looking at the evolution of the system for an infinite amount of time, i.e., pushing μ to the limit.

Determining that the switching system is asymptotically stable is equivalent to assessing that the joint spectral radius of the set of matrices Σ is less than 1. This condition is both sufficient and necessary [21, Theorem 1.2]. This means that if the joint spectral radius is higher than 1, there is at least a sequence of switches of hits and misses that destabilises the closed-loop system.

Joint Spectral Radius Computation. On the practical side, the problem of computing if the joint spectral radius is less than 1 is undecidable [8]. In many cases it is possible to approximate the joint spectral radius with satisfactory precision [6, 7, 17, 32] and obtain upper

[7] The notation used to define Σ is slightly simplified here, as the matrix A_H may be different depending on how many deadlines have been missed (for example, with the skip-next strategy the controller uses an old measurement of the state to compute the control signal). We will be more precise in the following when we show how to apply the procedure to the different cases. Furthermore, notice that the matrices in Σ represent the evolution across a different number of time steps: A_H advances the time in the system of π, while $A_H A_M$ advances the system time of $2\,\pi$. This is not a concern for the system analysis.

and lower bounds for $\rho(\Sigma)$. Clearly, the closest the two bounds are, the more precise is the estimation of the true value of the joint spectral radius. We can safely say that our controller design is sufficiently robust to deadline misses if the upper bound on the joint spectral radius $\rho(\Sigma)$ is less than 1.

Joint Spectral Radius with at most n Consecutive Misses. If the joint spectral radius of the set Σ is less than 1, the stability of all the combinations of realisations (of hits and misses, that include at most n consecutive misses) is proven, regardless of the window size.

For example, let us assume that we are analysing a system with the real-time guarantee that we cannot experience more than two consecutive misses. The realisations that we analyse are $\{A_H, A_H A_M, A_H A_M A_M\}$ and the joint spectral radius unfolds and checks all the possible (infinitely long) sequences of combinations of these realisations.

For a length of two, this means that we check: (1) $A_H A_H$ as the product of the first term twice, (2) $A_H A_H A_M$ as the product of the first two terms picking the first as final (in terms of time evolution of the system), (3) $A_H A_H A_M A_M$ as the product of the first and last terms picking the first as initial, (4) $A_H A_M A_H$ as the product of the first two terms picking the second as final, (5) $A_H A_M A_H A_M$ as the product of the second term twice, (6) $A_H A_M A_H A_M A_M$ as the product of the last two terms, picking the second as final, (7) $A_H A_M A_M A_H$ as the product of the last and first term, (8) $A_H A_M A_M A_H A_M$ as the product of the last and second term, (9) $A_H A_M A_M A_H A_M A_M$ as the last term twice. This procedure is repeated for more products, up to *infinitely long* sequences. From the computation side, the results are an upper and a lower bound on the value of the (true) joint spectral radius.

The analysis is sound on the control side, as stability is guaranteed if and only if the joint spectral radius is less than 1. On the theoretical side, this demonstrates that the $\tau \vdash \overline{\langle n \rangle}$ model can elegantly provide a necessary condition for the stability of the system. The only if part means that there is at least a sequence of hits and misses (where at most we experience n consecutive misses) that causes the system to be unstable if the (true value of the) joint spectral radius is larger than 1.

Considering that we only compute an upper and lower bound on the joint spectral radius, what we can conclude is: if the lower bound that we obtain is above 1, we are entirely certain that such a sequence exists, while if the lower bound is below 1 and the upper bound is above 1 we have no mathematical certainty that the system is unstable. Nonetheless, on the practical side, the bounds obtained with modern approximation techniques [6, 7, 17, 32] are usually very close to one another, implying that they are a very good estimate of the true value of the joint spectral radius.

It is important to note that even if the control system is able to stabilise the system in the presence of n consecutive misses, this does not mean that executing the controller with a period of $n\,\pi$, rather than its original period π, is a sensible choice. In fact, the performance (measured for example using the integral of the squared error) of the controller that is executing with a larger period would be dramatically worse than the performance of the controller with the shorter period that can experience misses. Being able to tolerate misses is very different than performing well when these misses occur.

Relation with m-K model. We here briefly discuss the relation between the guarantees that we obtain with the $\tau \vdash \overline{\langle n \rangle}$ model and the m-K model, $\tau \vdash \overline{\binom{m}{K}}$.

The $\tau \vdash \overline{\langle n \rangle}$ model includes all the realisations that are contained in the $\tau \vdash \overline{\binom{n}{K}}$ regardless of the value of K. However, it can also include additional realisations (that could generate instability) that are not included in the $\tau \vdash \overline{\binom{n}{K}}$ model, if $K > n + 1$. The $\tau \vdash \overline{\langle n \rangle}$ model

over-approximates the set of possible realisations that one can obtain with an m-K task (assuming that $n = m$). This means that there is a chance that an m-K control task stabilises the system when the corresponding task with $n = m$ consecutive deadline misses would not.

The difference between n and K determines the extent of the potential over-approximation. With a smaller difference, the set of possible realisations converges to the set of realisations that are included in the $\tau \vdash \overline{\langle n \rangle}$ model. More precisely, the set of possible realisations considered with the $\tau \vdash \overline{\binom{n}{n+1}}$ model is the same as the set obtained with the $\tau \vdash \overline{\langle n \rangle}$ model. However, increasing K, reduces the set of possibilities that are considered, shrinking the size of the set of valid realisations. Figure 3 shows the relationship between three sets when $K_2 > K_1 > n + 1$.

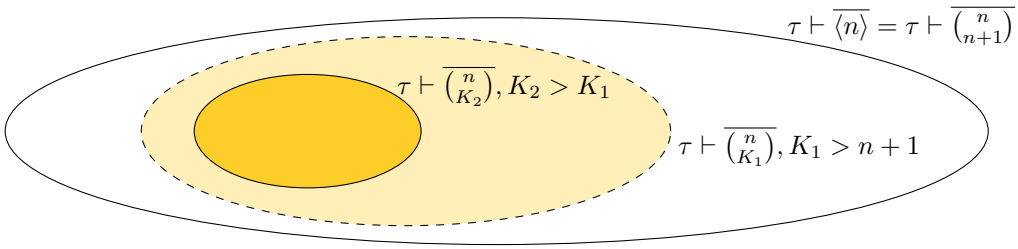


Figure 3 Sets of potential realisations with different models.

If the system with $\tau \vdash \overline{\langle n \rangle}$ is found stable, then control task τ that is given m-K guarantees also stabilises the plant if $m \leq n$. This means that as a first approximation, regardless of the value of K, when dealing with the m-K model, one can check the stability for a maximum of m consecutive deadline misses and if the condition is satisfied, then the closed-loop is stable regardless of the value of K.

As a final remark, from the industrial point of view, the analysis of the case when $K \gg n$ is not particularly interesting, because stability and quality of control guarantees are provided by the design of robust controllers [26] (i.e., a small perturbation in scheduling is anyway covered by the redundancy and design of control systems). On the contrary, there is a clear industrial interest in analysing the $K \approx n + 1$ case, due to transient heavy load perturbations.

Application. We now show how to apply the theory to practical case studies. We implemented our analysis methods in MATLAB®. The input values for our stability verification procedures are: n (the number of contiguous deadline that the system can miss), A_d and B_d (the matrices that determine the dynamics of the system), and K (the controller that is designed and should be validated). We used the JSR Matlab toolbox [22, 45] to compute bounds on the joint spectral radius. We constructed the set Σ based on the expressions derived for the given deadline-handling methods, i.e., for the combination of the control signal management policy (*zero* or *hold*) and the system-level action (*kill*, *skip-next*, or *queue(1)*).

Zero&Kill. The zero and kill strategy is the simplest to analyse. For this strategy we can look at Equations (3) and (5). The system behaves according to

$$\tilde{x}_{[k+1]} = \begin{bmatrix} x_{[k+1]} \\ u_{[k+1]} \end{bmatrix} = \underbrace{\begin{bmatrix} A_d & B_d \\ K & 0_{r \times r} \end{bmatrix}}_{A_H} \begin{bmatrix} x_{[k]} \\ u_{[k]} \end{bmatrix} = A_H \, \tilde{x}_{[k]} \tag{11}$$

in case of a hit. Notice that this is in principle exactly the same for each strategy, as when the controller hits the deadline the behaviour is the same. Kill implies that in case of a deadline

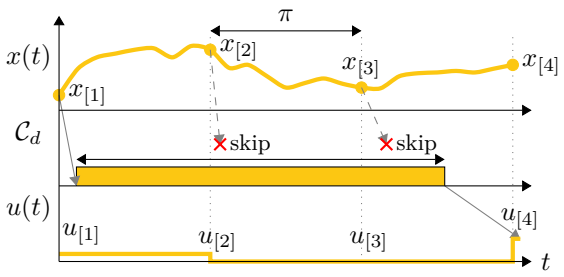


Figure 4 System evolution in case of two consecutive deadline misses with the skip-next strategy and the zero policy.

miss there is an abort of what the control task has been computing up to its deadline, which means there is no need to take into account its (partial) behaviour. In case of a deadline miss in the k-th iteration, the control signal $u_{[k+1]}$ is set to zero. This means that the system evolves according to

$$\tilde{x}_{[k+1]} = \begin{bmatrix} x_{[k+1]} \\ u_{[k+1]} \end{bmatrix} = \underbrace{\begin{bmatrix} A_d & B_d \\ 0_{r\times r} & 0_{r\times r} \end{bmatrix}}_{A_M} \begin{bmatrix} x_{[k]} \\ u_{[k]} \end{bmatrix} = A_M\, \tilde{x}_{[k]} \tag{12}$$

in case of a miss. With n maximum consecutive misses, we can then compute all the matrices in $\Sigma = \{A_H A_M^i \mid i \in \mathbb{Z}^{\geq}, i \leq n\}$[8] and then compute the upper bound on $\rho(\Sigma)$.

We would like to remark that while this strategy is simple to analyse, for practical applications it is hard to guarantee – using kill – that the control task τ will miss at most n consecutive deadlines (as the state of the running task is always re-set at every period start).

Zero&Skip-Next. The difference between Zero&Kill and Zero&Skip-Next lays in the freshness of the measurements that are used for the computation of the control signal when the task τ hits its deadline. In fact, if the task was killed and a new job was activated then the state measurement would occur at the beginning of the new activation, while if the task was allowed to continue, it would use old measurements. Figure 4 shows how the system evolves in case of consecutive deadline misses. Suppose that the control task τ completes its execution in the third period. Then this is not equivalent to experiencing two misses and a hit, because the completion uses old state measurements. We need to then express the state matrix evolution when there is a *recovery hit*, rather than a regular hit (in the figure $u_{[4]}$ is set using $x_{[1]}$).

To properly analyse this system, our state has to include the previous values that can be used for the control signal computation. With $\tau \vdash \overline{\langle n \rangle}$ guarantees, we can then define our augmented state vector as $\tilde{x}_{[k]} = [x_{[k]}^{\mathrm{T}}, x_{[k-1]}^{\mathrm{T}}, ..., x_{[k-n]}^{\mathrm{T}}, u_{[k]}^{\mathrm{T}}]^{\mathrm{T}}$, i.e., the state vector of the closed-loop system is composed of $n+1$ elements of the state vector and 1 element for the control signal.

[8] As defined in the introductory part of Section 4 (see point 2 in *Fundamentals*), $\mathbb{Z}^{\geq}$ indicates the set of integers including zero.

Then we can write the closed-loop system in case of a deadline hit, i.e., the equivalent of Equation (7), as

$$\begin{bmatrix} x_{[k+1]} \\ x_{[k]} \\ \dots \\ x_{[k-n+1]} \\ u_{[k+1]} \end{bmatrix} = \underbrace{\begin{bmatrix} A_d & 0_{p\times(n\cdot p)} & B_d \\ & I_{n\cdot p} & 0_{(n\cdot p)\times(p+r)} \\ K & 0_{r\times(n\cdot p)} & 0_{r\times r} \end{bmatrix}}_{A_H} \begin{bmatrix} x_{[k]} \\ x_{[k-1]} \\ \dots \\ x_{[k-n]} \\ u_{[k]} \end{bmatrix}. \tag{13}$$

Here, we added some padding to the matrix to identify that state variables are transferred from one time instant to the next; i.e., to add the trivial equations $x_{[i]} = x_{[i]}, \forall i \mid x-n+1 \leq i \leq k$. In fact, when the deadline is hit (not after a miss) there is no use of the previous values of the state.

When a miss occurs, the control signal is set to zero. This means that we can use the A_H matrix defined in Equation (13) and substitute the value of K with a zero matrix of appropriate size, i.e., $0_{r\times p}$ to obtain the A_M matrix,

$$\begin{bmatrix} x_{[k+1]} \\ x_{[k]} \\ \dots \\ x_{[k-n+1]} \\ u_{[k+1]} \end{bmatrix} = \underbrace{\begin{bmatrix} A_d & 0_{p\times(n\cdot p)} & B_d \\ & I_{n\cdot p} & 0_{(n\cdot p)\times(p+r)} \\ & 0_{r\times(n+1)\cdot p+r} & \end{bmatrix}}_{A_M} \begin{bmatrix} x_{[k]} \\ x_{[k-1]} \\ \dots \\ x_{[k-n]} \\ u_{[k]} \end{bmatrix}. \tag{14}$$

Now we should remark that a hit and a *recovery* hit have two different matrix realisation, i.e., a hit that follows a certain number of misses (up to n) has a different matrix with respect to A_H. In fact, we have to take into account the use of the old state measurement to produce the control signal of the recovery hit. We denote with A_{R_i} the matrix that represents the evolution of the closed-loop system when a recovery happens after i deadlines were missed. This matrix can be constructed modifying the last row of A_H, and switching the position of K to use the correct state vector (i.e., the one that corresponds to the measurements obtained n steps before).

For $i = 1$, i.e., with one deadline miss, we can write

$$\begin{bmatrix} x_{[k+1]} \\ x_{[k]} \\ \dots \\ x_{[k-n+1]} \\ u_{[k+1]} \end{bmatrix} = \underbrace{\begin{bmatrix} A_d & 0_{p\times(n\cdot p)} & B_d \\ & I_{n\cdot p} & 0_{(n\cdot p)\times(p+r)} \\ 0_{r\times p} & K \quad 0_{r\times(n-1)\cdot p} & 0_{r\times r} \end{bmatrix}}_{A_{R_1}} \begin{bmatrix} x_{[k]} \\ x_{[k-1]} \\ \dots \\ x_{[k-n]} \\ u_{[k]} \end{bmatrix}. \tag{15}$$

Consistently with our treatise, $A_{R_0} = A_H$. We can then compute the set Σ as $\Sigma = \{A_{R_i} A_M^i \mid i \in \mathbb{Z}^{\geq}, i \leq n\}$ and then compute the upper bound on $\rho(\Sigma)$ using the computed set of matrices[9].

[9] The drawback of constructing the set Σ as shown above is that the size of the matrices in the set grows linearly with the number of deadline misses. It is possible to construct a compact representation that uses as state vector $\tilde{x}_{[k]} = [x_{[k]}^{\mathrm{T}}, u_{[k]}^{\mathrm{T}}]^{\mathrm{T}}$ but writes the evolution of the system directly as the relation between $\tilde{x}_{[k]}$ and $\tilde{x}_{[k+n+1]}$. This second way of expressing the system dynamics has the disadvantage of hiding misses and hits and only showing the evolution at each hit (still keeping track of what happened at the instants in which the misses occurred). We implemented both versions in our code and checked that the obtained results are the same except for the computational speedup. This remark applies to all the strategies using *skip-next*.

Zero&Queue(1). The behaviour of the combination of the zero policy and the queue(1) strategy vary depending on the possibility of the queued job to complete before the deadline or not. We are going to make the additional hypothesis that the worst-case response time for a job is less than $n\,\pi$, where n is the maximum number of consecutive deadline misses. We can start from the set Σ used for the Zero&Skip-Next combination and add to the set all the matrices $A_H A_M^i$, that take into account the possibility that the queued job completed before its deadline. We should also include in the set Σ the matrices A_{R_i} alone, as it could happen that a queued job doesn't terminate in the period it was started in. We then obtain $\Sigma = \{A_H A_M^i, A_{R_i}, A_{R_i} A_M^i \mid i \in \mathbb{Z}^{\geq}, i \leq n\}$, and we can use the set to compute the upper bound on the joint spectral radius $\rho(\Sigma)$.

Hold&Kill. The hold and kill strategy aborts the task but applies the previously computed control signal to the plant. An easy way to analyse this case is to include in the state of the system also the control signal, such that we can determine the switch between two different matrices without having the matrices grow. We denote with $\tilde{x}_{[k]} = [x_{[k]}^{\mathrm{T}}, u_{[k]}^{\mathrm{T}}]^{\mathrm{T}}$. We recall that r is used to represent the number of input variables.

We can write the evolution of the system when a deadline is hit as

$$\begin{bmatrix} x_{[k+1]} \\ u_{[k+1]} \end{bmatrix} = \underbrace{\begin{bmatrix} A_d & B_d \\ K & 0_{r\times r} \end{bmatrix}}_{A_H} \begin{bmatrix} x_{[k]} \\ u_{[k]} \end{bmatrix}, \tag{16}$$

meaning that the computation (the third row of the A_H matrix) is completed and the new control variable is updated with the information from the plant. When a deadline is missed, we compute the system evolution as

$$\begin{bmatrix} x_{[k+1]} \\ u_{[k+1]} \end{bmatrix} = \underbrace{\begin{bmatrix} A_d & B_d \\ 0_{r\times r} & I_r \end{bmatrix}}_{A_M} \begin{bmatrix} x_{[k]} \\ u_{[k]} \end{bmatrix}, \tag{17}$$

encoding the hold as the identity matrix that multiplies the old control value for the equation that determines the evolution of $u_{[k]}$. As done for the zero&kill alternative, if we assume there can be a maximum of n consecutive deadline misses, we can then compute all the matrices in $\Sigma = \{A_H A_M^i \mid i \in \mathbb{Z}^{\geq}, i \leq n\}$ and then compute the upper bound on $\rho(\Sigma)$.

Hold&Skip-Next. In order to analyse the combination of hold and skip-next we need to augment the state vector as we did for the zero&skip-next handling strategy. Also in this case, we use our newly defined state vector $\tilde{x}_{[k]} = [x_{[k]}^{\mathrm{T}}, x_{[k-1]}^{\mathrm{T}}, \ldots, x_{[k-n]}^{\mathrm{T}}, u_{[k]}^{\mathrm{T}}]^{\mathrm{T}}$. We obtain the following expression for the closed-loop system when we hit a deadline,

$$\begin{bmatrix} x_{[k+1]} \\ x_{[k]} \\ \cdots \\ x_{[k-n+1]} \\ u_{[k+1]} \end{bmatrix} = \underbrace{\begin{bmatrix} A_d & 0_{p\times(n\cdot p)} & B_d \\ & I_{n\cdot p} & 0_{(n\cdot p)\times(p+r)} \\ K & 0_{r\times(n\cdot p)} & 0_{r\times r} \end{bmatrix}}_{A_H} \begin{bmatrix} x_{[k]} \\ x_{[k-1]} \\ \cdots \\ x_{[k-n]} \\ u_{[k]} \end{bmatrix}. \tag{18}$$

When we miss a deadline we use the old control value, introducing an identity matrix in the last column and last row of the closed-loop state matrix to indicate that the previous control

signal is saved,

$$\begin{bmatrix} x_{[k+1]} \\ x_{[k]} \\ \dots \\ x_{[k-n+1]} \\ u_{[k+1]} \end{bmatrix} = \underbrace{\begin{bmatrix} A_d & 0_{p\times(n\cdot p)} & B_d \\ & I_{n\cdot p} & 0_{(n\cdot p)\times(p+r)} \\ 0_{r\times(n+1)\cdot p} & & I_r \end{bmatrix}}_{A_M} \begin{bmatrix} x_{[k]} \\ x_{[k-1]} \\ \dots \\ x_{[k-n]} \\ u_{[k]} \end{bmatrix}. \tag{19}$$

Finally, we should define the behaviour of system in the case of a recovery hit, as done for the zero&skip-next strategy, but including the dynamic evolution of the control signal that has been added to $\tilde{x}$. For one deadline miss we obtain A_{R_1} as

$$\begin{bmatrix} x_{[k+1]} \\ x_{[k]} \\ \dots \\ x_{[k-n+1]} \\ u_{[k+1]} \end{bmatrix} = \underbrace{\begin{bmatrix} A_d & 0_{p\times(n\cdot p)} & B_d \\ & I_{n\cdot p} & 0_{(n\cdot p)\times(p+r)} \\ 0_{r\times p} & K \quad 0_{r\times(n-1)\cdot p} & 0_{r\times r} \end{bmatrix}}_{A_{R_1}} \begin{bmatrix} x_{[k]} \\ x_{[k-1]} \\ \dots \\ x_{[k-n]} \\ u_{[k]} \end{bmatrix}, \tag{20}$$

and the following matrices are obtained by moving the position of the term K in the state evolution matrix to reflect how old is the sensed data that is being used for the computation of the control signal.

Again, $A_{R_0} = A_H$, and we can define the set Σ as $\Sigma = \{A_{R_i} A_M^i \mid i \in \mathbb{Z}^{\geq}, i \leq n\}$. With this, we can compute the upper bound on $\rho(\Sigma)$.

Hold&Queue(1). To analyse the hold&queue(1) strategy, we follow the same principles used for the zero&queue(1) strategy. We start from the hold&skip-next matrices and determine $\Sigma = \{A_H A_M^i, A_{R_i}, A_{R_i} A_M^i \mid i \in \mathbb{Z}^{\geq}, i \leq n\}$.

5 Experimental Validation

In this section we present a few examples of how the analysis presented in Section 4 can be applied to determine the robustness to deadline misses of control system implementations. In particular, we first present some results obtained with an unstable second-order system, which could be used to approximate unstable plants such as a segway that has to be stabilised about the top position. Then, we verify the stability of a permanent magnet synchronous motor for an automotive electric steering application.

Unstable Second-Order System. We analyse the following continuous-time linear time-invariant open-loop system,

$$\dot{x}(t) = \underbrace{\begin{bmatrix} 10 & 0 \\ -2 & -1 \end{bmatrix}}_{A_c} x(t) + \underbrace{\begin{bmatrix} 5 & 1 \\ 4 & 10 \end{bmatrix}}_{B_c} u(t), \tag{21}$$

where both the state and the input vector are composed of two variables. The expression above is the equivalent of Equation (2). Since the A_c matrix is a lower triangular matrix, one can immediately see that the poles of the system are 10 and -1. Since one pole is in the right half plane, the system has one unstable mode and there is a need for control to stabilise the system.

An optimal linear quadratic regulator [26] is designed for this system, assuming that the controller execution is instantaneous and there is no one-step delay actuation, obtaining

$$K = \begin{bmatrix} -4.7393 & 0.2430 \\ 0.2277 & -0.8620 \end{bmatrix}. \tag{22}$$

First, we check the stability of the closed loop system when the controller is executed with one step delay. We select a sampling period of $10\,ms$ and discretise the system obtaining

$$x_{[k+1]} = \underbrace{\begin{bmatrix} 1.1053 & 0.0000 \\ -0.0209 & 0.9900 \end{bmatrix}}_{A_d} x_{[k]} + \underbrace{\begin{bmatrix} 0.0526 & 0.0105 \\ 0.0393 & 0.0994 \end{bmatrix}}_{B_d} u_{[k]}. \tag{23}$$

This corresponds to Equation (3). The matrix A_d is also lower triangular, which means that the poles of the open-loop system are the numbers indicated in the main diagonal. Since one of them is outside the unit circle, the discretised version of the continuous-time system is (unsurprisingly) also unstable and needs control. The poles of the closed-loop system corresponding to the execution of the LET controller every $10\,ms$ are $\{0.8911, 0.8141, 0.3013, 0.0888\}$ and they are all inside the unit circle, meaning that the system is stabilised by the LET controller K from Equation (22), and our control design is a good choice.

Our research question is how many deadlines can we miss when we execute the controller with all the possible deadline miss handling strategies. Table 1 summarises the results we obtain for the analysis.

With the zero strategy, irrespective of the choice of how to handle the job that misses the deadline (kill, skip-next, or queue), the system is robust to missing one deadline (the upper bound on the joint spectral radius is less than 1 for one deadline miss). A subsequent miss is not tolerated, and the closed-loop system becomes provably unstable (the lower bound on the joint spectral radius is above 1). We now look at what happens when the control signal is kept constant in case of misses, i.e., when we hold. Hold&kill ensures that we could miss five deadlines in a row without the emergence of unstable behaviour. However, if a sixth deadline is missed, the system could become unstable. In this case, in fact, the upper-bound on the joint spectral radius exceeds 1. The lower bound, however, is below 1. This means that there is no complete certainty that the system is unstable, but there is a high risk. The true value (for which we have certainty) lies in between the the two bounds. If we continue our analysis, the situation where the true values is around 1 and uncertain persists up to 7 deadline misses. When we introduce the possibility of missing 8 consecutive deadlines, both the lower-bound and the upper-bound are above 1, meaning that the system is provably unstable for some sequences. Notice that this does not mean that the instability is found when we repeat the sequence with 8 consecutive misses followed by a hit. It could happen that the unstable sequence is a combination of a number of deadline misses up to 8 followed by a different number, followed by a number of deadline hits, and so forth. In fact, in our investigation we have encountered cases in which the closed-loop system was stable in case of the repetition of the sequence with n misses followed by a hit, but unstable with a number of consecutive misses up to n. For any practical application, terminating the investigation when the upper-bound crosses 1 ensures a safety margin and guarantees the correct system operation.

When hold is paired with skip-next the system tolerates 2 misses. When queue(1) is uses, on the contrary, the system does not tolerate even a single miss. As a final remark, notice that while the number of tolerated deadline misses is higher for hold&kill, when a task is killed it is difficult to guarantee – with real-time analysis – that the subsequent job will

Table 1 Stability Results for the Unstable Second-Order System.

Misses		Stability	Lower Bound	Upper Bound
Zero&Kill	1	✓	0.961037	0.961975
	2	✗	1.071911	1.071915
Zero&Skip-Next	1	✓	0.914298	0.920769
	2	✗	1.059819	1.059822
Zero&Queue(1)	1	✓	0.961037	0.964287
	2	✗	1.071911	1.071915
Hold&Kill	1	✓	0.891089	0.891090
	2	✓	0.891089	0.891090
	3	✓	0.891089	0.891098
	4	✓	0.891089	0.891251
	5	✓	0.891089	0.935272
	6	✗	0.891089	1.004593
	7	✗	0.961344	1.083038
	8	✗	1.065537	1.172249
Hold&Skip-Next	1	✓	0.891089	0.891090
	2	✓	0.914556	0.944458
	3	✗	1.076507	1.091171
Hold&Queue(1)	1	✗	1.347066	1.370827

not miss its deadline (especially due to locality effects). On the contrary, in general, when skip-next is applied, it is easier to guarantee the termination of in the consecutive periods. This is true unless the deadline misses are caused by a bug in the control task itself, in which case the control task may never terminate.

Electric Steering Application. Here we verify the stability of a permanent magnet synchronous motor for an automotive electric steering application in the presence of deadline misses. We first present a standard model for the motor and a proportional and integral (PI) controller for setpoint tracking, written in the state-feedback form.

When modeling the motor, our system state is $x(t) = [i_d(t), i_q(t)]^{\mathrm{T}}$ where $i_d(t)$ and $i_q(t)$ represent respectively the currents in the d and q coordinates over time. The aim of our control design is to track arbitrary reference values for the currents. The control signal is $u(t) = [u_d(t), u_q(t)]^{\mathrm{T}}$, where $u_d(t)$ and $u_q(t)$ represent respectively the voltages applied to the motor in the d and q coordinate system (subject to an affine transformation). The model of the motor can be written as

$$\dot{x}(t) = \begin{bmatrix} -R/L_d & L_q\,\omega_{el}/L_d \\ -L_d\,\omega_{el}/L_q & -R/L_q \end{bmatrix} x(t) + \begin{bmatrix} 1/L_d & 0 \\ 0 & 1/L_q \end{bmatrix} u(t). \tag{24}$$

Here, L_d [ht] and L_q [ht] denote respectively the inductance in the d and q direction, R [Ohm] is the winding resistance, and ω_{el} [rad/s] is the frequency of the rotor-induced voltage (assumed as constant). We used the parameters of our motor[10] and discretised the plant

[10] The constants used for the calculations are: $R = 0.025$ [Ohm], $\omega_{el} = 6283.2$ [rad/s], $L_d = 0.0001$ [ht], $L_q = 0.00012$ [ht].

using Tustin's method[11], and a sampling period of 10μs, obtaining

$$x_{[k+1]} = \underbrace{\begin{bmatrix} 0.996 & 0.075 \\ -0.052 & 0.996 \end{bmatrix}}_{A_{d,\text{base}}} x_{[k]} + \underbrace{\begin{bmatrix} 0.100 & 0.003 \\ -0.003 & 0.083 \end{bmatrix}}_{B_{d,\text{base}}} u_{[k]}. \tag{25}$$

Notice that the eigenvalues of $A_{d,\text{base}}$ are $0.9957 \pm 0.0626i$ and their absolute value is 0.9977, meaning that the open-loop system is stable (even without control). Control is here added to constrain the behaviour of the system and make sure it tracks current setpoints without errors.

To achieve zero steady-state error, we would like to design our controller in the PI form, using a term that is proportional to the error between the actual current vector and the setpoint vector and a term that is proportional to the integral of the error. We then need to augment the state vector $x_{[k]}$ and keep track of the error at the current time step and at the previous time step. We also need to add to the input vector the setpoints for the currents in the D and Q directions. This allows us to write our PI controller in the state-feedback form.

After this transformation, we denote the new system input as $v_{[k]} = [u_{[k]}^{\mathrm{T}}, w_{[k]}^{\mathrm{T}}]^{\mathrm{T}}$ where $w_{[k]}$ denotes a vector that contains the desired values for the currents i_d and i_q at time k. We also define the new system state as $s_{[k]} = [x_{[k]}^{\mathrm{T}}, e_{[k]}^{\mathrm{T}}, e_{[k-1]}^{\mathrm{T}}]^{\mathrm{T}}$, where $e_{[k]}$ is the error at time k, i.e., $e_{[k]} = w_{[k]} - x_{[k]}$. We therefore model the system as

$$s_{[k+1]} = \underbrace{\begin{bmatrix} A_{d,\text{base}} & 0_{2\times2} & 0_{2\times2} \\ -I_2 & 0_{2\times2} & 0_{2\times2} \\ 0_{2\times2} & I_2 & 0_{2\times2} \end{bmatrix}}_{A_d} s_{[k]} + \underbrace{\begin{bmatrix} B_{d,\text{base}} & 0_{2\times2} \\ 0_{2\times2} & I_2 \\ 0_{2\times2} & 0_{2\times2} \end{bmatrix}}_{B_d} v_{[k]}. \tag{26}$$

We design our PI controller as

$$K = \begin{bmatrix} & \overbrace{\begin{bmatrix} 5 & 0 \\ 1 & 7 \end{bmatrix}}^{K_1} & \overbrace{\begin{bmatrix} -4 & 0 \\ -3 & 7 \end{bmatrix}}^{K_2} \\ 0_{2\times2} & 0_{2\times2} & 0_{2\times2} \\ 0_{2\times2} & 0_{2\times2} & 0_{2\times2} \end{bmatrix}. \tag{27}$$

We can test that in absence of deadline misses, when the controller is implemented with LET (i.e., with one-step delay), the system preserves stability. We now move on to investigate the stability property of the system when some deadlines are missed. Table 2 presents a summary of the results we obtained.

The zero&kill strategy presents a special property. Using this strategy, the closed-loop system is always going to be stable, regardless of the number of deadlines that are missed. In fact, using the joint spectral radius, it is possible to prove that the system that switches between A_H and A_M is always stable, when the matrices are the ones identified in Section 4 for zero&kill. This special property comes from the fact that the open-loop system is stable and control is applied only using fresh measurements (assuming that the kill procedure is able to rollback the system to a clean state). The fact that A_M and A_H are stable individually is not enough to guarantee stability, but all of their combinations prove to be contractions, making the switching system stable as well. Notice that this is not generalisable, but only

[11] While Tustin's method introduces frequency distortion, the method is what is currently applied in our industrial case study. Similar results can be obtained with the exact matrix exponential, changing the discretisation command parameters in the Matlab code.

Table 2 Stability Results for the Electric Steering Application.

Misses		Stability	Lower Bound	Upper Bound
Zero&Kill	∞	✓	0.997713	0.997713
Zero&Skip-Next	1	✓	0.892575	0.892575
	2	✓	0.892575	0.892575
	3	✓	0.892575	0.892575
	4	✓	0.892575	0.892575
	5	✓	0.892575	0.892575
	6	✓	0.892575	0.892575
	7	✓	0.892575	0.892575
	8	✓	0.900922	0.900922
	9	✓	0.912902	0.912902
	10	✓	0.938565	0.938565
	11	✓	0.940823	0.940823
	12	✓	0.942610	0.942610
	13	✓	0.951092	0.951092
	14	✓	0.962846	0.962846
	15	✓	0.973776	0.973776
	16	✓	0.983954	0.983954
	17	✓	0.993436	0.993436
	18	✗	1.002273	1.002273
Zero&Queue(1)	1	✓	0.925966	0.925966
	2	✗	1.001620	1.001620
Hold&Kill	1	✓	0.892575	0.892575
	2	✓	0.938332	0.938332
	3	✗	1.073542	1.073542
Hold&Skip-Next	1	✓	0.968574	0.968574
	2	✗	1.107390	1.107390
Hold&Queue(1)	1	✗	1.423968	1.423968

due to the properties of the particular system we are controlling – in fact, this is not true for the the second-order system example above. For this specific system, this tells us that implementing a clean rollback is very beneficial, and goes a long way to ensure fault tolerance. However, it is not always possible to implement a clean rollback with the given hardware and software setup.

When zero is paired with skip-next, old measurements of the state are used when a recovery hit happens. This means that the system can be unstable even though the behaviour is similar to the zero&kill strategy one. In fact, differently from zero&kill, the system is not able to tolerate an infinite number of misses. We then ask how many many consecutive deadline misses the system experience without violating the stability property. As reported in Table 2, both the lower-bound and the upper-bound on the joint spectral radius are less than 1 for a system that experiences up to 17 consecutive misses. However, with 18 consecutive misses the closed-loop system becomes provably unstable (lower-bound above 1).

This result means that if the controller can miss 18 deadlines in a row, then the delay introduced between the sensing and the actuation is harmful for the system and can cause instability. In this case, there is at least one sequence of deadline misses (which satisfies the

condition that there cannot be more than 18 consecutive misses) that leads to instability. In this case, it is certain that the closed-loop switching system is unstable. Notice that the harmful sequence does not necessarily have to be a repetition of 18 misses and 1 hit, but can be a combination of different terms (for example it could happen that due to the time constant of the system missing 16 deadlines, hitting 2 of them, and then missing 18 would cause instability). For this particular case, however, it is simple enough to check that the spectral radius of the closed-loop matrix with 18 misses and 1 hit is above 1 and this immediately means that the sequence of 18 misses and 1 hit destabilises the system (although it might not be the only one).

If we pair zero with queue(1), the results differ dramatically. The number of matrices in the set Σ used to compute the joint spectral radius increases, and there is a combination of events (specific number of misses, recovery, and recovery with immediate information) that can lead to instability even when just 2 consecutive deadlines are missed. This immediately tells us that using the queue(1) strategy is a bad idea for this system and should be avoided.

When the control signal is held, the kill strategy guarantees stability for 2 deadline misses, while a third potential miss makes the system provably unstable. The skip-next strategy, paired with hold, can tolerate one miss, but a second one makes the system unstable. The queue(1) strategy cannot even tolerate a single miss.

From this experimental campaign, we can conclude that for this system kill is working better than any other system-level strategy, and zero works better than hold in terms of guaranteeing the system stability. We can also conclude that while queuing a task when a previous one is executing could improve the control performance, the risk of harming the system is much higher. We advocate that performing the analysis presented in this paper can give important runtime information to determine how robust control systems are to temporary faults and problems.

6 Related Work

Studying the non-ideal behaviour that emerges from the implementation and execution of control systems is an important problem for practical applications of control. Control tasks for example may have variable periods and may require to be executed with different rates depending on the operating conditions [4]. Also, late information can affect the system's performance [24, 30, 31, 37], especially for networked systems. These timing effects are usually characterised as independent events with stochastic distributions [13], or using worst case bounds [2, 15]. Control researchers proposed deadline-aware design methods to guarantee stability [5, 29] and improve the control performance [38]. Sinopoli et al. [40] proposed an optimal control design for networked system leveraging the probability of packet losses. Lincoln and Cervin [28] proposed a design tool for optimal controllers in the presence of probabilistic delays, that can be exploited for LET design setting the delay to one period. In many circumstances, the control designer can usually trade off computing time and accuracy [39]. In some cases, an inaccurate and faster solution that can be executed at a faster rate is preferable to an accurate and precise solution that can only be executed at a slower rate [23]. However, it is extremely hard in practice to guarantee that the delays will *never* exceed the control period.

These types of systems have been studied both from the schedulability perspective [43] and using model checking [12]. The performance cost of deadline misses was investigated [34, 44], together with the role of the strategy used to handle the misses [33, 41, 42]. All these papers used the m-K model, starting from the assumption that windows of hits and misses have to

be analysed in order to determine the behaviour of the system. We build on the previous literature to determine how the implementation is going to react to missed deadlines, both in terms of selection of the control signal [29] and in terms of management of the job that misses its deadline [33]. There are some important differences between [33] and the contribution of this paper: [33] presents a control design technique to guarantee probabilistic robustness to deadline misses, on the contrary we assume that the controller is already designed and executes without any change and we demonstrate an analysis method to guarantee exactly that the controller tolerates (in terms of stability) a maximum number of consecutive misses.

In this work we showed that it may not be necessary to study windows of hits and misses. We argue that before looking at the m-K model representation, which is harder to analyse, one should check the stability of the $\overline{\langle m \rangle}$ model. If the $\overline{\langle m \rangle}$ is stable, there is no need to include complex window-based analysis – it happens quite often that the information needed to study the stability of the closed-loop systems is already contained in the number of consecutive deadline misses.

Ghosh et. al. [16] studied how to design control systems in the presence of faults that cause them to miss at most n deadlines, providing a synthesis method to achieve fault-tolerance – without specifying how the system is going to react to the misses (e.g., kill or skip-next). Here we take a different perspective and want to validate the behaviour of a control system (in terms of stability) when deadlines are missed, including the deadline management strategy from a system and implementation perspective.

7 Conclusion

In this paper we revisited the weakly hard real-time model for control tasks. We formalised the problem of determining the stability of the closed-loop system in the presence of a given number of consecutive misses that the controller task can experience. With the number of consecutive deadline misses, we derived stability criteria for systems where the deadline miss is handled in different ways, both from the perspective of the control signal applied (either zeroing or holding the previous value of the control signal) and with respect to the management of the task that misses the deadline (that can be either killed or allowed to continue in the subsequent control period). We solved this problem using a mathematical tool called joint spectral radius, for the computation of which open-source toolboxes are available. We applied our analysis to two different examples: an unstable system and an industrial application for electric steering. In both cases, we showed the limitations of the controller implementations that miss a given number of deadlines.

In the future, we plan to look at applying the joint spectral radius analysis to systems specified using the m-K model. This is particularly challenging, because there is no direct mapping between the potential sequences of hits and misses and a set of matrices. Furthermore, we want to investigate the performance of controllers that experience deadline misses.

References

1 Leonie Ahrendts, Sophie Quinton, Thomas Boroske, and Rolf Ernst. Verifying Weakly-Hard Real-Time Properties of Traffic Streams in Switched Networks. In *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, volume 106 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:22. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.ECRTS.2018.15.

2 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.

3 G. Bernat, A. Burns, and A. Liamosí. Weakly hard real-time systems. *IEEE Transactions on Computers*, 50(4):308–321, April 2001. doi:10.1109/12.919277.

4 Alessandro Biondi, Marco Di Natale, Giorgio C. Buttazzo, and Paolo Pazzaglia. Selecting the transition speeds of engine control tasks to optimize the performance. *ACM Transaction on Cyber-Physical Systems*, 2(1):1:1–1:26, January 2018. doi:10.1145/3127022.

5 R. Blind and F. Allgöwer. Towards networked control systems with guaranteed stability: Using weakly hard real-time constraints to model the loss process. In *54th IEEE Conference on Decision and Control (CDC)*, pages 7510–7515, December 2015. doi:10.1109/CDC.2015.7403405.

6 V. Blondel and Y. Nesterov. Computationally efficient approximations of the joint spectral radius. *SIAM Journal on Matrix Analysis and Applications*, 27(1):256–272, 2005. doi:10.1137/040607009.

7 Vincent Blondel, Yurii Nesterov, and Jacques Theys. On the accuracy of the ellipsoid norm approximation of the joint spectral radius. *Linear Algebra and its Applications*, 394:91–107, 2005. doi:10.1016/j.laa.2004.06.024.

8 Vincent Blondel and John N. Tsitsiklis. The boundedness of all products of a pair of matrices is undecidable. *Systems & Control Letters*, 41(2):135–140, 2000. doi:10.1016/S0167-6911(00)00049-9.

9 Tobias Bund and Frank Slomka. Controller/platform co-design of networked control systems based on density functions. In *Proceedings of the 4th ACM SIGBED International Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems*, CyPhy '14, pages 11–14. ACM, 2014. doi:10.1145/2593458.2593467.

10 Tobias Bund and Frank Slomka. Worst-case performance validation of safety-critical control systems with dropped samples. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS '15, pages 319–326. ACM, 2015. doi:10.1145/2834848.2834860.

11 Lane Desborough. Increasing customer value of industrial control performance monitoring-honeywell's experience. *Preprints of CPC*, pages 153–186, 2001.

12 G. Frehse, A. Hamann, S. Quinton, and M. Woehrle. Formal analysis of timing effects on closed-loop properties of control software. In *2014 IEEE Real-Time Systems Symposium*, pages 53–62, December 2014. doi:10.1109/RTSS.2014.28.

13 Maximilian Gaukler, Andreas Michalka, Peter Ulbrich, and Tobias Klaus. A new perspective on quality evaluation for control systems with stochastic timing. In *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (Part of CPS Week)*, HSCC, pages 91–100. ACM, 2018. doi:10.1145/3178126.3178134.

14 Maximilian Gaukler, Tim Rheinfels, Peter Ulbrich, and Günter Roppenecker. Convergence rate abstractions for weakly-hard real-time control. *arXiv preprint arXiv:1912.09871*, 2019.

15 Maximilian Gaukler and Peter Ulbrich. Worst-case analysis of digital control loops with uncertain input/output timing. In Goran Frehse and Matthias Althoff, editors, *ARCH19. 6th International Workshop on Applied Verification of Continuous and Hybrid Systems*, volume 61 of *EPiC Series in Computing*, pages 183–200. EasyChair, 2019. doi:10.29007/c4zl.

16 Saurav Kumar Ghosh, Soumyajit Dey, Dip Goswami, Daniel Mueller-Gritschneder, and Samarjit Chakraborty. Design and validation of fault-tolerant embedded controllers. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1283–1288, 2018. doi:10.23919/DATE.2018.8342212.

17 N. Guglielmi, F. Wirth, and M. Zennaro. Complex polytope extremality results for families of matrices. *SIAM Journal on Matrix Analysis and Applications*, 27(3):721–743, 2005. doi:10.1137/040606818.

18 M. Hamdaoui and P. 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.

19 Z. A. H. Hammadeh, R. Ernst, S. Quinton, R. Henia, and L. Rioux. Bounding deadline misses in weakly-hard real-time systems with task dependencies. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 584–589, March 2017. doi:10.23919/DATE.2017.7927054.

20 Zain A. H. Hammadeh, Sophie Quinton, Marco Panunzio, Rafik Henia, Laurent Rioux, and Rolf Ernst. Budgeting Under-Specified Tasks for Weakly-Hard Real-Time Systems. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:22. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.ECRTS.2017.17.

21 R. Jungers. *The Joint Spectral Radius: Theory and Applications*. Lecture Notes in Control and Information Sciences. Springer Berlin Heidelberg, 2009.

22 Raphael Jungers. JSR Toolbox. https://www.mathworks.com/matlabcentral/fileexchange/33202-the-jsr-toolbox, accessed October 23, 2019.

23 E. C. Kerrigan, G. A. Constantinides, A. Suardi, A. Picciau, and B. Khusainov. Computer architectures to close the loop in real-time optimization. In *2015 54th IEEE Conference on Decision and Control (CDC)*, pages 4597–4611, December 2015. doi:10.1109/CDC.2015.7402937.

24 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.

25 Huibert Kwakernaak. *Linear Optimal Control Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1972.

26 W.S. Levine. *The Control Handbook*. Electrical Engineering Handbook. Taylor & Francis, 1996.

27 D. Liberzon. *Switching in Systems and Control*. Systems & Control: Foundations & Applications. Birkhäuser Boston, 2003.

28 B. Lincoln and A. Cervin. JITTERBUG: a tool for analysis of real-time control performance. In *41st IEEE Conference on Decision and Control*, volume 2, pages 1319–1324, December 2002. doi:10.1109/CDC.2002.1184698.

29 S. Linsenmayer and F. Allgower. Stabilization of networked control systems with weakly hard real-time dropout description. In *IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 4765–4770, December 2017. doi:10.1109/CDC.2017.8264364.

30 L. Palopoli, L. Abeni, G. Buttazzo, F. Conticelli, and M. Di Natale. Real-time control system analysis: an integrated approach. In *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*, pages 131–140, 2000. doi:10.1109/REAL.2000.896003.

31 P. Park, S. Coleri Ergen, C. Fischione, C. Lu, and K. H. Johansson. Wireless network design for control systems: A survey. *IEEE Communications Surveys Tutorials*, 20(2):978–1013, 2018. doi:10.1109/COMST.2017.2780114.

32 Pablo A. Parrilo and Ali Jadbabaie. Approximation of the joint spectral radius using sum of squares. *Linear Algebra and its Applications*, 428(10):2385–2402, 2008. doi:10.1016/j.laa.2007.12.027.

33 Paolo Pazzaglia, Claudio Mandrioli, Martina Maggio, and Anton Cervin. DMAC: Deadline-Miss-Aware Control. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:24, 2019. doi:10.4230/LIPIcs.ECRTS.2019.1.

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, 2018. doi:10.4230/LIPIcs.ECRTS.2018.10.

35 Gang Quan and Xiaobo Hu. Enhanced fixed-priority scheduling with (m,k)-firm guarantee. In *Proceedings of the 21st IEEE Conference on Real-time Systems Symposium*, RTSS, pages 79–88. IEEE Computer Society, 2000.

36 Gian–Carlo Rota and W. Gilbert Strang. A note on the joint spectral radius. *Indagationes Mathematicae*, 63:379–381, 1960. doi:10.1016/S1385-7258(60)50046-1.

37 Abusayeed Saifullah, Chengjie Wu, Paras Babu Tiwari, You Xu, Yong Fu, Chenyang Lu, and Yixin Chen. Near optimal rate selection for wireless control systems. *ACM Transactions on Embedded Computing Systems*, 13(4s):128:1–128:25, April 2014. doi:10.1145/2584652.

38 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.

39 Amir Shahzad, Eric C. Kerrigan, and George A. Constantinides. A stable and efficient method for solving a convex quadratic program with application to optimal control. *SIAM Journal on Optimization*, 22(4):1369–1393, 2012. doi:10.1137/11082960X.

40 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.

41 D. Soudbakhsh, L. T. X. Phan, A. M. Annaswamy, and O. Sokolsky. Co-design of arbitrated network control systems with overrun strategies. *IEEE Transactions on Control of Network Systems*, 5(1):128–141, March 2018. doi:10.1109/TCNS.2016.2583064.

42 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. ACM, 2013. doi:10.1145/2502524.2502542.

43 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*, 16(5s):171:1–171:19, September 2017. doi:10.1145/3126497.

44 E. P. van Horssen, A. R. B. Behrouzian, D. Goswami, D. Antunes, T. Basten, and W. P. M. H. Heemels. Performance analysis and controller improvement for linear systems with (m, k)-firm data losses. In *2016 European Control Conference (ECC)*, pages 2571–2577, June 2016. doi:10.1109/ECC.2016.7810677.

45 Guillaume Vankeerberghen, Julien Hendrickx, and Raphaël M. Jungers. JSR: a toolbox to compute the joint spectral radius. In *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control*, HSCC, pages 151–156. ACM, 2014. doi:10.1145/2562059.2562124.

46 Dirk Ziegenbein and Arne Hamann. Timing-aware control software design for automotive systems. In *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pages 56:1–56:6, 2015. doi:10.1145/2744769.2747947.

Abstract Response-Time Analysis: A Formal Foundation for the Busy-Window Principle

Sergey Bozhko
Max Planck Institute for Software Systems, Kaiserslautern, Germany
sbozhko@mpi-sws.org

Björn B. Brandenburg
Max Planck Institute for Software Systems, Kaiserslautern, Germany
bbb@mpi-sws.org

Abstract

This paper introduces the first general and rigorous formalization of the classic busy-window principle for uniprocessors. The essence of the principle is identified as a minimal set of generic, high-level hypotheses that allow for a unified and general *abstract response-time analysis*, which is independent of specific scheduling policies, workload models, and preemption policy details. From this abstract core, the paper shows how to obtain concrete analysis instantiations for specific uniprocessor schedulers via a sequence of refinement steps, and provides formally verified response-time bounds for eight common schedulers and workloads, including the widely used fixed-priority (FP) and earliest-deadline first (EDF) scheduling policies in the context of fully, limited-, and non-preemptive sporadic tasks. All definitions and proofs in this paper have been mechanized and verified with the Coq proof assistant, and in fact form the common core and foundation for verified response-time analyses in the Prosa open-source framework for formally proven schedulability analyses.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Software and its engineering → Scheduling; Theory of computation → Scheduling algorithms

Keywords and phrases hard real-time systems, response-time analysis, uniprocessor, busy window, fixed priority, EDF, verification, Coq, Prosa, preemptive, non-preemptive, limited-preemptive

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.22

Related Version Extended paper with full proofs: `https://mpi-sws.org/tr/2020-003.pdf`.

Supplementary Material ECRTS 2020 Artifact Evaluation approved artifact available at `https://doi.org/10.4230/DARTS.6.1.3`.
The Coq proof, a browsable specification, and artifact evaluation instructions can be found at `https://people.mpi-sws.org/~sbozhko/ECRTS20/AbstractRTA.html`.

Funding This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 803111). Funded in part by Deutsche Forschungsgemeinschaft (DFG) – 391919384.

Acknowledgements We thank the members of the ANR-DFG joint project RT-PROOFS, in particular Maxime Lesourd and Sophie Quinton, for their valuable feedback and suggestions.

1 Introduction

The *busy-window principle* is one of the most fundamental and widely known real-time scheduling concepts. The basic idea – to bound a task's *worst-case response time* by analyzing the interval (or "window") during which the processor remains continuously "busy" executing a given task or interfering workload – has been applied in scores of papers on many different scheduling policies, system models, and workload types. In practical terms, the busy-window principle, in the form of *response-time analysis* (RTA) [3, 12, 28, 30, 34], provides the theoretical underpinnings of popular commercial analysis tools (e.g., [26]).

However, despite its undeniable significance, to date no unifying theoretical framework has been presented that explains, and rigorously justifies, why the busy-window principle is applicable in so many diverse settings. Rather, the general idea has become part of the real-time folklore, spread across many papers, where it is frequently re-developed "from scratch" using ad-hoc notation and problem-specific definitions. As previously noted by Bril et al. [8], this results in papers differing in subtle but critical ways, especially when proofs are rooted more in intuition and implicit assumptions than formal derivation.

To place this central element of real-time scheduling theory on firm footings, and to enable a systematic development of verified RTAs in the Prosa [10, 35] open-source framework for formally proven schedulability analyses, we present a rigorous, formally verified proof of the busy-window principle using the Coq proof assistant [42]. Our proof is general and proceeds from first principles, without the use of a "critical instant" argument. At its core is a new *abstract RTA* that is independent of specific scheduling policies, workload models, and preemption details. The existence of this abstract RTA, which relies only on a minimal set of generic, high-level assumptions, explains why the busy-window principle is so widely applicable and clearly identifies its mathematical essence.

From this abstract core, our proof proceeds via a sequence of refinement steps to obtain concrete RTA instantiations for specific schedulers. As practical examples, we provide *formally verified* RTAs for *eight* scheduler and workload combinations: *earliest-deadline first* scheduling of *fully preemptive* tasks (EDF), *fully non-preemptive* tasks (NP-EDF), *segmented limited-preemptive* tasks (LP-EDF), and tasks with *floating non-preemptive sections* (EDF-NPS), as well as *fixed-priority* scheduling of the same task models (denoted FP, NP-FP, LP-FP, and FP-NPS, respectively), all for *arbitrary arrival curves* and *arbitrary deadlines*. Of these, three RTAs are completely novel in that they were not yet derived in prior work (NP-EDF, LP-EDF, and EDF-NPS),[1] and all but one (fully preemptive FP [14]) are formally verified here for the first time. In particular, we provide the first verification of Davis et al.'s [12] revised analysis of the CAN protocol (an instance of NP-FP scheduling).

At a more technical level, this paper makes a contribution in the area of mechanized proofs for real-time systems. Despite many advances in proof assistants in recent years, mechanized proofs still suffer from a key problem: even a conceptually simple change in the underlying model can render a mechanized proof thoroughly inapplicable, and thereby easily induce dozens of person-hours of "proof maintenance." This problem is even more relevant in the area of real-time systems, where a lot of proofs rely on similar ideas but apply to (slightly) different models. It would thus be a Sisyphean task to mechanize the analysis of many such related models without a general underlying theory independent of minor model variations. Abstract RTA provides such a foundational theory for the busy-window principle.

Related Work. The origins of the busy-window principle date back more than three decades; a good account of its history is provided by Audsley et al. [2] and Sha et al. [38].

Liu and Layland [31], in their classic analysis of FP scheduling with *rate-monotonic* (RM) priorities, established that a periodic task exhibits its worst-case response time if all higher-priority tasks release jobs simultaneously and at their maximum possible rate, which they called the *critical instant* [31]. More than a decade later, a closer examination of the critical instant led to the independent discovery of exact RTAs for FP scheduling (with any priority order) based on the busy-window principle by several groups of authors [3, 28, 29, 30, 34].

[1] Note that our claim is specific to RTAs – there are prior *schedulability analyses* of NP-EDF [27], LP-EDF [4] and EDF-NPS [5, 40], which however do not yield nontrivial response-time bounds (i.e. they provide only a "schedulable"/"not schedulable" answer). We focus in this paper exclusively on RTAs.

Lehoczky [30] in particular formulated the concept of a "level-i busy period" (a.k.a. busy window), which he defined as "a time interval $[a, b]$ within which jobs of priority i or higher are processed throughout $[a, b]$ but no jobs of level i or higher are processed in $(a - \varepsilon, a)$ or $(b, b + \varepsilon)$ for sufficiently small $\varepsilon > 0$" [30]. Based on this definition, he provided an RTA for tasks with arbitrary deadlines (i.e., relative deadlines independent of a task's period), a case which Joseph and Pandya's earlier RTA [28] failed to account for. Also relying on the busy-window principle (but without giving a precise definition), Audsley et al. [3] and Tindell et al. [43] improved the approach to support release jitter and other workload extensions.

From these beginnings, the busy-window concept spread far and wide (e.g., see [7, 8, 12, 13, 21, 22, 24, 25, 26, 37, 41, 44, 46], to list a few examples across time), and can still be encountered on a regular basis in recent work (e.g., [23, 36, 47]). However, while the busy-window *idea* spread, Lehoczky's *definition* [30] did not. Rather, the concept was reused and adapted in many ways, to the effect that remarkably few papers agree on an *exact* definition of a busy window, or even provide one at all [8].

As a result of this evolution, papers tend to use diverging definitions, notations, and proof strategies, so that it may seem that there are a lot of different methods to obtain a response-time bound, when in fact most papers follow essentially a common argument. Time and again, papers reason about a critical instant or otherwise construct a worst-case execution scenario, infer a recurrence, argue that the result of a fixed-point search implies a response-time bound, define a search space, etc. Often this is done in analogy with earlier results and not supported by rigorous proof, which can all too easily lead to misconceptions and flaws – and unfortunately has done so more than once (e.g., [10, 11, 12, 33]).

Another byproduct of the state of the art's paper-by-paper approach is that it obscures commonalities within superficially different proofs. Case in point, EDF and FP historically have been analyzed using quite different and policy-specific terminology. Indeed, the existing RTAs for FP [3, 8, 28, 30, 43, 46] and RTAs for EDF [21, 22, 24, 41] *look* like substantially different analyses. However, as we show in this paper, RTAs for EDF and FP share the same fundamental proof scheme, and can in fact be obtained as instantiations of our abstract RTA.

Bril et al. [8] took an important first step towards rectifying the status quo by introducing the concept of an "active period" as a general, foundational concept that is independent of a particular scheduling algorithm and thus reusable across papers. We argue that this is a vital direction – a new RTA should not start from first principles, but rather build on a well-understood *general* and *formal* foundation that comes with clear and simple proof obligations that justify the application of the busy-window principle.

In contrast to Bril et al.'s "active period" [8], the foundation provided in this paper is backed in full by a formal, mechanized proof checked with the Coq proof assistant. The first successful attempt to mechanize schedulability analysis is due to Wilding [45], who proved optimality of EDF on uniprocessors using the early Nqthm theorem prover. In another early effort more closely related to our work, Dutertre [14] proved the correctness of the classic RTA for preemptive FP scheduling with blocking terms already 20 years ago using the PVS proof system, albeit not for arbitrary arrival curves, nor as part of a general, reusable framework. In the same work, Dutertre described and verified the behavior of the priority-ceiling protocol [39]. Much more recently, Zhang et al. [48] formally specified the priority-inheritance protocol [39] using Isabelle/HOL, and proved a blocking bound for the protocol.

We build in this paper on Prosa [35], a Coq-based framework for real-time scheduling theory that emphasizes readability of the specification, which was introduced by Cerqueira et al. [10] in 2016. In 2018, Fradet et al. [15] presented a result (also using Prosa/Coq) that in principle could be used to obtain RTAs for some of the scheduling policies considered

herein, but this possibility was never pursued. In a more applied direction, Fradet et al. [16] introduced CertiCAN, a Prosa-based tool for the certification of CAN schedulability analysis results. Finally, in particularly impressive recent work, Guo et al. [20] connected Prosa [10, 35] with a real-time extension of CertiKOS [18], thereby obtaining a verified OS kernel with an end-to-end, machine-checked schedulability proof.

Contributions. In summary, this paper advances the state of the art in three ways. First, we introduce the first *general* and *rigorous* formalization of the busy-window principle by means of a unified and general *abstract RTA*, which follows from a small, explicit set of *precise* hypotheses.[2] The proposed abstract RTA is intentionally independent of most practical details (such as specific scheduling policies, how and when tasks can be preempted, arrival models, whether they share resources or self-suspend, etc.) and can serve as a common basis for the analysis of a wide range of uniprocessor schedulers and workloads.

Second, *all definitions and proofs in this paper have been mechanized and verified* with the Coq proof assistant [42] in the open-source Prosa framework [10, 35]. Moreover, the work presented in this paper has become the common foundation for mechanized uniprocessor RTAs and enabled support for multiple scheduler and workload combinations in the recently released Prosa version 0.4. Our machine-checked proofs ($\approx$ 9,000 lines of code and comments) are fully documented and freely available for reuse and inspection [35]. As part of the artifact evaluation, we provide an overview of our formal proof and cross-reference key results in the paper with the corresponding lemmas and theorems in the formal Coq development [1].

Third, as a case study in applying abstract RTA to specific scheduling problems, we provide eight formally verified response-time analyses for non-self-suspending sporadic tasks with arbitrary deadlines and arbitrary arrival curves under EDF, NP-EDF, LP-EDF, EDF-NPS, FP, NP-FP, LP-FP, and FP-NPS uniprocessor scheduling. For seven of these schedulers, this paper presents *the first formal verification* of a state-of-the-art RTA. In fact, to the best of our knowledge, for three of these policies – NP-EDF, LP-EDF, and EDF-NPS – we provide *the first known RTAs*. For the other five policies, this paper verifies the known exact RTAs. In particular, the four RTAs for FP scheduling verified in this paper correspond to results that can also be obtained from Yao et al.'s general analysis of LP-FP scheduling [46].

2 System Model

In this section, we describe the general system model on which we base our analysis. We focus on unit-speed uniprocessor systems in this paper and assume a discrete-time model, where the smallest quantity $\varepsilon \triangleq 1$ represents an indivisible unit of time (e.g., a processor cycle).

Workload. We consider workloads modeled as a set of n sporadic real-time tasks $\tau = \{\tau_1, \ldots, \tau_n\}$. Each task $\tau_i = (C_i, D_i, \alpha_i)$ is characterized by its *worst-case execution time* (or *cost*) C_i, its *relative deadline* D_i, and an *arrival-bound function* $\alpha_i(\Delta)$, which upper-bounds the number of times that τ_i is activated in any interval of length Δ. We also define the *request-bound function* (*RBF*) of task τ_i as $RBF_i(\Delta) \triangleq C_i \times \alpha_i(\Delta)$.

Whenever a task is activated, a corresponding *job* is released. We let $J_{i,j}$ denote the j-th job (or activation) of task τ_i. Each job $J_{i,j}$ has a *release* (or *activation*) time $a_{i,j}$, *absolute deadline* $d_{i,j} = a_{i,j} + D_i$, and *execution time* $c_{i,j}$, where $0 \leq c_{i,j} \leq C_i$. To finish, $J_{i,j}$ must

[2] We use the term "hypothesis" in the mathematical sense to refer to an explicitly stated assumption upon which a proof rests, not in the colloquial sense meaning an "unproven theory or conjecture."

receive exactly $c_{i,j}$ units of service from the scheduler. We denote $J_{i,j}$'s *completion* (or *finish*) time as $f_{i,j}$. A job's cost may be zero, in which case it is trivially finished immediately upon release. A job is *pending* at time t if it is released and not yet completed (i.e., if $a_{i,j} \leq t < f_{i,j}$).

A job's *response time* $r_{i,j}$ is given by $r_{i,j} \triangleq f_{i,j} - a_{i,j}$. The goal of a *response-time analysis* of task τ_i is to establish an upper bound R_i such that $r_{i,j} \leq R_i$ for any job $J_{i,j}$ of τ_i. Note that such an upper bound need not be exact, nor does it necessarily exist.

Schedule and Service. We next define the central notions that relate the workload to the underlying processor model. For clarity, we let $\mathbb{T}$ denote the *time* domain (the natural numbers including zero, i.e., $\mathbb{T} = \mathbb{N}$), $\mathbb{B} = \{0, 1\}$ the boolean domain, and $\mathbb{J} = \{J_{i,j}\}_{\forall i,j}$ the set of all jobs. Furthermore, we let $\mathbb{S}$ denote the *service* domain, which is another synonym of the natural numbers (i.e., $\mathbb{S} = \mathbb{N}$) that we use to disambiguate the notions of an instant or duration $t \in \mathbb{T}$ and the amount of accumulated service $\rho \in \mathbb{S}$ received by a job.

A *schedule* is a function $\sigma : \mathbb{T} \to \{\bot\} \cup \mathbb{J}$ that maps each point in time t to the job (if any) that is scheduled at time t, or to a constant $\bot$ that indicates that the processor is idle. A job can be scheduled only when it is pending.

A *scheduling policy* determines how the schedule is constructed. In Sections 3–6.1, we place no restriction on the type of scheduling policy considered. In Section 6.2, we focus our attention on *job-level fixed-priority* (JLFP) policies [9], which are policies that assign a fixed priority to each pending job, and then execute (any one of) the job(s) with maximal priority. Finally, in Section 7, we instantiate our abstract RTA for two specific JLFP policies, namely the two most prominent representatives of this class: FP and EDF scheduling. We let $J_k \succeq J_i$ denote that J_k has a priority that is higher than or equal to J_i's priority, and let $J_k \succ J_i$ denote that J_k has priority strictly higher than J_i.

The scheduler is assumed to be work-conserving, which we define precisely in Section 3.2. We further consider all scheduling overheads to be negligible, or equivalently, to already be included in each job's cost $c_{i,j}$. While our framework and proof strategy do not preclude the explicit modeling of overheads, a detailed consideration of overhead accounting issues is beyond the scope of this paper and left to future work.

As we assume a unit-speed uniprocessor, a job $J_{i,j}$ receives one unit of service at time t iff $\sigma(t) = J_{i,j}$. The *cumulative service* of a job $J_{i,j}$ received within a time interval $[t_1, t_2)$, denoted $serv_\sigma(J_{i,j}, [t_1, t_2))$, is given by $serv_\sigma(J_{i,j}, [t_1, t_2)) \triangleq |\{t \mid t \in [t_1, t_2) \ \wedge \ \sigma(t) = J_{i,j}\}|$. We also rely on the notions of the *total service* received up to time t, denoted $serv_\sigma(J_{i,j}, t) \triangleq serv_\sigma(J_{i,j}, [0, t))$, and the service received by task τ_i in time interval $[t_1, t_2)$, which is the total service received by jobs of τ_i in the given interval: $serv^{\tau_i}_\sigma([t_1, t_2)) \triangleq \sum_j serv_\sigma(J_{i,j}, [t_1, t_2))$.

Arrivals and Workload. For notational clarity, we introduce the *arrival sequence* $a(t) \triangleq \{J_{i,j} \mid \forall i, j : a_{i,j} = t\}$, which is a function mapping each time t to the (possibly empty) set of jobs released at time t. The *workload* of jobs of task τ_i in a given time interval $[t_1, t_2)$ is the cumulative cost of all jobs of τ_i released in that interval: $wl^{\tau_i}([t_1, t_2)) \triangleq \sum \{c_{i,j} \mid \forall j : t_1 \leq a_{i,j} < t_2\}$. Similarly, $wl([t_1, t_2)) \triangleq \sum_{i=1}^{n} wl^{\tau_i}([t_1, t_2))$ is the workload of all jobs. (We write $\sum \{F(x) \mid \forall x : P(x)\}$ to denote $\sum_{x \in \{x \mid P(x)\}} F(x)$, for any F and P.)

Preemption Model. A job $J_{i,j}$ is represented by a sequence of $q_{i,j}$ non-preemptive segments $c_{i,j,k}$ such that $\sum_{k=1}^{q_{i,j}} c_{i,j,k} = c_{i,j}$. (In case of a fully preemptive job, each segment is simply of length ε, the smallest discrete quantity of time.) We denote the longest segment of job $J_{i,j}$ as $c^{max}_{i,j} = \max\{c_{i,j,k}\}_{1 \leq k \leq q_{i,j}}$, and correspondingly let NPS_i denote the *maximum non-preemptive segment length* of task τ_i, such that $c^{max}_{i,j} \leq NPS_i$ for any $J_{i,j}$.

We denote a job $J_{i,j}$'s last segment as $c_{i,j}^{last} = c_{i,j,(q_{i,j})}$. Importantly, once $J_{i,j}$ has received enough service to start $c_{i,j}^{last}$, it cannot be preempted until it completes. We call this amount of service the job's *run-to-completion threshold* $rct_{i,j} \triangleq c_{i,j} - c_{i,j}^{last} + \varepsilon$. Correspondingly, task τ_i's *run-to-completion bound* is a constant RCT_i such that $rct_{i,j} \leq RCT_i$ for each $J_{i,j}$. That is, once a job of task τ_i has received at least RCT_i units of service, it is guaranteed to have reached its last non-preemptive segment and thus to complete without further preemptions.

3 High-Level Overview and Abstract Foundation

We begin with a high-level overview of our approach and then introduce the foundation upon which the analysis rests, namely the definitions and hypotheses that form the abstract RTA.

We employ one of the most commonly used tools in computer science, namely *reductions*. Reductions play a central role in computer science as they allow transferring a solution for a fundamental problem to solve another, often more concrete problem. For example, recall the Boolean Satisfiability Problem (SAT), which asks to determine whether a given formula has a satisfying assignment. Now if one faces another, domain-specific problem A, as long as one has an algorithm that decides SAT (e.g., a SAT-solver) *and* a reduction that maps an instance of problem A to an equivalent instance of problem SAT, one can decide problem A by applying the reduction from A to SAT and then running a SAT-solver.

In this work, we use the conceptually same idea, but do not reduce to SAT. Rather, we define a small *abstract model*, which does not have all the intricacies of a conventional scheduling model, but which nevertheless allows us to state the problem of finding a response-time bound. If the resulting abstract model correctly captures the essential properties that are necessary for an RTA via the busy-window principle, then we can expect that a response-time bound for the abstract model can be transferred to a given concrete model via a sequence of reductions from the concrete to the abstract model.

As shown in Figure 1, the central result of this paper is the abstract RTA theorem (Theorem 18), which solves the RTA problem for the abstract model (Section 3.1). On top of the abstract model, we provide a sequence of model refinements along with proofs of correctness for the corresponding reductions (in Figure 1, the refinement sequence proceeds in reverse direction of the arrows). In particular, by assuming that tasks are sequential, it is possible to obtain a more accurate bound on interference (Section 5). From this common base, we then obtain general EDF and FP models (Theorems 31 and 32), which however still abstract from specific preemption models (Section 6). Ultimately, the refinement sequence connects the abstract model to concrete models; conversely, the reduction sequence (indicated by the arrows in Figure 1) yields RTAs for concrete models (Section 7).

As indicated in Figure 1, we split reductions along two orthogonal concerns:

- the *scheduling policy* – how concurrently pending jobs are sequenced; and
- the *preemption model* – when is it possible for the scheduler to change the executing job.

Indeed, both the scheduling policy and preemption model are essentially just parameters of the underlying abstractions. As they are independent of each other, we can construct reductions by combining available scheduling models and preemption policies, which we have done in this paper for all combinations of the EDF and FP scheduling policies and four preemption models, as illustrated in Figure 1 and discussed in Section 7.

Importantly, our proof framework is flexible in the sense that one has the freedom to expand the graph of reductions depicted in Figure 1 at any node. For instance, in future work it will be possible to add branches for FIFO scheduling (e.g., as a sibling to Theorems 31 and 32), another preemption model (e.g., scheduling with preemption thresholds), other task models (e.g., self-suspending tasks), or support for real-time locking protocols.

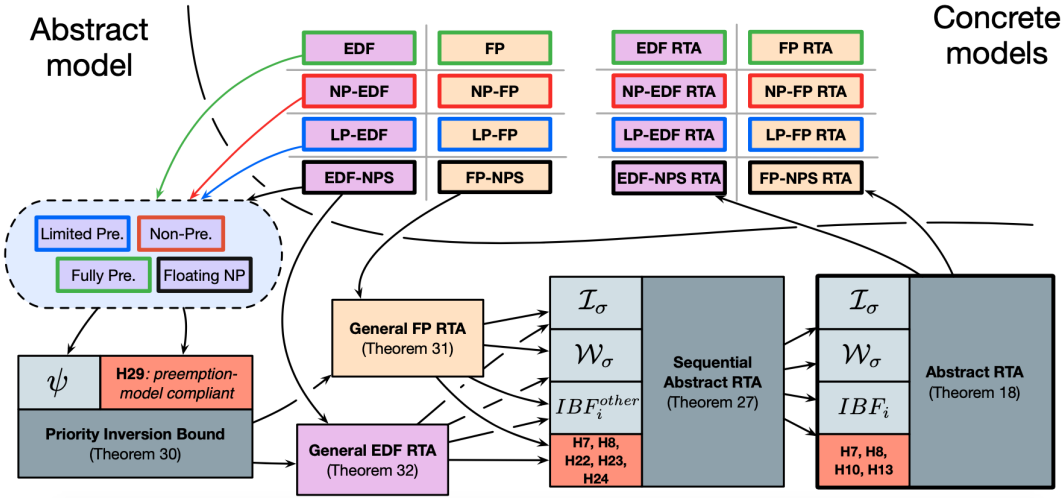


Figure 1 Structure of the presented abstract RTA framework. Black arrows indicate reduction steps from concrete to abstract models. For each reduction (e.g., NP-EDF RTA ↦ abstract RTA), concrete definitions of the interface functions must be instantiated and all abstract analysis hypotheses must be proven to hold for the specific preemption model (e.g., NP) and scheduler (e.g., EDF).

3.1 Abstract Model

At the core of the proposed framework lies the abstract model, which rests on four central hypotheses. Intuitively speaking, we assume that the processor is not overloaded (Hypothesis 7), that the scheduler is work-conserving (Hypothesis 8), that worst-case interference is bounded (Hypothesis 10), and that a solution to the response-time recurrence is known (Hypothesis 13). We define these assumptions precisely in Sections 3.4, 3.5, and 3.7 below.

▶ **Definition 1.** *An* abstract model *is a tuple* $(\tau_i, \mathcal{I}_\sigma, \mathcal{W}_\sigma, IBF_i)$*, where* τ_i *is the task under analysis,* $\mathcal{I}_\sigma$ *is the (abstract)* interference *function,* $\mathcal{W}_\sigma$ *is the (abstract)* interfering workload *function,* IBF_i *is the* interference-bound function*, and where* $\mathcal{I}_\sigma$*,* $\mathcal{W}_\sigma$*, and* IBF_i *satisfy the requirements imposed in Hypotheses 7, 8, 10, and 13.*

The three functions describe the evolution of an (abstract) system and thereby form the *interface* of abstract RTA. Specifically, reducing a concrete model to the abstract model means **(i)** *instantiating* $\mathcal{I}_\sigma$, $\mathcal{W}_\sigma$, and IBF_i such that they capture the concrete model's semantics, and **(ii)** *proving* that the chosen definitions satisfy Hypotheses 7, 8, 10, and 13.

Given an abstract model $(\tau_i, \mathcal{I}_\sigma, \mathcal{W}_\sigma, IBF_i)$, the problem of bounding the worst-case response time of τ_i is defined as follows.

▶ **Definition 2** (abstract RTA problem)**.** *Given a constant* R*, decide whether the inequality* $f_{i,j} - a_{i,j} \leq R$ *holds for every job* $J_{i,j}$ *of task* τ_i*, for any arrival sequence* $a(t)$ *and schedule* $\sigma(t)$ *consistent with* $(\tau_i, \mathcal{I}_\sigma, \mathcal{W}_\sigma, IBF_i)$ *in the sense of Hypotheses 7, 8, 10, and 13.*

We emphasize that we place no restrictions on $a(t)$ and $\sigma(t)$ other than the (arguably quite weak) restrictions imposed by the task model and Hypotheses 7, 8, 10, and 13. In particular, note that Definitions 1 and 2 are silent on matters of scheduling policy, preemption model, task-model specifics, etc., which are all abstracted by the interface functions, as discussed next.

3.2 Abstract Interference

A job's execution may be postponed by the environment and/or the system due to many different factors such as preemption by higher-priority jobs, non-preemptive lower-priority jobs, jitter, black-out periods in hierarchical scheduling, etc. We collectively refer to *any* such delay as (*abstract*) *interference*. In particular, we consider the execution of other jobs of the same task, frequently called self-interference in prior work, to also constitute interference.

Formally speaking, interference is a function $\mathcal{I}_\sigma : \mathbb{J} \times \mathbb{T} \to \mathbb{B}$ that satisfies Hypotheses 7, 8, 10, and 13. Intuitively, $\mathcal{I}_\sigma$ is a predicate that determines, in the context of a given schedule σ, for any job $J_{i,j}$ and any time t, whether $J_{i,j}$ *could* execute at time t *if it were pending* at time t. Note that a job does not have to be *actually* pending to experience (abstract) interference. This seemingly unnatural definition will turn out to be useful when bounding the response time of jobs that arrive after the beginning of a busy window (Hypothesis 10). To illustrate the idea, we provide a simple example; further examples can be found in Section 7.

▶ **Example 3.** For a fully preemptive JLFP model assuming Liu & Layland [31] tasks (i.e., no self-suspensions or shared resources), $\mathcal{I}_\sigma$ can be defined as follows:

$$\mathcal{I}_\sigma(J_{i,j}, t) \triangleq \exists\, J_{h,k} :\ J_{h,k} \neq J_{i,j} \ \wedge\ \sigma(t) = J_{h,k} \ \wedge\ J_{h,k} \succeq J_{i,j}.$$

That is, interference occurs if another job with higher or equal priority is scheduled. See Figure 2 for an illustration. In Section 7, we later provide a similar instantiation that also works in the presence of non-preemptive sections.

In general, conceptually similar interference functions can be defined for many schedulers and workload models, but finding the most appropriate definition can require some ingenuity in case of complex workloads or intricate scheduling policies.

Building on the interference function $\mathcal{I}_\sigma$, we define the *cumulative interference* $C_\mathcal{I}$ of a job $J_{i,j}$ within a time interval $[t_1, t_2)$ as $C_\mathcal{I}(J_{i,j}, [t_1, t_2)) \triangleq \sum_{t=t_1}^{t_2-1} [\![\mathcal{I}_\sigma(J_{i,j}, t)]\!]_\mathbb{1}$, where $[\![x]\!]_\mathbb{1}$ denotes the *indicator function* that evaluates to 1 if x is true, and 0 otherwise.

3.3 Abstract Interfering Workload

The second function of the abstract interface, called *interfering workload* $\mathcal{W}_\sigma$, is of a more technical nature and is intended to describe the *potential for future interference*, in the sense that it allows "foreseeing" (in the context of a fixed schedule σ) the amount of interference that a job can incur in the future. For example, the release of a higher-priority job with cost $c_{i,j}$ means that a lower-priority job may subsequently suffer $c_{i,j}$ units of interference.

Formally, the interfering workload $\mathcal{W}_\sigma : \mathbb{J} \times \mathbb{T} \to \mathbb{N}$ is a function that satisfies Hypotheses 7, 8, 10, and 13. Intuitively, it is useful to think of the function as indicating, for any job $J_{i,j}$ and any time t, the amount of potential interference for job $J_{i,j}$ that is introduced into the system at time t in a given schedule σ. This idea will become clearer with an example.

▶ **Example 4.** Continuing Example 3, an appropriate *interfering workload* function for a fully preemptive JLFP scheduling model is given by

$$\mathcal{W}_\sigma(J_{i,j}, t) \triangleq \sum \{ c_{h,k} \mid \forall J_{h,k} \in a(t) \setminus \{J_{i,j}\} : J_{h,k} \succeq J_{i,j} \}.$$

That is, the interfering workload introduced at time t is simply the total cost of all higher- or equal-priority jobs released at t (excluding job $J_{i,j}$ itself). See Figure 2 for an illustration.

While $\mathcal{W}_\sigma$ may appear to be a somewhat unfamiliar concept on first sight, its essential role will become fully apparent next with the introduction of a general notion of "busy window."

Analogously to cumulative interference, we define the *cumulative interfering workload* $C_\mathcal{W}$ of a job $J_{i,j}$ within a time interval $[t_1, t_2)$ as the aggregation of $\mathcal{W}_\sigma$ across said interval.

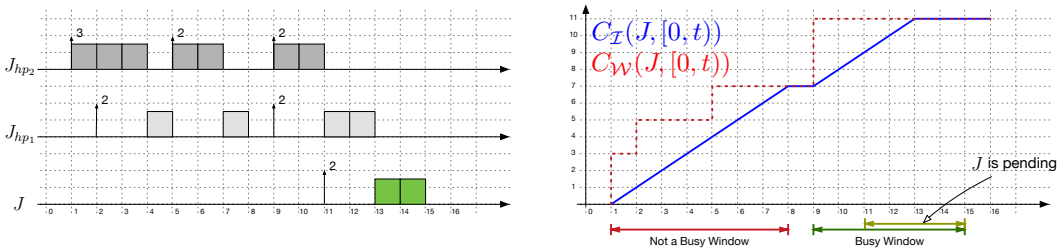


Figure 2 Illustration of an abstract busy window. Let J, J_{hp_1}, J_{hp_2} be arbitrary jobs such that $J_{hp_2} \succeq J_{hp_1} \succeq J$. The dashed red curve shows $C_{\mathcal{W}}(J,[0,t))$, and the solid blue curve shows $C_{\mathcal{I}}(J,[0,t))$, with $\mathcal{I}_\sigma$ and $\mathcal{W}_\sigma$ defined as in Examples 3 and 4. Intuitively, the difference $C_{\mathcal{W}}(J,[0,t)) - C_{\mathcal{I}}(J,[0,t))$ expresses the amount of "pending" interference that can defer the execution of job J.

3.4 Abstract Busy Window

As motivated in Section 1, the ultimate goal of this paper is to establish a general and reusable formal foundation for the busy-window principle, which necessarily requires a general and precise definition of the concept of a "busy window" (or, interchangeably, "busy interval"). While it is (deceptively) easy to state the *intuitive* idea of a busy window – the processor is continuously "busy" executing a job under analysis or some interfering workload – and while it is also usually not too difficult to define this idea for a given *specific* model, coming up with a good definition in the general case is another matter entirely.

In fact, this challenge is the motivation behind the choice of abstract interface functions $\mathcal{I}_\sigma$ and $\mathcal{W}_\sigma$. Using these two functions, we can define a single, general notion of a "busy interval" that is independent of any specific model, and which thus can capture the essence of the busy-window idea even across very different scheduler and workload models.

The key insight is that the workload function $\mathcal{W}_\sigma$ *produces* interfering workload, while the interference function $\mathcal{I}_\sigma$ *consumes* such workload. Thus, when $C_{\mathcal{W}}$ equals $C_{\mathcal{I}}$, there is no more interfering workload to process, so the processor necessarily becomes "quiet" and the corresponding busy interval ends. This observation leads us to the following definition.

▶ **Definition 5** (quiet time). *A point in time t is a quiet time w.r.t. a job $J_{i,j}$ if*

$$C_{\mathcal{I}}(J_{i,j}, [0,t)) = C_{\mathcal{W}}(J_{i,j}, [0,t)) \wedge (t \leq a_{i,j} \vee f_{i,j} \leq t).$$

In other words, at a quiet time t, we require the cumulative interference up to time t to be equal to the cumulative interfering workload, which indicates that the potential interference seen so far has been fully "consumed" (i.e., no more higher-priority work or other kinds of delay are pending). Furthermore, to ensure that a job $J_{i,j}$'s busy interval (defined next) actually captures its execution, we require that $J_{i,j}$ cannot *both* be pending before the quiet time t *and* also at time t (i.e., $\neg(a_{i,j} < t \wedge t < f_{i,j}) \Leftrightarrow t \leq a_{i,j} \vee f_{i,j} \leq t$). Thus:

▶ **Definition 6** (busy interval). *An interval $[t_1, t_2)$ is a busy interval w.r.t. job $J_{i,j}$ if* ***(i)*** *$a_{i,j} \in [t_1, t_2)$,* ***(ii)*** *t_1 is a quiet time* ***(iii)*** *t_2 is a quiet time, and* ***(iv)*** *no $t \in (t_1, t_2)$ is a quiet time.*

In other words, we say that a given interval is a job's busy interval if the interval contains the arrival of the job, starts with a quiet time, and remains non-quiet until it ends with a quiet time. Figure 2 illustrates the busy-window concept for a fully preemptive JLFP model.

Note that it follows from Definition 6 that a job's busy window, if it exists, is unique (which we formally establish in our Coq proof). It also bears repeating that Definition 6 applies to a specific job as this simplifies the formal Coq development, whereas prior work traditionally defines the notion of a busy window w.r.t. a task or priority level [30].

With the abstract notion of a busy interval in place, we can finally state our first two hypotheses, which constrain $\mathcal{I}_\sigma$ and $\mathcal{W}_\sigma$ via Definitions 5 and 6. First, obviously a response-time bound can exist only if busy windows are of finite length.

▶ **Hypothesis 7.** *Busy intervals are bounded by a constant L*: for any $J_{i,j}$, there is a busy interval $[t_1, t_2)$ w.r.t. $J_{i,j}$ and $t_2 - t_1 < L$.

Clearly, busy intervals are not actually always bounded; for example, if the processor is overloaded. However, no response-time analysis is applicable to workloads that exhibit unbounded busy intervals. Hypothesis 7 thus must be checked and proven to be satisfied in order for abstract RTA to be applicable. In fact, this hypothesis is analogous to the common requirement that a task set's total utilization cannot exceed 100%. However, Hypothesis 7 is more general since it reflects also any other abstract interference factors modeled by $\mathcal{I}_\sigma$.

Next, we require "work conservation" in an abstract sense.

▶ **Hypothesis 8.** The scheduler is *work-conserving in the abstract sense*: for any job $J_{i,j}$, its busy interval $[t_1, t_2)$, and any point in time $t \in [t_1, t_2)$, $J_{i,j}$ incurs interference at time t iff $J_{i,j}$ is not scheduled at time t: $\mathcal{I}_\sigma(J_{i,j}, t) \iff \sigma(t) \neq J_{i,j}$.

Intuitively, Hypothesis 8 requires the abstract interference predicate to describe the interference "correctly" – the scheduler must either schedule $J_{i,j}$, or there is something else that interferes with $J_{i,j}$, which matches the intuitive understanding of "work conservation." Note that we consider only time instants within a busy interval; we can thus be sure that there is some pending interference in the system (otherwise t would be a quiet time).

However, it is also interesting to note that Hypothesis 8 does *not* state whether the processor idles at time t (i.e., $\sigma(t) = \bot$ is permissible under Hypothesis 8 if $\mathcal{I}_\sigma(J_{i,j}, t)$ holds), which is necessary for generality: otherwise, abstract RTA would not be applicable to workload models with self-suspensions, release jitter, or delayed budget replenishments.

3.5 Abstract Interference Bound Function

The *interference-bound function* $IBF_i : \mathbb{T} \times \mathbb{T} \to \mathbb{T}$ is the third and last element of the behavioral interface of our analysis. The notion of an IBF as a bound on interference during some interval is well-known from prior work on the analysis of specific schedulers. However, to obtain the generality needed for abstract RTA, our definition differs from prior work in that it explicitly considers the *relative offset* of the job under analysis within its busy interval.

▶ **Definition 9.** *Given a job $J_{i,j}$ and its busy interval $[t_1, t_2)$, we call $A \triangleq a_{i,j} - t_1$ the* offset *(or* relative arrival time*) of job $J_{i,j}$ w.r.t. the beginning of the busy interval t_1.*

Based on this notion, abstract RTA requires the existence of a function $IBF_i(A, \Delta)$ that yields a bound on the maximum interference incurred by any job $J_{i,j}$ of task τ_i during an interval of length Δ starting with $J_{i,j}$'s busy window *assuming $J_{i,j}$ has a relative offset A.*

▶ **Hypothesis 10.** Given any job $J_{i,j}$ of a task τ_i and a schedule σ, $J_{i,j}$'s busy interval $[t_1, t_2)$ in σ, and a subinterval $[t_1, t_1 + \Delta) \subseteq [t_1, t_2)$, $IBF_i(a_{i,j} - t_1, \Delta)$ bounds the cumulative interference incurred by $J_{i,j}$ during $[t_1, t_1 + \Delta)$: $C_{\mathcal{I}}(J_{i,j}, [t_1, t_1 + \Delta)) \leq IBF_i(a_{i,j} - t_1, \Delta)$.

Note that IBF_i bounds the total interference within an interval starting at time t_1 regardless of the job's arrival time. That is, the interval starts from the beginning of the job's busy interval, and not from its arrival time. The reason to add the job's relative offset as a parameter of IBF_i is that it extends the set of possible interference bounds, which is

important when analyzing EDF scheduling, where a job's priority depends on its arrival time. To illustrate the idea, consider the following example assuming FP scheduling, where we write $\tau_h \succeq \tau_i$ to indicate that task τ_h has priority no lower than task τ_i.

▶ **Example 11.** For a fully preemptive FP model, IBF_i can be defined as follows:

$$IBF_i(A, \Delta) \triangleq \left(\sum \{ RBF_h(\Delta) \mid \forall \tau_h : \tau_h \succeq \tau_i \} \right) - X,$$

where $X = C_i$ if $\Delta > A$, and $X = 0$ otherwise. The term X is subtracted to exclude the cost of the job under analysis itself. Note that IBF_i considers as interference even jobs of task τ_i that arrive *later* than A time units after the beginning of a busy interval (i.e., *after* the job under analysis arrives). While this is required in the most general case (e.g., for non-sequential tasks models), it is overly pessimistic in the context of sequential tasks. In Section 5, we define a refined model for sequential tasks to eliminate this pessimism.

To clarify, both $\mathcal{W}_\sigma$ and IBF_i bound the interference of a given job. However, $\mathcal{W}_\sigma$ is primarily a modeling construct (for defining abstract busy windows) that yields a bound depending on a *specific* job arrival sequence and schedule σ, whereas IBF_i is an analysis tool that bounds interference in *any* schedule σ (compliant with the task model).

3.6 Abstract Response-Time Bound

Using IBF_i, we can proceed to state the actual response-time bound. Consider a job $J_{i,j}$ that arrives A time units after the beginning of its busy window $[t_1, t_2)$. Following a line of reasoning similar to those given by Davis et al. [12] and Yao et al. [46], we use the fact that once $J_{i,j}$ receives $rct_{i,j} \leq RCT_i$ units of service, it becomes non-preemptive and runs to completion, requiring no more than $c_{i,j}^{last} - \varepsilon \leq C_i - RCT_i$ additional units of service. The central question thus is: by when will $J_{i,j}$ receive RCT_i units of service at the latest?

By Hypothesis 8, $J_{i,j}$ receives service whenever it does not incur interference. Thus, if we find some $F \geq 0$ such that $A + F = RCT_i + IBF_i(A, A + F)$, then we can conclude that $J_{i,j}$ has received *at least* RCT_i units of service by time $t_1 + A + F$ (or has already completed), and thus will complete no later than by time $t_1 + A + F + (C_i - RCT_i)$. Since $f_{i,j} \leq t_1 + A + F + (C_i - RCT_i)$, and by assumption $a_{i,j} = t_1 + A$, we have $r_{i,j} \leq F + (C_i - RCT_i)$.

However, since $J_{i,j}$'s relative arrival time A is unknown in general, this inequality does not immediately yield a useful bound. Nonetheless, as a first step, we observe that the solution F that is maximal for all $A \in \mathbb{T}$ implies a response-time bound. More precisely, a constant R is a response-time bound for τ_i, if for each $A \in \mathbb{T}$ there exists an F such that

$$A + F = RCT_i + IBF_i(A, A + F) \quad \text{and} \quad F + (C_i - RCT_i) \leq R. \tag{1}$$

3.7 Finite Search Space

Equation (1) has no bound on the set of possible offsets (i.e., $A \in \mathbb{T}$), thus while being correct, it is not practical in the sense that it cannot be used to actually compute such an R as that would require enumerating all $A \in \mathbb{T}$. Therefore, to obtain a foundation for a practical analysis, we must restrict Equation (1) to a finite search space. In this section, we focus on the high-level idea underlying the abstract search space, and defer the proof that supports this reasoning to Section 4.2.

First, given Hypothesis 7, it is easy to see that no job of task τ_i has a relative arrival time $A \geq L$ as there is no busy interval longer than L. However, simply limiting the maximum magnitude of A is insufficient since L can be very large (i.e., enumerating every $A < L$ is impractical, given billions of processor cycles per second).

To obtain a sparse search space, it is useful to note that in Equation (1) IBF_i is the only varying term dependent on A. The function IBF_i in turn depends on only two parameters, A and Δ. We can thus ignore any relative offset A for which IBF_i "does not change." Conversely, and more precisely, we say that an offset $A > 0$ is *in the search space* if the partially applied function $IBF_i(A - \varepsilon, \cdot)$ is not equivalent to the partially applied function $IBF_i(A, \cdot)$, that is, if there exists a $\Delta < L$ such that $IBF_i(A - \varepsilon, \Delta) \neq IBF_i(A, \Delta)$.

Finally, as a base case, we always include $A = 0$, which yields the following search space.

▶ **Definition 12.** *The* search space *of task τ_i is given by*

$$\mathcal{A}_i \triangleq \{0\} \cup \left(\bigcup \{A \mid 0 < A < L \ \wedge \ \exists \Delta,\, IBF_i(A - \varepsilon,\, \Delta) \neq IBF_i(A,\, \Delta)\} \right).$$

This leads us to the final hypothesis upon which abstract RTA rests, namely the assumption that a bound R satisfying Equation (1) is known for all relevant offsets (i.e., for any $A \in \mathcal{A}_i$).

▶ **Hypothesis 13.** There exists a constant R such that $\forall A \in \mathcal{A}_i, \exists F$,

$$A + F = RCT_i + IBF_i(A,\, A + F) \text{ and } F + (C_i - RCT_i) \leq R. \tag{2}$$

In more traditional terms, we assume that R is an upper bound on the fixed point of the response-time recurrence for each "relevant" release offset A. Note that we intentionally do not specify how such a constant R is obtained (i.e., by classic fixed-point iteration, exhaustive search, querying some oracle such as a linear-program solver, or by any other means). To establish that abstract RTA is sound (i.e., to show that R is indeed a response-time bound), it is sufficient to know that R bounds each fixed point, regardless of how it was found.

4 Abstract Response-Time Analysis

In this section, we present key proofs that are essential to understanding our analysis. The core of the analysis consists of three theorems: Theorem 15 relates the service received by a job and an interference bound, Theorem 17 proves the reduced search space to be sufficient, and finally Theorem 18 joins the prior two theorems to obtain the abstract response-time bound. The proofs in this section rely solely on Hypotheses 7, 8, 10, and 13 and the definitions in Section 2 (i.e., no additional implicit assumptions are made, which is verified by our Coq proof).

4.1 Service Theorem

Consider an arbitrary job $J_{i,j}$ and its busy interval $[t_1, t_2)$. We begin by observing that, within any time interval $[t, t + \Delta) \subseteq [t_1, t_2)$, the sum of the cumulative service received by $J_{i,j}$ and the cumulative interference experienced by $J_{i,j}$ equals Δ.

▶ **Lemma 14.** *In any time interval $[t, t + \Delta) \subseteq [t_1, t_2)$:*

$$serv_\sigma(J_{i,j}, [t, t + \Delta)) + C_{\mathcal{I}}(J_{i,j}, [t, t + \Delta)) = \Delta.$$

Proof. At any instant $t \in [t, t + \Delta)$, $J_{i,j}$ either receives a unit of service or incurs one unit of interference (but not both) since the scheduler is work-conserving (Hypothesis 8). From the definitions of cumulative service (Section 2) and cumulative interference (Section 3.2), it follows that the left-hand side is a sum of Δ terms equal to one, with the rest being zero. ◀

Next, note that the service received by $J_{i,j}$ outside its busy interval $[t_1, t_2)$ is equal to zero (which follows from the definition of busy interval), and that a job must be completed by the end of its busy interval. Thus, $J_{i,j}$ receives $c_{i,j}$ units of service during $[t_1, t_2)$, which implies that, for every $\rho \in [0, c_{i,j}]$, there exists a Δ such that $serv_\sigma(J_{i,j}, [t_1, t_1 + \Delta)) = \rho$.

Put differently, a bound of the form $\rho \leq serv_\sigma(J_{i,j}, [t_1, t_1 + \Delta))$ states that $J_{i,j}$ has received *at least* ρ units of service by time $t_1 + \Delta$. The main service theorem, which we establish next, serves to relate such a lower bound on cumulative service to the cumulative interference incurred by $J_{i,j}$ so far. More precisely, if the sum of ρ and the cumulative interference experienced by $J_{i,j}$ within the interval $[t_1, t_1 + \Delta)$ does not exceed Δ, then it follows that $J_{i,j}$ has indeed received at least ρ units of service by time $t_1 + \Delta$.

▶ **Theorem 15.** *For any $\rho \in [0, c_{i,j}]$ and Δ, if $\rho + C_{\mathcal{I}}(J_{i,j}, [t_1, t_1 + \Delta)) \leq \Delta$, then*

$\rho \leq serv_\sigma(J_{i,j}, [t_1, t_1 + \Delta))$.

Proof. By Lemma 14, $\Delta = serv_\sigma(J_{i,j}, [t_1, t_1 + \Delta)) + C_{\mathcal{I}}(J_{i,j}, [t_1, t_1 + \Delta))$, and from the premise we have $\rho + C_{\mathcal{I}}(J_{i,j}, [t_1, t_1 + \Delta)) \leq \Delta$, which implies $\rho \leq serv_\sigma(J_{i,j}, [t_1, t_1 + \Delta))$. ◀

This simple but important fact shows that we automatically gain a lower bound on the total amount of service received by a job whenever we upper-bound the cumulative abstract interference that it has incurred. Theorem 15 will be the main tool for proving the abstract RTA theorem (Theorem 18).

4.2 Reduction of the Search Space

As discussed in Section 3.7, the notion of an explicit search space serves to shrink the set of equations that must be solved to obtain a safe response-time bound for a task. While we have already stated and *intuitively* explained the search space in Section 3.7, we now provide a more rigorous argument.

Our method to justify the reduced search space differs from the approaches employed in prior papers, which pursue roughly the following argument: in certain situations, it is possible to transform a schedule σ into an equivalent schedule σ', where the job under analysis arrives one time unit earlier (i.e., $a'_{i,j} = a_{i,j} - \varepsilon$), so that it is sufficient to consider only σ'. That is, one can "move" the arrival of a critical job "to the left" until a schedule belonging to the search space is found. However, while this sketch is intuitively appealing, it is difficult to find a simple argument for why the schedule transformation is correct even for *specific* models, let alone in the abstract general case. Our approach instead automates the reduction procedure based on a more rigorous (but less visual) argument.

Consider two offsets A_1 and A_2: we say that IBF_i is *equivalent* on these offsets if $IBF_i(A_1, \Delta) = IBF_i(A_2, \Delta)$ for any $\Delta < L$, which we denote as $IBF_i(A_1, \cdot) \equiv IBF_i(A_2, \cdot)$.

In the following, we prove that, if $IBF_i(A_1, \cdot) \equiv IBF_i(A_2, \cdot)$, then there is no point in keeping both A_1 and A_2 in the search space. To this end, we show how to transform the equation $A_1 + F_1 = RCT + IBF_i(A_1, A_1 + F_1)$ into the equation $A_2 + F_2 = RCT_i + IBF_i(A_2, A_2 + F_2)$ such that the solution F_1 can be easily computed from the solution F_2.

First, we show that, for *any* offset A, there is an offset A_s in the search space $\mathcal{A}_i$ such that IBF_i is equivalent on A and A_s.

▶ **Lemma 16.** *For any $A < L$, $\exists A_s \in \mathcal{A}_i$ such that $A_s \leq A$ and $IBF_i(A_s, \cdot) \equiv IBF_i(A, \cdot)$.*

Proof. By induction on A. The *base case* $A = 0$ is trivial since 0 is in the search space.
Induction step: given that the property holds for A, we need to prove the claim for $A + \varepsilon$.

Consider two cases.

Case 1: $\boldsymbol{IBF_i(A, \cdot) \equiv IBF_i(A + \varepsilon, \cdot)}$**.** By the induction hypothesis, there is an $A_s \in \mathcal{A}_i$ such that $A_s \leq A < A + \varepsilon$. Moreover, we have $IBF_i(A_s, \cdot) \equiv IBF_i(A, \cdot) \equiv IBF_i(A + \varepsilon, \cdot)$ by transitivity. Hence A_s satisfies the claim also for $A + \varepsilon$.

Case 2: $\boldsymbol{IBF_i(A, \cdot) \not\equiv IBF_i(A + \varepsilon, \cdot)}$**.** In this case, $A + \varepsilon$ is in the search space itself, because the fact that IBF_i is not equivalent on A and $A + \varepsilon$ implies that there exists a $\Delta < L$ such that $IBF_i((A + \varepsilon) - \varepsilon, \Delta) \neq IBF_i(A + \varepsilon, \Delta)$, which matches the criterion for inclusion in the search space. ◀

However, since the second parameter of $IBF_i(A, A + F)$ in Equation (2) also depends on A, Lemma 16 by itself does not yet allow us to substitute any arbitrary offset with an offset from the search space. Rather, we also need to transform the solution F. The following theorem justifies the replacement.

▶ **Theorem 17.** *Let $A_s \in \mathcal{A}_i$ and let F_s be the corresponding solution such that $A_s + F_s < L$ and $A_s + F_s = RCT_i + IBF_i(A_s, A_s + F_s)$. Then, for any offset $A \in [A_s, A_s + F_s]$ such that $IBF_i(A_s, \cdot) \equiv IBF_i(A, \cdot)$, there exists a solution F such that the following conditions are met:* ***(i)*** *$A_s + F_s = A + F$,* ***(ii)*** *$F \leq F_s$, and* ***(iii)*** *$A + F = RCT_i + IBF_i(A, A + F)$.*

Proof. It is easy to verify that for $F \triangleq A_s + F_s - A$ the conclusion of the theorem holds.

(i) The equality $A_s + F_s = A + F$ trivially holds.

(ii) From (i), by moving the term A_s to the right-hand side, we obtain $F_s = F + (A - A_s)$. Since $(A - A_s) \geq 0$, we have $F \leq F_s$.

(iii) By assumption, $A_s + F_s = RCT_i + IBF_i(A_s, A_s + F_s)$ and $IBF_i(A_s, \cdot) \equiv IBF_i(A, \cdot)$. Substitute $IBF_i(A_s, \cdot)$ in the former to obtain $A_s + F_s = RCT_i + IBF_i(A, A_s + F_s)$. Finally, we rewrite both sides using (i) to obtain $A + F = RCT_i + IBF_i(A, A + F)$.

◀

With Theorem 17 in place, we can obtain a result for any offset by (instead) analyzing an offset from the search space, which justifies the restriction to $A \in \mathcal{A}_i$ in Hypothesis 13.

4.3 Abstract RTA

In this section, the main proof is presented: using Theorems 15 and 17, we obtain that the constant R given in Hypothesis 13 is in fact a response-time bound for task τ_i.

The high-level idea of the proof is as follows. For an arbitrary job of task τ_i and its busy interval, we calculate the relative arrival time A. This offset is not necessarily in the search space, and therefore there is no direct access to the solution of the response-time recurrence with offset A. However, by Theorem 17, there is an equivalent solution of the recurrence for an offset included in the search space. We can use this solution to obtain an upper bound on interfering workload and use this bound to satisfy the premise of Theorem 15. In order to prove the theorem, we rely on all previously stated hypotheses.

▶ **Theorem 18** (Abstract RTA)**.** *Under the assumptions stated in Hypotheses 7, 8, 10, and 13, the response-time of task τ_i is bounded by R.*

We prove the theorem in the remainder of this section by case analysis in Lemmas 19–21. Let $J_{i,j}$ denote an arbitrary job of task τ_i. We must show that job $J_{i,j}$ is complete by time $a_{i,j} + R$ (or, equivalently, that $f_{i,j} \leq a_{i,j} + R$). By Hypothesis 7, there exists a finite busy interval $[t_1, t_2)$ of job $J_{i,j}$: let $A = a_{i,j} - t_1$ be the relative arrival time of job $J_{i,j}$ w.r.t. $[t_1, t_2)$. Further, as a zero-cost job trivially has a response time of zero, assume $c_{i,j} > 0$.

The goal is to apply Theorem 15 to obtain the time when $J_{i,j}$ receives sufficient service to become non-preemptive and run to completion, that is, we seek a Δ such that $rct_{i,j} \leq$

$RCT_i \leq serv_\sigma(J_{i,j}, [t_1, t_1 + \Delta))$. From Hypothesis 13, we can obtain the required bound on cumulative interference needed to satisfy the premise of Theorem 15 (as argued below).

Unfortunately, offset A does not necessarily belong to the search space; thus one cannot apply Hypothesis 13 directly. However, from Theorem 17 we know that, for any offset A, there exists another offset $A_s \in \mathcal{A}_i$ with an equivalent function IBF_i and solution F_s of the corresponding equation $A_s + F_s = RCT_i + IBF_i(A_s,\, A_s + F_s)$.

Depending on the value of $A_s + F_s$, consider the following cases: **(1)** $t_2 \leq t_1 + A_s + F_s$; **(2)** $t_1 + A_s + F_s < t_2$ and $A \leq A_s + F_s$; and **(3)** $t_1 + A_s + F_s < t_2$ and $A > A_s + F_s$. Clearly, these three cases cover all possibilities.

Case 1. In the first case, the solution $A_s + F_s$ of the response-time recurrence is larger than the length of the busy interval.

▶ **Lemma 19.** *If $t_2 \leq t_1 + A_s + F_s$, then $f_{i,j} \leq a_{i,j} + R$.*

Proof. We show that $f_{i,j} \leq t_2 \leq a_{i,j} + R$. The first inequality $f_{i,j} \leq t_2$ is a corollary of the definition of a busy interval (Definition 6). Time t_2 is a quiet time, and job $J_{i,j}$ arrives before t_2. Thus, by the definition of a quiet time (Definition 5), job $J_{i,j}$ is complete by time t_2. To prove the second inequality $t_2 \leq a_{i,j} + R$, consider the following chain of inequalities: $t_2 \leq t_1 + A_s + F_s \leq t_1 + A + F_s \leq a_{i,j} + F_s \leq a_{i,j} + F_s + (C_i - RCT_i) \leq a_{i,j} + R$. ◀

Case 2. Next, consider the case where the fixed point $A_s + F_s$ lies inside the busy interval, which is the theorem's main case.

Some additional reasoning is required since the term $C_i - RCT_i$ does not necessarily bound the term $c_{i,j} - rct_{i,j}$. That is, a job can have a small run-to-completion threshold $rct_{i,j}$, thereby becoming non-preemptive much *earlier* than guaranteed according to RCT_i, while simultaneously executing a final non-preemptive segment that is *longer* than RCT_i (e.g., this is possible in the case of floating non-preemptive sections). In this case, we cannot directly apply Theorem 15, because the response-time recurrence gives a "weak" bound on workload with $\rho = RCT_i$, whereas we need $\rho = rct_{i,j}$ in this case.

Intuitively, however, this is a good situation: if a job has a longer final non-preemptive segment, it necessarily also has a shorter maximum preemptive part (since $c_{i,j} \leq C_i$). Observing that such a job will become non-preemptive earlier, the response time of the job will not be worse than the response time of a job with a shorter last segment. To formally express this reasoning, we introduce a notion of *optimism* $\mu \triangleq RCT_i - rct_{i,j}$.

▶ **Lemma 20.** *If $t_1 + A_s + F_s < t_2$ and $A \leq A_s + F_s$, then $f_{i,j} \leq a_{i,j} + R$.*

Proof. We show that $J_{i,j}$ will complete by time $a_{i,j} + R$ by deriving that it completes by time instant $t_1 + (A + F - \mu) + (c_{i,j} - rct_{i,j})$, where $t_1 + (A + F - \mu)$ is the point in time by which $J_{i,j}$ receives $rct_{i,j}$ units of service, and $c_{i,j} - rct_{i,j}$ is the duration of $J_{i,j}$'s last non-preemptive segment.

Due to space constraints, we omit the step-by-step proof of the inequality $t_1 + (A + F - \mu) + (c_{i,j} - rct_{i,j}) \leq a_{i,j} + R$, but note that we have verified this fact in our Coq proof.

We apply Theorem 15 with $\rho = rct_{i,j}$ and $\Delta = A + F - \mu$. It then remains to be shown that $rct_{i,j} + C_{\mathcal{I}}(J_{i,j}, [t_1,\, t_1 + A + F - \mu)) \leq A + F - \mu$, which trivially follows since $rct_{i,j} + C_{\mathcal{I}}(J_{i,j}, [t_1, t_1 + A + F - \mu)) \leq RCT_i + IBF_i(A, A + F) - \mu$. Thus we know that job $J_{i,j}$ receives at least $rct_{i,j}$ units of service by time $t_1 + A + F - \mu$, which implies that $J_{i,j}$ completes by time $t_1 + (A + F - \mu) + (c_{i,j} - rct_{i,j}) \leq a_{i,j} + R$. ◀

Case 3. The final case can never arise and thus can be discarded from further consideration.

▶ **Lemma 21.** *The case $t_1 + A_s + F_s < t_2$ and $A > A_s + F_s$ is impossible.*

Proof. By contradiction. Suppose the converse is true. From the equation $A_s + F_s = RCT_i + IBF_i(A_s, A_s + F_s)$, it follows that $rct_{i,j} + C_{\mathcal{I}}(J_{i,j}, [t_1, t_1 + A_s + F_s)) \leq A_s + F_s$. Thus, we can apply Theorem 15 with parameters $\rho = rct_{i,j}$ and $\Delta = A_s + F_s$. By Theorem 15, $J_{i,j}$ receives at least $rct_{i,j}$ units of service by time $t_1 + A_s + F_s$, where $0 < \varepsilon \leq rct_{i,j}$. However, this is impossible since $J_{i,j}$ arrives after time $t_1 + A_s + F_s$, and $J_{i,j}$ cannot receive service (i.e., be scheduled) before it arrives. ◀

Lemmas 19–21 cover all cases, and in each possible case job $J_{i,j}$ completes by time $a_{i,j} + R$. Thus Theorem 18 holds.

5 Abstract Sequential Response-Time Analysis

In the analysis of uniprocessor systems, it is usually assumed that tasks are *sequential*, that is, jobs of the same task execute in order of their arrival. This assumption allows for tighter analyses because it allows for a better bound on *self-interference* (i.e., the case when a job is, possibly indirectly, delayed by other jobs of the same task). In this section, we consider such an extension (i.e., refinement) of the underlying abstract RTA, and to this end formally introduce the sequential-tasks assumption.

▶ **Hypothesis 22.** In schedule σ, *tasks are sequential*: for any task τ_i and any two jobs $J_{i,j}$ and $J_{i,k}$ of task τ_i, if $a_{i,j} < a_{i,k}$, then $\forall t,\ \sigma(t) = J_{i,k} \implies f_{i,j} \leq t$.

Assuming that tasks are sequential prevents a later-arriving job from interfering with an earlier job. This allows us to refine the interference bound function by isolating and removing the term that represents the self-interference contribution.

To support this property, in addition to Hypothesis 22, we require a technical condition to ensure that $\mathcal{I}_\sigma$ and $\mathcal{W}_\sigma$ are consistent with the sequential-tasks hypothesis (i.e., to rule out nonsensical definitions of $\mathcal{I}_\sigma$ and $\mathcal{W}_\sigma$). Specifically, Hypothesis 22 can theoretically contradict the priority policy, which is implicitly encoded by the two functions, through the following effect. Suppose there are two jobs $J_{i,j}$ and $J_{i,k}$ of task τ_i that are pending simultaneously and $a_{i,j} < a_{i,k}$. By Hypothesis 22, $J_{i,j}$ must execute before $J_{i,k}$. However, since we have a completely generic, abstract model, nothing stops a (pathological) priority policy from assigning a higher priority to the second job (i.e., $J_{i,k} \succ J_{i,j}$) so that $J_{i,k}$ must execute before $J_{i,j}$. To prevent such a contradiction, we impose an additional restriction on the interference function $\mathcal{I}_\sigma$ and the interfering workload function $\mathcal{W}_\sigma$.

▶ **Hypothesis 23.** *Functions $\mathcal{I}_\sigma$ and $\mathcal{W}_\sigma$ are consistent with Hypothesis 22*: for any job $J_{i,j}$ of task τ_i and its busy interval $[t_1, t_2)$, the total workload of the task up to time t_1 is equal to the total service received by the task up to time t_1. Formally, $wl^{\tau_i}([0, t_1)) = serv_\sigma^{\tau_i}([0, t_1))$.

To understand the intuition behind Hypothesis 23, recall that a busy interval is defined in terms of $\mathcal{I}_\sigma$ and $\mathcal{W}_\sigma$ (Definition 6). Thus, at the start of a job's busy interval, there should be no pending interference from other jobs of the same task. In other words, if upon arrival of a job $J_{i,j}$ there is already another pending job $J_{i,k}$ of the same task, then the busy interval of $J_{i,j}$ must extend to the arrival of job $J_{i,k}$.

Given Hypotheses 22 and 23, we can refine the response-time equation from Hypothesis 13 by pulling out the bound on self-interference from IBF_i. To this end and analogously to IBF_i, we let IBF_i^{other} denote a bound on any interference incurred by a job $J_{i,j}$ *excluding* any self-interference, and require that it satisfies the following invariant.

▶ **Hypothesis 24.** Given any job $J_{i,j}$ of a task τ_i and a schedule σ, $J_{i,j}$'s busy interval $[t_1, t_2)$ in σ, and a subinterval $[t_1, t_1 + \Delta) \subseteq [t_1, t_2)$, $IBF_i^{other}(a_{i,j} - t_1, \Delta)$ bounds the cumulative interference incurred by $J_{i,j}$ during $[t_1, t_1 + \Delta)$ *excluding* any self-interference: $C_{\mathcal{I}}(J_{i,j}, [t_1, t_1 + \Delta)) - (serv_\sigma^{\tau_i}([t_1, t_1 + \Delta)) - serv_\sigma(J_{i,j}, [t_1, t_1 + \Delta))) \leq IBF_i^{other}(a_{i,j} - t_1, \Delta)$.

Using the new notion, we can refine IBF_i from Example 11 as follows.

▶ **Example 25.** An appropriate IBF_i^{other} function for a fully preemptive FP model with *sequential tasks* is given by

$$IBF_i^{other}(A, \Delta) \triangleq \sum \{RBF_h(\Delta) \mid \forall \tau_h : \tau_h \neq \tau_i, \tau_h \succeq \tau_i\}.$$

Compared to IBF_i in Example 11, IBF_i^{other} explicitly excludes any contributions due to jobs of τ_i to discount any self-interference, which makes it less pessimistic for sequential tasks.

Recall from Section 2 that the system model assumes an arrival curve α_i to upper-bound the maximum number of activations of task τ_i, and that τ_i's request-bound function RBF_i is defined in terms of α_i. We use this information and Hypotheses 22–24 to upper-bound the maximum self-interference, and once we know the maximum self-interference, there is no need to include this term in the abstract function IBF_i^{other}. We state this fact as follows.

▶ **Lemma 26.** *Under Hypotheses 22–24, for any A and Δ:*

$$IBF_i(A, \Delta) \leq RBF_i(A + \varepsilon) - C_i + IBF_i^{other}(A, \Delta).$$

Proof. Consider a job $J_{i,j}$ of task τ_i and its busy interval $[t_1, t_2)$, and let $A = a_{i,j} - t_1$ denote $J_{i,j}$'s relative offset. By Hypothesis 23, all jobs of task τ_i that arrived prior to time t_1 are complete by t_1. By Hypothesis 22, **(i)** all jobs of task τ_i that arrive during $[t_1, t_1 + A)$ *will* interfere with $J_{i,j}$, **(ii)** jobs that arrive *at* time $a_{i,j} = t_1 + A$ (simultaneously with $J_{i,j}$) *may* interfere with $J_{i,j}$, and **(iii)** *none* of the jobs of task τ_i that arrive after time $a_{i,j}$ can interfere with $J_{i,j}$. Therefore, we must consider only jobs of τ_i that arrive during the interval $[t_1, t_1 + A]$ of length $A + \varepsilon$. According to the system model (Section 2), these jobs execute for no more than $RBF_i(A + \varepsilon)$ time units in total. Furthermore, we can subtract τ_i's cost C_i once since $J_{i,j}$ itself is one of the jobs accounted for by $RBF_i(A + \varepsilon)$. The final inequality thus follows from Hypothesis 24. ◀

Lemma 26 implies that we can bound the term $IBF_i(A, A + F)$ in Hypothesis 13 with the refined term $RBF_i(A + \varepsilon) - C_i + IBF_i^{other}(A, A + F)$, which yields the following theorem.

▶ **Theorem 27.** *Under the assumptions stated in Hypotheses 7, 8, and 22–24, if there exists a constant R such that $\forall A \in \mathcal{A}_i, \exists F$,*

$$A + F = RCT_i + (RBF_i(A + \varepsilon) - C_i) + IBF_i^{other}(A, A + F) \text{ and}$$
$$F + (C_i - RCT_i) \leq R,$$

then the response-time of task τ_i is bounded by R.

Proof. The claim follows from Theorem 18: the set of hypotheses remains mostly the same, with the only difference being that Hypotheses 10 and 13 have been refined with more specific assumptions. However, by Lemma 26, the refined bound $RBF_i(A+\varepsilon) - C_i + IBF_i^{other}(A, A+F)$ safely upper-bounds $IBF_i(A, A + F)$; abstract RTA thus applies. ◀

With Theorem 27 in place, we next clarify when preemptions can take place.

6 Preemption Model

Recall from Section 2 that any job $J_{i,j}$ can be represented as a sequence of $q_{i,j}$ segments during which the job executes non-preemptively. Under a discrete-time model as assumed herein, this is true even if $J_{i,j}$ is fully preemptive since, as a degenerate case, such a job can be seen as consisting of $c_{i,j}$ single-quantum segments (i.e., each of length ε).

Any non-preemptive segment starts with a corresponding *preemption point*, that is, a job can be preempted only in between segments. Accordingly, a *preemption model* is a policy that governs the placement of preemption points. Until now, we did not touch upon this aspect, because abstract RTA does not depend on any particular preemption model. Or rather, the specifics of the preemption model in use have been abstracted by the notion of RCT. However, to instantiate the analysis for specific schedulers and workloads, we need to specify an exact rule that determines when a job can be preempted.

To allow for a wide range of possible preemption models while retaining the ability to reason about preemptions in a high-level, general way, we use the notion of a *preemption predicate* $\psi : \mathbb{J} \times \mathbb{S} \to \mathbb{B}$ as a generic interface that abstracts from a job's specific structure and preemption points. More precisely, given any job $J_{i,j}$ and its *progress* $\rho \in [0, c_{i,j}]$ (i.e., an amount of service received so far), predicate $\psi(J_{i,j}, \rho)$ holds iff $J_{i,j}$ can be preempted at this point of its execution (i.e., after receiving exactly ρ units of service). Due to space constraints, we provide instantiations of ψ for concrete models in the extended version of this paper [6]. Interestingly, while one can think of a limited-preemptive model as a generalization (or super-model) of the other preemption models [8, 46], we found it actually easier to ignore this hierarchy in our proof. We thus instantiate ψ *directly* for each preemption model.

Since a preemption model restricts when a scheduler can enact changes to the schedule, we next relate the preemption predicate to a schedule σ and its underlying priority policy $\preceq$.

▶ **Definition 28.** *A schedule σ is* preemption-model compliant *if, for any job $J_{i,j}$ and time t, **(i)** $\psi(J_{i,j}, serv_\sigma(J_{i,j}, t)) \wedge \sigma(t) = J_{i,j}$ implies that $J_{i,j}$ has the maximal priority among all pending jobs and **(ii)** jobs can be non-preemptive only while they are executing: $\sigma(t) \neq J_{i,j} \implies \psi(J_{i,j}, serv_\sigma(J_{i,j}, t))$.*

In other words, all non-executing jobs are preemptable and the priority policy is respected whenever the scheduled job is preemptable, which in turn implies that jobs are preemptable at the moment of a context switch (i.e., whenever they start or stop executing).

6.1 Priority Inversion

While a job $J_{l,k}$ is executing a non-preemptive segment, there may be a pending higher-priority job $J_{i,j}$ waiting to be scheduled at $J_{l,k}$'s next preemption point, which is commonly known as *priority inversion*. To reason about the total duration of priority inversion that $J_{i,j}$ incurs, we first define a weaker predicate $lps(J_{i,j}, t)$ that is true whenever a lower-priority job $J_{l,k}$ is scheduled, regardless of whether $J_{i,j}$ is actually pending at time t.

$$lps(J_{i,j}, t) \triangleq \exists J_{l,k},\ \sigma(t) = J_{l,k} \wedge J_{i,j} \succ J_{l,k} \tag{3}$$

Intuitively, it may seem somewhat unnatural to ignore whether $J_{i,j}$ is pending at time t. However, the definition turns out to be convenient because it captures cases where a priority inversion at the start of a busy interval indirectly influences the response times of jobs that arrive later in the busy interval. Since we consider priority inversion, and thus apply Equation (3), only within a busy interval, it is sufficient (and easier) to use $lps(J_{i,j}, t)$.

Table 1 Preemption Model Parameters.

Preemption Model	RCT_i	B_i (where $\tau_l \in LP_i$)
Fully Preemptive	$C_i - \varepsilon$	$\varepsilon - \varepsilon = 0$
Fully Non-Preemptive	ε	$\max_l\{C_l - \varepsilon\}$
Fixed Preemption Points	$C_i - C_{i,m_i} + \varepsilon$	$\max_l\{\max_k\{C_{l,k}\} - \varepsilon\}$
Floating Non-Preemptive Regions	$C_i - \varepsilon$	$\max_l\{NPS_l - \varepsilon\}$

Therefore, building on Equation (3), we say that the *cumulative priority inversion* of a task τ_i is bounded by a constant B_i if, for any job $J_{i,j}$ of task τ_i and its busy interval $[t_1, t_2)$, it holds that $\sum_{t=t_1}^{t_2-1} [\![lps(J_{i,j}, t)]\!]_{\mathbb{1}} \leq B_i$.

While this notion of priority-inversion bound is more general than what is needed for non-preemptive segments, we chose it in anticipation of future work on locking protocols and other sources of priority inversion that can occur throughout a busy interval.

6.2 Bounded Non-Preemptive Segments

In this paper, we focus on priority inversions caused by lower-priority jobs executing non-preemptive segments of bounded length under a JLFP scheduling policy. To this end, we strengthen our assumptions about the schedule in the following.

▶ **Hypothesis 29.** The schedule σ is **(i)** preemption-model compliant (Definition 28) and **(ii)** *work-conserving in the classic sense*: for any job $J_{i,j}$ and any time t, if $J_{i,j}$ is pending and not scheduled at time t (i.e., $a_{i,j} \leq t < f_{i,j}$ and $\sigma(t) \neq J_{i,j}$), then there exists another job that is scheduled at time t: $\sigma(t) \neq \bot$.

Classic work-conservation is a strong property that implies abstract work-conservation (when instantiated as in Section 7), and also the absence of self-suspensions, delayed budget replenishments, or any other rules or workload properties that defer a pending job's execution.

It follows from Hypothesis 29 that a busy interval contains at most one contiguous interval of priority inversion, which necessarily occurs at the beginning of the busy interval (if at all), and which is bounded by the maximum non-preemptive segment length.

The following theorem serves as an intermediate layer between abstract RTA, which has no explicit notions of preemption or priority inversion (both are considered abstract interference), and concrete schedulers and workload models, which come with specific preemption models. The benefit of this intermediate layer is that we can address several preemption models together, irrespective of which scheduling policies they are combined with, so that we do not need to repeat identical proofs as part of each instantiation.

▶ **Theorem 30.** *Under Hypothesis 29, the cumulative priority inversion of τ_i is bounded by $B_i \triangleq \max\{NPS_l - \varepsilon \mid \forall \tau_l \in LP_i\}$, where $LP_i \subseteq \tau$ denotes the subset of tasks that can generate jobs that can cause jobs of task τ_i to suffer priority inversion.*

Due to the space constraints we provide the proof in the extended version of this paper [6]. However, intuitively this theorem holds because NPS_l bounds the maximum non-preemptive segment length of any job of task τ_l, which means that for any job $J_{l,j}$ and at any point in its execution $\rho \in [0, c_{l,j}]$, there exists a $\delta \leq NPS_l$ such that $\psi(J_{l,j}, \rho + \delta)$ (i.e., the job will be preempted no later than NPS_l time units after the beginning of a non-preemptive segment).

Based on Theorem 30, Table 1 states concrete instantiations of the four well-known preemption models for which we obtain concrete RTAs in the next section.

7 RTA Instantiation for FP and EDF

Finally, we put everything together to obtain concrete response-time bounds for EDF and FP via reductions to the sequential abstract model. Recall from Section 3 that we first must define the interface functions. As EDF and FP are both JLFP policies, we provide general definitions of $\mathcal{I}_\sigma$ and $\mathcal{W}_\sigma$ for a generic JLFP priority relation $\preceq : \mathbb{J} \times \mathbb{J} \to \mathbb{B}$.

Interference. Conceptually, a job $J_{i,j}$ incurs interference whenever it is delayed (i.e., prevented from being scheduled). Under a work-conserving JFLP policy, this occurs when either a higher- or equal-priority job preempts $J_{i,j}$, or when a lower-priority job causes priority inversion. In both cases, some job other than $J_{i,j}$ is scheduled, which results in the following trivial definition: $\mathcal{I}_\sigma(J_{i,j}, t) \triangleq \sigma(t) \neq J_{i,j}$.

Interfering Workload. The definition of $\mathcal{W}_\sigma(J_{i,j}, t)$ is slightly more involved. Naturally, it includes the total cost of all higher- or equal-priority jobs released at time t. *Additionally*, $J_{i,j}$ accrues one time unit of interfering workload if there is a priority inversion at time t (i.e., if $lps(J_{i,j}, t)$ holds) since no higher- or equal-priority workload is being "consumed" while a priority inversion persists.

$$\mathcal{W}_\sigma(J_{i,j}, t) \triangleq [\![lps(J_{i,j}, t)]\!]_{\mathbb{1}} + \sum \{ c_{h,k} \mid \forall J_{h,k} \in a(t) : J_{h,k} \succeq J_{i,j}, J_{i,j} \neq J_{h,k} \}$$

FP Interference Bound. Next, we need to provide an appropriate bound on interference. It is well-known [3, 7, 30, 46] (and formally shown in our Coq proof) that

$$IBF_i^{other}(A, \Delta) \triangleq B_i + \sum \{ RBF_{hp}(\Delta) \mid \forall \tau_{hp} : \tau_{hp} \neq \tau_i : \tau_{hp} \succeq \tau_i \}$$

bounds interference under FP, with B_i given by Theorem 30 for $LP_i \triangleq \{\tau_l \mid \tau_i \succ \tau_l\}$.

EDF Interference Bound. It is further known [17] (and verified in our Coq proof) that

$$IBF_i^{other}(A, \Delta) \triangleq B_i + \sum \{ RBF_o(\min\{A + \varepsilon + D_i - D_o,\ \Delta\}) \mid \forall \tau_o : \tau_o \neq \tau_i \}$$

bounds interference under EDF, with B_i given by Theorem 30 for $LP_i \triangleq \{\tau_l \mid D_o < D_i\}$.

To show that these reductions to the sequential abstract model are indeed correct, one needs to *prove* that Hypotheses 7, 8, and 22–24 are satisfied (which follows directly from properties of EDF, FP, and sporadic tasks). We omit these intuitively obvious proofs here for brevity, but stress that they form an integral part of the verified Coq proof.

As all hypotheses are satisfied, abstract sequential RTA (Theorem 27) yields the following verified general RTAs for *any* preemption model. By combining Theorems 31 and 32 below with the preemption models defined in Section 6, we obtain in total eight concrete, formally verified RTAs for FP and EDF.

▶ **Theorem 31.** *Given a preemption-model-compliant FP scheduler, a non-overloaded processor, and sequential, non-suspending tasks, if there exists an R such that $\forall A \in \mathcal{A}_i, \exists F$,*

$$\begin{aligned} A + F = B_i &+ (RBF_i(A + \varepsilon) - (C_i - RCT_i)) \\ &+ \sum \{ RBF_{hp}(A + F) \mid \forall \tau_{hp} : \tau_{hp} \neq \tau_i, \tau_{hp} \succeq \tau_i \} \end{aligned}$$

and $F + (C_i - RCT_i) \leq R$, where the search space is given by $\mathcal{A}_i \triangleq \{0\} \cup \{A < L \mid RBF_i(A) \neq RBF_i(A + \varepsilon)\}$, then the response-time of task τ_i is bounded by R.

Interestingly, the definition of the search space $\mathcal{A}_i$ (i.e., any offset A where RBF_i "steps") coincides with job releases in Lehoczky's critical-instant-based busy-window analysis [30]. The interference and self-interference bounds similarly match known results. We have thus independently obtained a proof of correctness of the well-known RTAs for FP, NP-FP, LP-FP, and FP-NPS scheduling [8, 46], notably without relying on a critical-instant argument. With the exception of FP scheduling [14], these bounds have not been formally verified before. In particular, Theorem 31 verifies Davis et al.'s revised CAN analysis [12]. For EDF, we obtain:

▶ **Theorem 32.** *Given a preemption-model-compliant EDF scheduler, a non-overloaded processor, and sequential, non-suspending tasks, if there exists an R such that $\forall A \in \mathcal{A}_i, \exists F$,*

$$A + F = B_i + (RBF_i(A + \varepsilon) - (C_i - RCT_i)) + \sum \{RBF_o(\min\{A + \varepsilon + D_i - D_o, A + F\}) \mid \forall \tau_o : \tau_o \neq \tau_i\}$$

and $F + (C_i - RCT_i) \leq R$, where the search space is given by $\mathcal{A}_i \triangleq \{0\} \cup \{A < L \mid RBF_i(A) \neq RBF_i(A + \varepsilon) \ \vee \ \exists \tau_o \in \tau, \tau_o \neq \tau_i \ \wedge \ RBF_o(A + D_i - D_o) \neq RBF_o(A + \varepsilon + D_i - D_o)\}$, then the response-time of task τ_i is bounded by R.

While similar bounds were previously known for fully preemptive EDF [19, 21, 41], to the best of our knowledge, no RTAs have yet been proposed in prior work for NP-EDF, LP-EDF, or EDF-NPS. Further, no RTA for EDF has previously been verified, nor was it known that RTAs for EDF and FP share exactly the same proof scheme: namely, abstract RTA.

8 Conclusion and Future Work

Paul K. Harten Jr., in his 1987 paper [34] introducing one of the first RTAs for FP scheduling [29, 34],[3] which he obtained through formal reasoning in a temporal logic, stated as his closing remark: "The problem of developing a mechanical aid for the construction of proofs of real-time properties is a far more difficult one, and beyond the capabilities of this author" [34].

With the benefit of over 30 years of progress in the area of interactive theorem proving, by building on powerful tools and frameworks such as Coq [42] and Coq's Mathematical Components library [32], we have obtained the first fully mechanized formalization and machine-checked proof of the busy-window principle, a general proof scheme for mechanized RTAs that we call abstract RTA. Abstract RTA allowed us to include RTAs for eight practically relevant scheduler and workload combinations in Prosa [10, 35], which demonstrates its utility as a powerful foundation and proof framework for the development of verified RTAs.

Many opportunities for future work and further generalization remain. First, our definition of *RBF* currently precludes an efficient analysis of multi-frame task models, but this limitation can be easily removed. Second, it would be useful to introduce RTAs with support for precedence constraints. In a similar direction, it would be interesting to hoist Fradet et al.'s work [15] on a generalized digraph model in Prosa on top of our abstract RTA. Third, in this paper we focus only on the issue of proving that an instantiation of abstract RTA provides *sound* results, which is in keeping with Prosa project's focus on question of safety [10]. However, the problem of establishing necessary and sufficient conditions for an RTA to be precise is interesting and worth considering. Last but not least, abstract RTA is sufficiently general to obtain a verified RTA for self-suspending tasks, but obtaining a reasonably tight interference bound for such tasks (i.e., instantiating IBF^{other}) remains a challenging problem.

[3] One of several independent, concurrent discoveries of the concept [2, 38].

References

1 AbstractRTA - ECRTS'20 Artifact Evaluation. URL: https://people.mpi-sws.org/~sbozhko/ECRTS20/AbstractRTA.html.

2 Neil C. Audsley, Alan Burns, Robert I. Davis, Ken Tindell, and Andy J. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems*, 8(2-3):173–198, 1995. doi:10.1007/BF01094342.

3 Neil C. Audsley, Alan Burns, Mike M. Richardson, Ken Tindell, and Andy J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993. doi:10.1049/sej.1993.0034.

4 Sanjoy K. Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *17th Euromicro Conference on Real-Time Systems, ECRTS 2005*, pages 137–144. IEEE Computer Society, 2005. doi:10.1109/ECRTS.2005.32.

5 Marko Bertogna and Sanjoy K. Baruah. Limited preemption EDF scheduling of sporadic task systems. *IEEE Trans. Industrial Informatics*, 6(4):579–591, 2010. doi:10.1109/TII.2010.2049654.

6 Sergey Bozhko and Björn B. Brandenburg. Abstract response-time analysis: A formal foundation for the busy-window principle (extended version). Technical Report MPI-SWS-2020-003, Max Planck Insitute for Software Systems, 2020. URL: https://www.mpi-sws.org/tr/2020-003.pdf.

7 Reinder J. Bril, Johan J. Lukkien, and Wim F. J. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited. In *19th Euromicro Conference on Real-Time Systems, ECRTS 2007*, pages 269–279. IEEE Computer Society, 2007. doi:10.1109/ECRTS.2007.38.

8 Reinder J. Bril, Johan J. Lukkien, and Wim F. J. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption. *Real-Time Systems*, 42(1-3):63–119, 2009. doi:10.1007/s11241-009-9071-z.

9 John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James Anderson, and Sanjoy K. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms, 2004.

10 Felipe Cerqueira, Felix Stutz, and Björn B. Brandenburg. PROSA: A case for readable mechanized schedulability analysis. In *28th Euromicro Conference on Real-Time Systems, ECRTS 2016*, pages 273–284. IEEE Computer Society, 2016. doi:10.1109/ECRTS.2016.28.

11 Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn B. Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil C. 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*, 55(1):144–207, 2019. doi:10.1007/s11241-018-9316-9.

12 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, 2007. doi:10.1007/s11241-007-9012-7.

13 Robert I. Davis, A. Zabos, and Alan Burns. Efficient exact schedulability tests for fixed priority real-time systems. *IEEE Trans. Computers*, 57(9):1261–1276, 2008. doi:10.1109/TC.2008.66.

14 Bruno Dutertre. Formal analysis of the priority ceiling protocol. In *21st IEEE Real-Time Systems Symposium, RTSS 2000*, pages 151–160. IEEE Computer Society, 2000. doi:10.1109/REAL.2000.896005.

15 Pascal Fradet, Xiaojie Guo, Jean-François Monin, and Sophie Quinton. A generalized digraph model for expressing dependencies. In *26th International Conference on Real-Time Networks and Systems, RTNS 2018*, pages 72–82. ACM, 2018. doi:10.1145/3273905.3273918.

16 Pascal Fradet, Xiaojie Guo, Jean-François Monin, and Sophie Quinton. CertiCAN: A tool for the Coq certification of CAN analysis results. In *25th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2019*, pages 182–191. IEEE, 2019. doi:10.1109/RTAS.2019.00023.

17 Laurent George, Nicolas Rivierre, and Marco Spuri. Preemptive and Non-Preemptive Real-Time UniProcessor Scheduling. Technical Report RR-2966, INRIA, 1996.

18 Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016*, pages 653–669. USENIX Association, 2016. URL: `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu`.

19 Nan Guan and Wang Yi. General and efficient response time analysis for EDF scheduling. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014*, pages 1–6. European Design and Automation Association, 2014. `doi:10.7873/DATE.2014.268`.

20 Xiaojie Guo, Maxime Lesourd, Mengqi Liu, Lionel Rieg, and Zhong Shao. Integrating formal schedulability analysis into a verified OS kernel. In *31st International Conference on Computer Aided Verification, CAV 2019*, pages 496–514. Springer, 2019. `doi:10.1007/978-3-030-25543-5_28`.

21 José C. Palencia Gutiérrez and Michael González Harbour. Offset-based response time analysis of distributed systems scheduled under EDF. In *15th Euromicro Conference on Real-Time Systems, ECRTS 2003*, pages 3–12. IEEE Computer Society, 2003. `doi:10.1109/EMRTS.2003.1212721`.

22 José C. Palencia Gutiérrez and Michael González Harbour. Response time analysis of EDF distributed real-time systems. *J. Embedded Computing*, 1(2):225–237, 2005. URL: `http://content.iospress.com/articles/journal-of-embedded-computing/jec00017`.

23 Zain Alabedin Haj Hammadeh, Rolf Ernst, Sophie Quinton, Rafik Henia, and Laurent Rioux. Bounding deadline misses in weakly-hard real-time systems with task dependencies. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017*, pages 584–589. IEEE, 2017. `doi:10.23919/DATE.2017.7927054`.

24 Michael González Harbour and José C. Palencia Gutiérrez. Response time analysis for tasks scheduled under EDF within fixed priorities. In *24th IEEE Real-Time Systems Symposium, RTSS 2003*, pages 200–209. IEEE Computer Society, 2003. `doi:10.1109/REAL.2003.1253267`.

25 Michael González Harbour, Mark H. Klein, and John P. Lehoczky. Fixed priority scheduling periodic tasks with varying execution priority. In *12th IEEE Real-Time Systems Symposium, RTSS 1991*, pages 116–128. IEEE Computer Society, 1991. `doi:10.1109/REAL.1991.160365`.

26 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):148–166, 2005.

27 Kevin Jeffay, Donald F. Stanat, and Charles U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *12th IEEE Real-Time Systems Symposium, RTSS 1991*, pages 129–139. IEEE Computer Society, 1991. `doi:10.1109/REAL.1991.160366`.

28 Mathai Joseph and Paritosh K. Pandya. Finding response times in a real-time system. *Comput. J.*, 29(5):390–395, 1986. `doi:10.1093/comjnl/29.5.390`.

29 Paul K. Harter Jr. Response times in level structured systems. Technical Report CU-CS-269-84, University of Colorado, 1984.

30 John P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *11th IEEE Real-Time Systems Symposium, RTSS 1990*, pages 201–209. IEEE Computer Society, 1990. `doi:10.1109/REAL.1990.128748`.

31 C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973. `doi:10.1145/321738.321743`.

32 Assia Mahboubi and Enrico Tassi. Mathematical components, 2017.

33 Geoffrey Nelissen, José Carlos Fonseca, Gurulingesh Raravi, and Vincent Nélis. Timing analysis of fixed priority self-suspending sporadic tasks. In *27th Euromicro Conference on Real-Time Systems, ECRTS 2015*, pages 80–89. IEEE Computer Society, 2015. `doi:10.1109/ECRTS.2015.15`.

34 Paul K. Harten Jr. Response times in level-structured systems. *ACM Trans. Comput. Syst.*, 5(3):232–248, 1987. doi:10.1145/24068.24069.

35 Prosa: The proven schedulability analysis repository, project web site. URL: http://prosa.mpi-sws.org.

36 Johannes Schlatow and Rolf Ernst. Response-time analysis for task chains in communicating threads. In *22th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2016*, pages 245–254. IEEE Computer Society, 2016. doi:10.1109/RTAS.2016.7461359.

37 Simon Schliecker, Jonas Rox, Matthias Ivers, and Rolf Ernst. Providing accurate event models for the analysis of heterogeneous multiprocessor systems. In *6th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2008*, pages 185–190. ACM, 2008. doi:10.1145/1450135.1450177.

38 Lui Sha, Tarek F. Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore P. Baker, Alan Burns, Giorgio C. Buttazzo, Marco Caccamo, John P. Lehoczky, and Aloysius K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2-3):101–155, 2004. doi:10.1023/B:TIME.0000045315.61234.1e.

39 Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990. doi:10.1109/12.57058.

40 Michael Short. Improved schedulability analysis of implicit deadline tasks under limited preemption EDF scheduling. In *16th IEEE Conference on Emerging Technologies & Factory Automation, ETFA 2011*, pages 1–8. IEEE, 2011. doi:10.1109/ETFA.2011.6059008.

41 Marco Spuri. Analysis of Deadline Scheduled Real-Time Systems. Technical Report RR-2772, INRIA, 1996.

42 The Coq proof assistant, project web site. URL: https://coq.inria.fr.

43 Ken Tindell, Alan Burns, and Andy J. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, 1994. doi:10.1007/BF01088593.

44 Yun Wang and Manas Saksena. Scheduling fixed-priority tasks with preemption threshold. In *6th International Workshop on Real-Time Computing and Applications Symposium, RTCSA 1999*, page 328. IEEE Computer Society, 1999. doi:10.1109/RTCSA.1999.811269.

45 Matthew Wilding. A machine-checked proof of the optimality of a real-time scheduling policy. In *10th International Conference on Computer Aided Verification, CAV 1998*, pages 369–378. Springer, 1998. doi:10.1007/BFb0028759.

46 Gang Yao, Giorgio C. Buttazzo, and Marko Bertogna. Feasibility analysis under fixed priority scheduling with limited preemptions. *Real-Time Systems*, 47(3):198–223, 2011. doi:10.1007/s11241-010-9113-6.

47 Man-Ki Yoon, Sibin Mohan, Chien-Ying Chen, and Lui Sha. TaskShuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems. In *22th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2016*, pages 111–122. IEEE Computer Society, 2016. doi:10.1109/RTAS.2016.7461362.

48 Xingyuan Zhang, Christian Urban, and Chunhan Wu. Priority inheritance protocol proved correct. *J. Autom. Reasoning*, 64(1):73–95, 2020. doi:10.1007/s10817-019-09511-5.

Analysis of Memory-Contention in Heterogeneous COTS MPSoCs

Mohamed Hassan
McMaster University, Hamilton, Canada
mohamed.hassan@mcmaster.ca

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

Abstract

Multiple-Processors Systems-on-Chip (MPSoCs) provide an appealing platform to execute Mixed Criticality Systems (MCS) with both time-sensitive critical tasks and performance-oriented non-critical tasks. Their heterogeneity with a variety of processing elements can address the conflicting requirements of those tasks. Nonetheless, the complex (and hence hard-to-analyze) architecture of Commercial-Off-The-Shelf (COTS) MPSoCs presents a challenge encumbering their adoption for MCS. In this paper, we propose a framework to analyze the memory contention in COTS MPSoCs and provide safe and tight bounds to the delays suffered by any critical task due to this contention. Unlike existing analyses, our solution is based on two main novel approaches. 1) It conducts a hybrid analysis that blends both request-level and task-level analyses into the same framework. 2) It leverages available knowledge about the types of memory requests of the task under analysis as well as contending tasks; specifically, we consider information that is already obtainable by applying existing static analysis tools to each task in isolation. Thanks to these novel techniques, our comparisons with the state-of-the art approaches show that the proposed analysis provides the tightest bounds across all evaluated access scenarios.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → System on a chip; Computer systems organization → Multicore architectures

Keywords and phrases DRAM, Memory, COTS, Multi-core, Real-Time, Embedded Systems, Analysis

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.23

1 Introduction

Unlike traditional embedded systems, Mixed Criticality Systems (MCS) such as those deployed in automotive and avionics embrace both safety-critical as well as high-performance tasks. Accordingly, low-end microcontrollers often used for traditional real-time embedded systems no longer meet the requirements of MCS. To address this challenge, researchers have explored the deployment of multi-core architectures (e.g. [8, 22, 27]). Among those architecture, Multiple-Processors Systems-on-Chip (MPSoCs) standout as a viable option to meet the various demands of MCS [12]. Their heterogeneity provides an opportunity to leverage different Processing Elements (PEs) to meet different tasks' requirements. For instance, real-time cores such as the ARM R5 in the Xilinx's Zynq Ultrascale+ [4] adopt a simpler architecture and hence are easier to analyze. Therefore, they can be used for time-sensitive safety-critical tasks. On the other hand, performance-oriented PEs such as GPUs and the ARM A-series cores can be utilized by high-performance tasks. That said, MPSoCs have their own challenges when deployed in MCS. Shared memory components such as on-chip caches and off-chip Dynamic Random Access Memories (DRAMs) create interference among PEs as they contend to access this shared memory. Memory interference can lead to a 300% increase in the total Worst-Case Execution Time (WCET) of a task in an

8-core system if a task spends only 10% of its execution time in memory accesses [28]. Similar trends were reported for the multi-core Freescale's P4080 platform [25]. In this paper, we focus on the analysis of memory contention delays in heterogeneous commercial-off-the-shelf (COTS) MPSoC platforms, where our goal is to derive a safe bound on these delays suffered by any critical task in a MCS executing on these platforms upon accessing the off-chip DRAM.

1.1 Related Work and Motivation

There exist several works whose goal is to manage interference due to contention while accessing the off-chip DRAM. Some of these works address this interference by entirely redesigning the memory controller to make DRAM accesses more predictable [7, 13, 17, 23, 34], which we refer the reader to the survey in [9] for their evaluation and comparison. Others propose operating system level solutions to alleviate the interference by partitioning DRAM banks among PEs [21, 26, 36], while controlling the maximum number of accesses issued by each PE [1, 2, 39].

Since this work focuses on analyzing DRAM interference in COTS architectures to provide safe memory delay bounds, the closest related efforts are [14, 20, 37]. The first two [20, 37] provide both job- and request-driven bounds, while the third [14] provides request-driven bounds only. *Job-driven* analysis utilizes information about total number of requests from competing cores to calculate the total Worst-Case Memory Delay (WCD) suffered by a core. *Request-driven* analysis, in contrast, derives a bound on the per-request WCD suffered by any single memory request. This bound is then multiplied by the total number of requests issued by the core to compute the total memory delay. Four observations about these efforts motivate our work. 1) Both [20] and [37] assume a specific platform with a particular architecture and OS configuration, and thus, the derived bounds are only limited to COTS platforms that follow these assumptions. 2) Although [14] addresses this limitation by exploring a wide set of COTS platforms, it only provides request-driven bounds. 3) Comparing both request- and job-driven analyses, we find that whichever one provides tighter bounds is dependent on the characteristics of running applications. In particular, it depends on the relative ratio between the number of requests of the core under analysis and the total number of interfering requests from competing cores. If the former is much smaller, then request-driven analysis will provide the tighter bound. On the other hand, if the latter is much smaller, then job-driven analysis will provide the tighter bound. Considering the minimum of both bounds as proposed in [20, 37] is certainly a viable approach. However, instead of conducting each analysis separately and then considering the smallest result, a hybrid approach that blends both analyses at a per-core basis can further tighten the bound. 4) All aforementioned works do not differentiate between different types of requests issued by cores such as reads vs writes, and DRAM row hits vs DRAM row conflict requests. Leveraging such information, as we show in this work, can significantly reduce the WCD and provide tighter bounds.

Motivated by these observations, this paper makes the following contributions.

1. It proposes an approach that blends both request- and job-driven analyses in the same framework. Both analyses are combined to form a single optimization problem. The solution to this problem provides a tighter, yet safe, bound on the cumulative memory delay suffered by requests of the core under analysis. We open-source the problem formulation that implements the analysis for the community to use and extend [1].

[1] `https://gitlab.com/FanusLab/memory-contention-analysis`

2. Unlike existing solutions, this framework leverages information, if available, about the requests issued by the core under analysis as well as interfering cores. Specifically, we consider the number of read and write requests, and the number of DRAM row hits and row conflicts issued by each task. This information can be obtained by analyzing all tasks in the system in isolation either statically using static analysis tools or experimentally. That said, we make no assumption about the times at which those requests are issued or their sequence patterns since this information is run-time dependent and is affected by the behavior of competing tasks, and hence, not possible to obtain by simply analyzing the tasks in isolation.
3. Contrary to existing job-analysis [20, 37], we cover a wide set of commodity COTS platforms. Namely, we consider the same 144 platform instances covered by [14].
4. Unlike [14], which provides bounds for only 81 out of those 144 platform instances and declares the remaining 63 instances unbounded, the proposed framework is able to safely bound all 144 instances thanks to its hybrid approach using both request- and job-driven analyses.
5. We conduct a comprehensive evaluation to compare with both job-driven analyses in [20,37] as well as request-driven analyses in [14, 20, 37] using a variety of interference scenarios. This comparison shows that the proposed approach achieves tighter bounds under all scenarios. The proposed approach provides 24% – 42% tighter bounds compared to [37], 23% – 21× tighter bound compared to [20], and a minimum of 4% tighter bound compared to [14], while it is able to provide bounds for scenarios that are deemed unbounded by [14] as aforementioned.

2 Background

2.1 Background on DRAM

DRAM consists of cells that are grouped in banks. Each bank resembles an array of columns and rows, and has a row buffer that holds the most recently accessed row in that bank. An on-chip Memory Controller (MC) manages accesses to the DRAM by issuing DRAM commands on the command bus. Namely, we have three main commands: ACT, CAS, and PRE. 1) If the requested row is already available in the row buffer, the request consists of only a CAS command that executes the actual read (R) or write (W) operation. We call the request in this case an *open* request. 2) If the requested bank is idle (i.e., does not have an activated row in the buffer), the MC issues an ACT command first to activate the row, followed by a CAS command. 3) If the requested row is different from the activated row in the row buffer (a *bank conflict*), the MC issues all three commands: PRE to precharge the old row, ACT to activate the requested row, and CAS to read/write. We call the request that suffers a bank conflict, a *close* request. The MC is able to issue only one command at any single cycle to the DRAM. Therefore, if there are more than one command that are ready to be sent to the DRAM at the same cycle, we say that there is a *command bus conflict*. Only one of them will be issued by the MC, while the others are delayed to subsequent cycles.

The JEDEC DRAM standard [18] defines a set of timing constraints on the three commands that must be satisfied by all MC designs; the value of each constraint depends on the specific DRAM device type and speed. Table 1 exemplifies with constraints from a single-rank DDR3 device; it also shows the value of the constraints for the particular device speed we use in the evaluation. It is important to note that the proposed analysis is not specific to this particular device and can be applied to any DRAM. For DDR4 devices, the bank-group timing constraints need to be also considered in addition to the ones in Table 1;

Table 1 JEDEC timing constraints for DDR3-1333H [18].

(a) Intra-bank (conflict) constraints.

Parameters	Description	cycles
$tRCD$	ACT to CAS delay	9
tRL	RD to Data Start	9
tRP	PRE to ACT Delay	9
tWL	WR to Data Start	8
$tRAS$	ACT to PRE Delay	24
tRC	ACT to ACT (same bank)	33
tWR	Data End of WR to PRE	10
$tRTP$	Read to PRE Delay	5

(b) Inter-bank constraints.

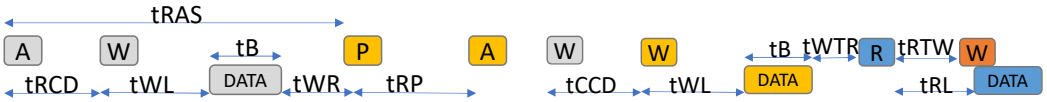
Parameters	Description	cycles
Inter-bank CAS constraints		
$tCCD$	CAS to CAS delay	4
$tRTW$	RD to WR Delay	6
$tWTR$	WR to RD Delay	5
Inter-bank ACT constraints		
$tRRD$	ACT to ACT (diff bank in same rank)	4
$tFAW$	Four bank activation window	20

however, a similar analysis can be applied. Each constraint represents the minimum number of clock cycles that must elapse between the transmission of a command or data and a successive command or data; with the exception of $tFAW$, which represents the minimum distance every four, rather than two, successive ACT commands. We distinguish between two types of constraints: *intra-bank constraints* are applied between data/commands issued to the same bank, while *inter-bank ACT/CAS constraints* are applied between data/commands of the same type (ACT or CAS) issued to any bank. Correspondingly, we shall say that a request causes intra-bank delay on another one if it triggers intra-bank constraints; or ACT/CAS delay if it triggers inter-bank ACT/CAS constraints. Note that there are no inter-bank constraints for PRE commands; however, due to the effect of command bus conflicts, a PRE command can still cause PRE delay on another PRE command. For ease of exposition, Figure 1 depicts an example of intra-bank constraints (Figure 1a) as well as inter-bank constraints (Figure 1b). Note that when considering two consecutive requests, intra-bank constraints can affect the latency of the second request only in the case of a bank conflict: if the two requests access the same bank without conflict, then the second request must be open and only the inter-bank CAS constraints apply. Hence, we also refer to intra-bank constraints and delay as *conflict constraints/delay*. A command (or request) is denoted as *intra-* or *(inter-)ready* when it meets all its intra- (or inter-)bank constraints. A command cannot be issued before it is both intra- and inter-ready. DRAM cells have to be periodically refreshed to prevent data leakage through issuing REF (*refresh*) commands. Refresh delays can be often neglected compared to other delays [20]. It can also be added as an extra delay term to the execution time of a task using existing methods [3, 35]. Accordingly and similar to previous works [14, 20, 37], we do not account for the refresh delay.

Arbitration. Requests are first queued into per-bank queues. Then two-level arbitration is deployed as follows: 1) *Intra-bank arbitration* is implemented between requests of the same bank. This usually uses a First Ready-First Come First Serve (FR-FCFS) scheduling mechansim [20,24,31]. FR-FCFS prioritizes open requests, which target data already available in the row buffer over close requests. 2) *Inter-bank arbitration*: the MC deploys a Round Robin (RR) mechanism to arbitrate among intra-ready commands at the head of the bank queues [6, 14, 16, 32, 33, 36]. In case of a command bus conflict, we assume the following priority order is enforced: CAS, ACT, and then PRE such that CAS have the highest priority upon bus conflicts, while PRE commands have the least. This is known as column-first scheduling and it targets to reduce latency [24, 31].

2.2 System Model and Platform Instances

We consider a system with P PEs, where some of these PEs are *critical* (P_{cr}) and others are non-critical (P_{ncr}) such that $P = P_{cr} + P_{ncr}$. PEs share write-back write-allocate Last-Level Cache (LLC); hence, writes to DRAM occurs only because of cache eviction of



(a) Intra-bank constraints. **(b)** Inter-bank constraints.

Figure 1 DRAM timing constraints example. tB is the data transfer time (4 cycles for a burst length of 8).

dirty cache blocks. We find this to be the common policy deployed in COTS architectures and it is also adopted by previous related works [14, 37]. Requests that miss in the LLC are sent to the DRAM. We consider a single-channel single-rank DRAM subsystem with N_B banks. Similar to related work [14, 20, 37], we do not make any assumption about the computation and memory access patterns of the PE under analysis, or any of the interfering PEs. Nonetheless, as we detail in Section 3, our goal is to improve upon existing DRAM analyses, and in particular the framework in [14], by incorporating knowledge about the number of requests produced by all PEs in the system. The overall behavior of the memory subsystem depends on both the MC configuration, as well as on the characteristics of PEs that generate memory requests. To this end, the work in [14] defined a set of fundamental platform features that affect the delay analysis; the combination of the features, specified as a tuple $\langle wb, thr, pr, breorder, pipe, part \rangle$, characterizes one of 144 possible platform instances. Since we reuse the same features in our analysis, here we summarize their values and corresponding behavior.

Read-Write Arbitration. $wb \in \{0, 1\}$. If $wb = 0$, the MC assigns the same priority for both reads and writes. If $wb = 1$, the MC employs write batching, where it prioritizes reads while queuing writes in a dedicated write buffer. We consider the same watermarking implementation discussed in related work [14, 30, 37]: the MC services a batch of W_{btch} writes when the number of buffered writes exceeds a given threshold.

First-Ready Threshold. $thr \in \{0, 1\}$. FR-FCFS arbitration reorders intra-ready requests over non intra-ready ones targeting the same bank. If $thr = 1$, the MC deploys a thresholding mechanism [15, 20] to avoid starvation, where at most N_{thr} intra-ready requests can be re-ordered ahead of any other request targeting the same bank. If $thr = 0$, then no reordering threshold is implemented.

PE Prioritization. $pr \in \{0, 1\}$. If $pr = 1$, the MC prioritizes requests of critical PEs over non-critical ones [15, 30]. If $pr = 0$, all PEs are treated equally.

Inter-bank Reordering. $breorder \in \{0, 1\}$. As discussed, the MC employs a RR arbiter which selects among banks with intra-ready commands. If the command of the highest priority bank is not inter-ready, then the MC can reorder ahead of it the command of the next highest priority bank (based on the RR order) with a ready command. If $breorder = 1$, then the reordered commands can be of the same type; in particular, a W command can be reordered ahead of a R command of vice-versa. As shown in [14], this can lead to a situation where an unbounded number of CAS commands is reordered ahead of another CAS command. To avoid starvation, we also consider $breorder = 0$, where inter-bank reordering is allowed only for commands of different type.

Table 2 System model symbols.

Symbol	Description	Instances	Symbol	Description	Instances
P	Number of PEs	all	N_{Bcr}	Number of banks assigned to critical PEs	$part = PartAll$
P_{cr}	Number of critical PEs	all	N_{Bncr}	Number of banks assigned to non-critical PEs	$part = PartAll$
P_{ncr}	Number of non-critical PEs	all	N_{thr}	Intra-bank reorder threshold	$thr = 1$
N_B	Number of DRAM banks	all	W_{btch}	Write batch length	$wb = 1$
N_{B_p}	Number of DRAM banks assigned to the p-th PE	all	PR	Number of outstanding requests	$pipe \neq IO$

PE pipeline architecture. $pipe \in \{IO, IOCr, OOO\}$. If $pipe = IO$, all PEs are in order and can generate only one pending memory request at a time. If $pipe = OOO$, all PEs are out-of-order, and we let PR to denote the maximum number of outstanding requests in the MC queue for each PE. If $pipe = IOCr$, then critical PEs are in order, while non-critical ones are out-of-order [4].

Bank Partitioning. $part \in \{PartAll, PartCr, NoPart\}$. Several previous works (e.g. [7, 10, 17, 35]) have proposed DRAM bank partitioning, where banks are partitioned among PEs, to reduce bank conflicts between PEs. Partitioning is typically implemented by manipulating the page table in the OS [21, 26, 36]. If $part = PartAll$, then partitioning is applied to all PEs. If $part = PartCr$, then partitioning is applied only to critical PEs, while non-critical PEs can use all banks. If $part = NoPart$, no partitioning is used.

Table 2 further summarizes the parameters associated with each platform instance. In Table 2, N_{B_p} depends on the applies bank partitioning scheme. For instance, if we have $NB = 8$ and $P_{cr} = P_{ncr} = 2$, under $NoPart$: $N_{B_p} = NB = 8$ for all PEs, for $PartAll$: $N_{B_p} = 8/4 = 2$, while for $PartCr$: $N_{B_p} = 8/2 = 4$ for critical PEs and $N_{B_p} = 8$ for non-critical PEs. $N_{B_{cr}}$ and $N_{B_{ncr}}$ apply only under $PartAll$ since it is the only partitioning scheme, where critical and non-critical PEs do not share banks; hence, each bank can be indicated as either critical or non-critical.

3 Preliminaries

We are interested in computing a bound on the cumulative delay $\Delta(t)$ suffered by requests generated by one or more tasks running on a critical PE under analysis PE_i in an interval of time t. Specifically, we bound the *processing* delay of requests of PE_i, that is, the extra delay suffered after the request arrives at the head of the request queue for PE_i. For an out-of-order architecture, we do not consider *queueing* delay due to a request being queued after other requests of PE_i itself; such delay depends on the exact time at which requests are issued and should be handled while statically analyzing PE_i. Let e be the WCET of the task(s) in isolation, that is, while the other PEs in the system are inactive and do not cause any delay. Further assume that delay is composable, that is, $e + \Delta(t)$ is an upper bound to the execution time of the task(s) when suffering a cumulative delay $\Delta(t)$ (note that even if the PE is not timing compositional, the delay can still be composed by computing an appropriate upper-bound to e as described in [11]). Then the execution time $\bar{e}$ of the task(s) can be bounded by the recurrence: $\bar{e} = e + \Delta(\bar{e})$.

We assume that either through static analysis or measurements, it is possible to formulate bounds on the number of requests that the task(s) produces in isolation (the *original schedule* of memory requests). The number and type of such constraints depends on the capability of the analysis or measurement framework. A coarse method might be only capable of deriving the maximum number of requests $H(i)$, while an improved method might be able to bound the maximum number $HR(i)$ and $HW(i)$ of read and write requests, respectively. There also exist analyses [5] that are able to differentiate between open and close requests,

hence deriving bounds $HR^o(i), HR^c(i)$ on the number of open and close read requests, and similarly $HW^o(i), HW^c(i)$ for write requests. Note that in this case it might hold $HR^o(i) + HR^c(i) > HR(i)$, as the analysis might not be able to classify as open or close some of the requests. Hence, to provide a general analysis, we will consider all presented terms, with the assumption that coarse estimation methods might result in a value of $+\infty$ for some of the terms (i.e., they cannot provide a useful bound).

We are now interested in determining the number of requests of each type produced by the task(s) when running together with the other $P-1$ interfering PEs (the *interfered schedule*). For simplicity, we will assume that the behavior of PE_i, in terms of produced memory requests, is not affected by interference; note that if the PE uses a cache, this implies that the cache must be private or partitioned. Hence, the bounds on the number of reads and write requests still hold. However, the type of each request (open or close) depends on the state of the device, which can be affected by other PEs. Therefore, with no loss of generality, let $R^o(i), R^c(i), W^o(i), W^c(i)$ to denote the number of open/close read and write requests for PE_i in the interfered schedule. We then have:

$$\text{if } wb = 0 : R^o(i) \leq HR^o(i), W^o(i) \leq HW^o(i) \tag{1}$$

$$\text{if } \mathit{PartAll} \text{ and } wb = 0 \ : R^c(i) \leq HR^c(i) \tag{2}$$

$$\text{if } \mathit{PartAll} \text{ and } wb = 0 \ : W^c(i) \leq HW^c(i) \tag{3}$$

$$\text{if } \mathit{PartAll} \text{ and } wb = 0 \ : R^c(i) + W^c(i) \leq HR^c(i) + HW^c(i) \tag{4}$$

$$R^c(i) + R^o(i) \leq HR(i) \tag{5}$$

$$W^c(i) + W^o(i) \leq HW(i) \tag{6}$$

$$R^c(i) + R^o(i) + W^c(i) + W^o(i) \leq H(i) \tag{7}$$

Equations 5-7 bound the number of reads, writes and all requests, respectively; based on our assumptions, they are always valid. Equations 1-4 bound the number of open and close requests, and instead depend on the platform features. If the platform employs write batching, then we make no assumption on the number of open or close requests: write requests produced by other PEs can change the time and order in which batches are issued, which in turn can change the type of any request. If $part = PartCr$ or $NoPart$, then PE_i shares banks with some other PE. In this case, bank conflicts can turn requests that were open in isolation into close requests. Hence, in this case we cannot consider the bounds on close requests (Equations 2-4), while the bound on open requests (Equation 1) still holds. Finally, we discuss how to bound the number of requests for an interfering PE_p with $p \neq i$. If PE_p is a core executing a known task set, then the same approach in Equation 1-7 can be employed, where $H(p)$ and related terms express the maximum number of requests produced by the task set in any interval of length t. In particular, related work [20] shows how to compute $H(p)$ assuming a partitioned, fixed priority scheduling scheme. Other work assumes memory regulation [38], where PE_p is assigned a memory budget Q_p, and cannot issue more than Q_p requests in a regulation interval of length P. In this case, assuming that the window of time t starts synchronously with the regulation interval, we simply compute the value in Equation 8. Note that for an out-of-order PE, term PR is added to account for requests that might be queued at the memory controller before the beginning of the first regulation period.

$$H(p) = \lceil t/P \rceil \cdot Q_p + \begin{cases} 0 \text{ if } IO \text{ or } (p \text{ is } cr \text{ and } IOCr) \\ PR \text{ otherwise} \end{cases} \tag{8}$$

4 Memory Delay Analysis

In this section, we show how to compute a cumulative WCD bound Δ for the requests of critical core under analysis PE_i. In details, we consider the delay due to additional timing constraints, as well as bus conflicts, caused by either interfering requests of other PEs, or previous requests of PE_i itself. For $wb = 0$, the WCD bound includes the delay suffered by both reads and writes requests of PE_i, which we call the *critical requests*. For $wb = 1$, we only consider delay suffered by read requests, as under write batching write requests of PE_i itself are queued so that they do not delay program execution; however, in this case we consider the delay caused by writes of PE_i on the critical read requests of PE_i. To facilitate accounting for the various timing constraints, we will obtain Δ by determining which delay is caused by each request (either conflict, PRE, ACT or CAS), and then adding together three corresponding *delay terms*: L^{Conf} represents the cumulative delay due to conflict constraints; while L^{ACT} and L^{CAS} represent the cumulative ACT and CAS delays. Note that we do not define a delay term for PRE because, as we prove in Section 4.4, in the worst-case interference pattern such delay is zero. We first categorize the effect of the interfering requests of other PE_i in Section 4.1, and then discuss the effect of self-interference caused by previous requests of PE_i in Section 4.2. Finally, Sections 4.3 and Sections 4.4 detail how to compute the delay terms.

4.1 Interfering Requests

We start with a set of observations, based on the timing constraints in Section 2.1, to help classifying interfering requests based on which type of delay they cause.

▶ **Observation 1.** *Consider two requests targeting different bank. If both requests are close, then the first one can cause PRE, ACT and CAS delay to the second one; otherwise, it can only cause CAS delay.*

Note that Observation 1 holds because in order to suffer PRE or ACT delay, both the delaying and the delayed request must issue a PRE/ACT command.

▶ **Observation 2.** *Consider two requests targeting the same bank. If the second request is close, then the first one can cause conflict delay to it; otherwise, it can only cause CAS delay. The conflict delay is larger than the CAS delay.*

▶ **Observation 3.** *Conflict constraints are larger than PRE, ACT and CAS constraints. Hence, when two consecutive requests can target either the same or different banks, the delay suffered by the second request is larger or equal if they target the same bank compared to different banks (specifically, equal if the request is open, and larger if close).*

We next discuss how to determine the number of interfering requests for each delay term. Based on Observation 3, we can maximize Δ by assuming that all interfering requests that can target the same bank as a request of PE_i do so. Therefore, when counting interfering requests, we classify them between *intra-bank requests*, which can delay each other and critical requests of PE_i on the same bank based on Observation 2, and *inter-bank requests*, which can delay intra-bank requests based on Observation 1; specifically, we next discuss how to systematically divide the interfering requests into several *interference components*.

(1) Intra-bank conflict requests: $R^{Conf,c}$, $W^{Conf,c}$ are the number of read and write interfering requests targeting the same bank as any one request of PE_i, and which are serviced ahead of that request because they arrived before it. As noted in Section 3, in

this case we can make no assumption on the type of the requests. Hence, we assume the worst case where all such requests, as well as the request of PE_i, are close [2].

(2) Intra-bank reorder requests: $R^{Reorder,o}$, $W^{Reorder,o}$ are the numbers of interfering requests of other PEs targeting the same bank as any one request of PE_i, and which arrived after that request but are reordered ahead of it due to first-ready arbitration. Since the interfering requests are ready, they must be open requests, while the request of PE_i must be close.

(3) Inter-bank-close requests: $R_c^{InterB,c}$, $R_c^{InterB,o}$, $W_c^{InterB,c}$, $W_c^{InterB,o}$ are interfering requests (read/write and open/close, based on the superscript) that target a different bank than any one request of PE_i, and delay close requests targeting the same bank as PE_i: the $R^c(i) + W^c(i)$ close requests of PE_i itself, and the $R^{Conf,c}/W^{Conf,c}$ conflict requests. By Observation 1, the open requests $R_c^{InterB,o}$ and $W_c^{InterB,o}$ contribute CAS delay, while the close requests $R_c^{InterB,c}$ and $W_c^{InterB,c}$ contribute to PRE, ACT and CAS delay.

(4) Inter-bank-open requests: R_o^{InterB}, W_o^{InterB} are interfering requests (R and W) that target a different bank than any one request of PE_i, and delay open requests targeting the same bank as PE_i: the $R^o(i) + W^o(i)$ open requests of PE_i itself, and the $R^{Reorder,o}$, $W^{Reorder,o}$ reorder requests. By Observation 1, these $R_o^{InterB} + W_o^{InterB}$ requests can only contribute CAS delay.

Note that for instances with $wb = 1$, the intra- and inter-bank components only include read requests, while write requests are considered in the write batching component. Hence we impose:

$$\text{if } wb : W^{Conf,c} = W^{Reorder,o} = W_c^{InterB,c} = W_c^{InterB,o} = W_o^{InterB} = 0 \tag{9}$$

(5) Write batching requests: For instances with $wb = 1$, W^{WB} represents the total number of write requests; contrarily to the previous components, W^{WB} includes both interfering write requests, as well as write requests of PE_i itself, since write requests of all PEs are reordered and issued in write batches. As again noted in Section 3, when $wb = 1$ we can make no assumption on the type (open or close) of write requests executed in write batches, hence we consider a worst case situation where all requests are close and target the same bank, thus contributing to L^{Conf}.

The described interference components depend on the total number of requests produced by each interfering PE, as well as on the platform instance. We detail how to bound the interference components in Section 5; while in the rest of this section we focus on computing the latency terms assuming that the values of the interference components are known. Finally, Section 5.6 shows that we can compute a bound on Δ by solving a Linear Programming (LP) problem. To facilitate the reader, Table 3 summarizes all variables used in the optimization problem [3].

It remains to summarize the impact of the intra- and inter-bank interfering requests on the delay terms. For intra-bank interfering requests, based on Observation 2 let x^{Conf} to denote the number of conflict delays triggered by the interfering requests, and x^{CAS} to

[2] Note that if PE_i shares a bank with another interfering PE, then Equations 2, 3 do not apply; hence the optimization problem can set $R^o(i) = W^o(i) = 0$ and maximize the number of close request $R^c(i), W^c(i)$ based on Equations 5, 6.

[3] Note that $H(i), HR(i), HW(i), HR^c(i), HR^o(i), HW^c(i), HR^o(i)$ for all cores, as introduced in Section 3, do not appear in the table because they are inputs to the analysis, hence constants in the LP problem.

Table 3 Optimization problem variables. We use L to denote a delay term; N to denote number of requests; R/W to denote number of read/write requests; and x to denote number of constraints. If a variable has the (p) index, then it refers to a specific PE_p. Otherwise, the variable indicates values over all PEs. For superscripts, c/o denotes the type (open or close) of the requests; for inter-bank requests, the subscript c/o denotes the type of the following intra-bank request.

Symbol	Interfering Direction	Interfering Request Type	Interfered Request Type (or PE ID)	Description	Delay
$R^{o}(i)$ ($W^{o}(i)$)	R (W)	Open	PE_i	Total number of open reads from PE_i	
$R^{c}(i)$ ($W^{c}(i)$)	R (W)	Close	PE_i	Total number of close reads from PE_i	
Self Interference Component					
$R^{OtC}(i)$ ($W^{OtC}(i)$)	R (W)	-	PE_i	Requests that were open and became close due to interference.	L^{Self}
$R^{CAS}(i)$ ($W^{CAS}(i)$)	R (W)	-	PE_i	Requests that cause self CAS delay	
$R^{Conf}(i)$ ($W^{Conf}(i)$)	R (W)	-	PE_i	Requests that cause extra self conflict delay.	
$N^{None}(i)$	R or W	-	PE_i	Requests that cause no extra self conflict delay.	
$N^{ACT}(i)$	R or W	-	PE_i	Close requests targeting different banks.	
$N^{ACT,a}(i)$	R or W	-	PE_i	Requests from $N^{ACT}(i)$ that were originally close.	
$N^{ACT,b}(i)$	R or W	-	PE_i	Requests from $N^{ACT}(i)$ that were originally open.	
Intra-Bank Conflict Requests					
$R^{Conf,c}[(p)]$ ($W^{Conf,c}[(p)]$)	R (W)	Close	Close	Requests causing conflict interference.	L^{Conf}
x^{Conf}	R or W	Close	Close	Total number of triggered conflict delays.	
Intra-Bank Reorder Requests					
$R^{Reorder,o}[(p)]$ ($W^{Reorder,o}[(p)]$)	R (W)	Open	Close	Number of requests causing intra-bank reorder interference.	L^{CAS}
x^{CAS}	R or W	Open or Close	Open or Close	Total number of triggered CAS delays.	
Inter-Bank-Close Requests					
$N_{reqs,c}$	R or W	–	Close	Number of interfered requests for inter-bank-close component.	L^{ACT}, L^{CAS}
$R_c^{InterB,c}[(p)]$ ($W_c^{InterB,c}[(p)]$)	R (W)	Close	Close	Number of requests that cause inter-bank interference on PE_i's close requests or any of the $N_{reqs,c}$ requests. They interfere either on ACT (N_{ACT}^{InterB}), CAS read ($R_{CAS,c}^{InterB}$), or CAS write ($W_{CAS,c}^{InterB}$) commands.	
$R_c^{InterB,o}[(p)]$ ($W_c^{InterB,o}[(p)]$)	R (W)	Open	Close		
$N_{ACT,c}^{InterB,c}$	R or W	Close	Close		
$R_{CAS,c}^{InterB}$ ($W_{CAS,c}^{InterB}$)	R (W)	Close or Open	Close		
Inter-Bank-Open Requests					
$N_{reqs,o}$	R or W	–	Open	Number of interfered requests for inter-bank-open component.	L^{CAS}
$R_o^{InterB}[(p)]$ ($W_o^{InterB}[(p)]$)	R (W)	Close or Open	Open	Number of interfering requests that cause inter-bank interference on PE_i's open requests or any of the $N_{reqs,o}$ requests .	
Auxiliary CAS Delay Variables					
R_{CAS}^{InterB} (W_{CAS}^{InterB})	R (W)	Open or Close	Open or Close	Requests from other PEs targeting other banks and causing inter-bank CAS delays.	L^{CAS}
$x^{CAS,RW}$ ($x^{CAS,WR}$)	R (W)	Open or Close	Open or Close	Total number of requests causing a R-to-W (W-to-R) CAS delays.	
Write Batching Requests					
$W^{btch}[(p)]$	W	Close or Open	–	Number of interfering write requests that arrive while no critical request is active.	L^{WB}
$W^{before}[(p)]$	W	Close or Open	–	Number of interfering write requests that arrive while a critical request is active,	
$W^{after}[(p)]$	W	Close or Open	–	and are executed before (after) the critical request	

denote the number of triggered CAS delays. We can then bound the total number of delays $x^{Conf} + x^{CAS}$ based on the total number of intra-bank interfering requests, and we can bound x^{Conf} based on the number of close requests targeting a same bank as PE_i: the conflict requests, and the close critical requests of PE_i:

$$x^{Conf} + x^{CAS} \leq R^{Conf,c} + W^{Conf,c} + R^{Reorder,o} + W^{Reorder,o} \tag{10}$$

$$x^{Conf} \leq R^{Conf,c} + W^{Conf,c} + R^{c}(i) + (1 - wb) \cdot W^{c}(i). \tag{11}$$

Note that we multiply the write requests of PE_i by $(1 - wb)$ since they are not critical if $wb = 1$. Instead, in the $wb = 1$ case, the conflict delays caused by the W^{WB} requests in write batches will be directly accounted for in the L^{Conf} term in Section 4.3.

For inter-bank interfering requests, $R_c^{InterB,o}$, $W_c^{InterB,o}$, R_o^{InterB} and W_o^{InterB} only induce CAS delays, as previously explained. To bound the cumulative delay induced by the $R_c^{InterB,c}$ and $W_c^{InterB,c}$ requests, we employ the following pipeline theorem from [37]:

▶ **Theorem 1** (Theorem 1 in [37]). *The delay caused by an interfering request to a request under analysis, where the two requests target different banks, is upper bounded by the delay caused by one interfering command on the same command of the request under analysis, i.e., either the PRE delay, or the ACT delay, or the CAS delay.*

Note that in Section 4.4 we will prove that the PRE delay is always less than the ACT delay. Hence, to maximize the bound on Δ, it suffices to assume that based on Theorem 1, each request in $R_c^{InterB,c}$ and $W_c^{InterB,c}$ can cause either ACT or CAS delay. We thus introduce terms $N_{ACT,c}^{InterB,c}, R_{CAS,c}^{InterB,c}, W_{CAS,c}^{InterB,c}$ to denote the number of requests (possibly

Figure 2 Self interference example.

distinguishing between R and W direction) that cause ACT and CAS delay, respectively, obtaining the following constraints:

$$N_{ACT,c}^{InterB,c} + R_{CAS,c}^{InterB,c} + W_{CAS,c}^{InterB,c} \leq R_c^{InterB,c} + W_c^{InterB,c}, \tag{12}$$

$$N_{ACT,c}^{InterB,c} + R_{CAS,c}^{InterB,c} \leq R_c^{InterB,c}, \tag{13}$$

$$N_{ACT,c}^{InterB,c} + W_{CAS,c}^{InterB,c} \leq W_c^{InterB,c}. \tag{14}$$

Finally, we use R_{CAS}^{InterB} (W_{CAS}^{InterB}) to denote the total number of reads (writes) from other PEs targeting other banks and causing inter-bank CAS delays. Hence, we get:

$$R_{CAS}^{InterB} = R_{CAS,c}^{InterB,c} + R_o^{InterB} + R_c^{InterB,o}, \tag{15}$$

$$W_{CAS}^{InterB} = W_{CAS,c}^{InterB,c} + W_o^{InterB} + W_c^{InterB,o}. \tag{16}$$

4.2 Self-Interference

Section 4.1 summarized the delay caused by interfering requests in terms of the timing constraints and bus conflicts induced by such requests. However, when two critical requests of PE_i are executed back-to-back in the original schedule, the first request of PE_i can induce further delays on any request that interferes with the second request of PE_i itself; ignoring such *self-interference* effects leads to an unsafe bound. Consider the example in Figure 2. Originally, PE_i issued two consecutive open R requests to the same bank; the minimum distance between the two requests, based on their CAS commands, is equal to the CAS-to-CAS constraint $tCCD$. In the interfered schedule, one W request of another core is interleaved between the two requests of PE_i; as a consequence, the distance between the two requests becomes equal to $tRTW + tWL + tB + tWTR$. Hence, the added delay is $tRTW + tWL + tB + tWTR - tCCD$, which is larger than the maximum delay $tWL + tB + tWTR$ of a single CAS.

To produce a safe delay bound, we thus proceed as follows: we carefully analyze each scenario (Cases (1a)-(3) below) involving two consecutive critical requests of PE_i, and whenever we found that the effect of self-interference is non-zero, we handle it by adding an additional delay term to the analysis (in the case of the example in Figure 2, to L^{CAS}), and subtracting the minimum distance between the requests in the original schedule ($tCCD$ in the example). When analyzing the scenarios, it is important to keep in mind, as discussed in Section 3, that requests of PE_i that were open in the original schedule can become close in the interfered schedule if write batching is enabled or PE_i shares banks with some other PE. Let $R^{OtC}(i)$ and $W^{OtC}(i)$ be upper bounds on the number of such R and W open-to-close requests; since $HR^o(i), HW^o(i)$ represent open requests in the original schedule, and $R^o(i), W^o(i)$ in the interfered schedule, it must hold:

$$R^{OtC}(i) \leq HR^o(i) - R^o(i), \tag{17}$$

$$W^{OtC}(i) \leq HW^o(i) - W^o(i), \tag{18}$$

$$\text{if } PartAll \text{ and } wb = 0 : R^{OtC}(i) = W^{OtC}(i) = 0. \tag{19}$$

- Case (1a) and (1b): the two requests of PE_i target the same bank, and the second one is close in the interfered schedule. In this case, the second request could be delayed by other close conflict requests in the interfered schedule, which could be in turn delayed by a conflict delay due to the first request of PE_i. However, if the second request of PE_i was also close in the original schedule (Case (1a)), then the minimum distance between the two requests in the original schedule is equal to the same conflict delay; hence, in this case self-interference does not add any extra delay. If instead the second request was open in the original schedule (Case (1b), meaning it is an open-to-close request), then the minimum distance in the original schedule could be $tCCD$; hence, to produce a safe bound, in this case we add one conflict delay to L^{conf}, and subtract $tCCD$ from the WCD Δ. We let $R^{Conf}(i), W^{Conf}(i)$ to denote the number of R and W requests for Case (1b), and $N^{None}(i)$ to denote requests that do not add any extra delay as per Case (1a).
- Case (2a) and (2b): assume that Case (1a), (1b) do not apply (that is, the requests target different banks and/or the second request is open in the interfered schedule). Then, it can still be possible for the first request of PE_i to cause either PRE, CAS or ACT delay on one or more interfering requests, which in turn cause the same type of delay to the second request of PE_i. Case (3), which we represented in Figure 2, covers the CAS delay; Case (2a) and (2b) cover the PRE and ACT delay. Since only close requests can cause or suffer PRE/ACT delay, it follows that for Case (2a), (2b) the two requests of PE_i must be close in the interfered schedule. Therefore, by assumption they must target different banks. The minimum distance between them is either the ACT-to-ACT constraint $tRRD$ if both were close in the original schedule (Case (2a)), or $tCCD$ if at least one was open (Case (2b), the request is open-to-close). As Section 4.1 mentions and Section 4.4 proves, the ACT delay is larger than the PRE delay; hence, for these cases we add an ACT term to L^{ACT} and subtract either $tRRD$ (2a) or $tCCD$ (2b). We use $N^{ACT,a}(i)$ and $N^{ACT,b}(i)$ for the number of requests added to L^{ACT} in Case (2a) and (2b), respectively, and $N^{ACT}(i)$ for their sum.

We can then bound the self-interference terms for Cases (1a), (1b), (2a), (2b) based on the number and type of requests of PE_i as follows:

$$R^{Conf}(i) + W^{Conf}(i) \leq R^{OtC}(i) + (1 - wb) \cdot W^{OtC}(i) \tag{20}$$

$$if NB_i = 1 : N^{None}(i) = R^c(i) - R^{OtC}(i) + (1 - wb) \cdot (W^c(i) - W^{OtC}(i)) \tag{21}$$

$$N^{ACT}(i) = N^{ACT,a}(i) + N^{ACT,b}(i) \tag{22}$$

$$N^{ACT,b}(i) \leq R^{OtC}(i) + (1 - wb) \cdot W^{OtC}(i) \tag{23}$$

$$N^{ACT,a}(i) + N^{ACT,b}(i) \leq R^c(i) + (1 - wb) \cdot W^c(i) \tag{24}$$

$$if NB_i = 1 : N^{ACT}(i) = 0 \tag{25}$$

Note that again we multiply write requests of PE_i by $(1 - wb)$ since if $wb = 1$ such requests are not critical, and thus do not contribute to self-interference. If $NB_i = 1$, all requests of PE_i target the same bank; hence, Case (2a) and (2b) cannot hold (Equation 25), and instead all critical requests that were close in the original schedule ($R^c(i) - R^{OtC}(i)$ for reads and $W^c(i) - W^{OtC}(i)$ for writes) must be included in Case (1a) (Equation 21).

- Case (3): finally, we cover the CAS delay case. For each of the $R^{CAS}(i), W^{CAS}(i)$ R and W requests of PE_i that add extra CAS delay, we add a CAS term to L^{CAS} and substract $tCCD$ from the WCD bound Δ. Next, consider again the example in Figure 2. Note that to cause extra delay, the interfering request must have the opposite direction compared to the first request of PE_i: otherwise, the delay would be equal to the minimum CAS separation of $tCCD$, and no extra delay would be added. Hence, we can bound

$R^{CAS}(i), W^{CAS}(i)$ based on the total number of W and R interfering requests that can cause CAS delay, respectively:

$$R^{CAS}(i) \leq W^{Conf,c} + W^{Reorder,o} + W^{InterB}_{CAS} \tag{26}$$
$$W^{CAS}(i) \leq R^{Conf,c} + R^{Reorder,o} + R^{InterB}_{CAS} \tag{27}$$

The cumulative number of self-interfering requests (either R only, W only, or either R or W) can then be bounded based on the number of critical requests of PE_i:

$$\begin{aligned} &R^{Conf}(i) + W^{Conf}(i) + N^{ACT}(i) + R^{CAS}(i) + W^{CAS}(i) + N^{None}(i) \\ &\leq R^c(i) + R^o(i) + (1 - wb) \cdot (W^c(i) + W^o(i)) - 1 \end{aligned} \tag{28}$$
$$R^{Conf}(i) + R^{CAS}(i) \leq R^c(i) + R^o(i) \tag{29}$$
$$W^{Conf}(i) + W^{CAS}(i) \leq (1 - wb) \cdot W^c(i) + (1 - wb) \cdot W^o(i) \tag{30}$$

Note that we subtract 1 in Equation 28 because the last request of PE_i cannot cause self-interference to another request of PE_i.

Finally, we shall use L^{self} to denote the sum of the self-delay in the original schedule that must be subtracted from the WCD Δ. We obtain:

$$L^{self} = (R^{Conf}(i) + W^{Conf}(i) + N^{ACT,b}(i) + R^{CAS}(i) + W^{CAS}(i)) \cdot tCCD + N^{ACT,a}(i) \cdot tRRD. \tag{31}$$

4.3 Conflict delay L^{Conf}

Based on Sections 4.1, 4.2, the total number of requests causing conflict delay is bounded by x^{Conf} intra-bank requests; plus $R^{Conf}(i) + W^{Conf}(i)$ self-interference requests; plus N^{WB} write requests if $wb = 1$. Based on Figure 1a, the conflict delay for a pair of successive requests can either be the larger $tRCD+tWL+tB+tWR+tRP$ or the smaller $tRAS+tRP$. Hence, we use variable $x^{Conf,W}$ to denote the number of conflicts of the first type, which require the first request in the pair to be a (open or close) write. We can then bound $x^{Conf,W}$ based on both the number of conflict delays $x^{Conf} + R^{Conf}(i) + W^{Conf}(i) + wb \cdot N^{WB}$; and the number of write requests that can trigger a conflict delay, which includes all write intra-bank requests $W^{Conf,c} + W^{Reorder,o}$, the write self-interference requests $W^{Conf}(i)$, and the write batching requests N^{WB}. This yields the following expressions:

$$\begin{aligned} L^{Conf} \leq\ & x^{Conf,W} \cdot (tRCD + tWL + tB + tWR + tRP) \\ &+ (x^{Conf} + R^{Conf}(i) + W^{Conf}(i) + wb \cdot N^{WB} - x^{Conf,W}) \cdot (tRAS + tRP) \end{aligned} \tag{32}$$

$$x^{Conf,W} \leq x^{Conf} + R^{Conf}(i) + W^{Conf}(i) + wb \cdot N^{WB} \tag{33}$$
$$x^{Conf,W} \leq W^{Conf,c} + W^{Reorder,o} + W^{Conf}(i) + wb \cdot N^{WB} \tag{34}$$

4.4 L^{ACT} and L^{CAS} delays

To compute the maximum PRE and ACT delays, we make use of the following observation:

▶ **Observation 4.** *Since for modern memory devices (e.g. DDR3/4), the value of inter-bank constraints tRRD and tCCD is at least 4, no more than one* ACT *and one* CAS *command can be issued every 4 cycles. Since furthermore the command priority is* CAS > ACT > PRE*, it follows that every* PRE *command can suffer at most 2 cycles of command bus conflict, and every* ACT *command at most 1 cycle.* CAS *commands do not suffer bus conflicts.*

Based on Observation 4, a PRE command can delay another PRE command by at most 3 cycles: one for the PRE command itself, and two more due to bus conflicts. For the case of ACT delay, we need to consider the $tRRD$ and $tFAW$ inter-bank ACT constraints. Note that $tRRD > 3$; hence, ACT delay is always greater than PRE delay as previously noticed. Since the $tFAW$ constraints is applied every 4 consecutive ACT, a valid upper bound to L^{ACT} can be constructed by multiplying the number of ACT delay terms, with is equal to $N^{InterB,c}_{ACT,c} + N^{ACT}(i)$, by the maximum of $tRRD$ and $tFAW/4$, then adding 1 to account for bus conflicts:

$$L^{ACT} \leq (N^{InterB,c}_{ACT,c} + N^{ACT}(i)) \cdot \big(\max(tRRD, tFAW/4) + 1\big) \tag{35}$$

Next, we discuss the CAS delay L^{CAS}. Based on Sections 4.1, 4.2, the total number of requests that cause CAS delay is $x^{CAS} + R^{CAS}(i) + W^{CAS}(i) + R^{InterB}_{CAS} + W^{InterB}_{CAS}$. The inter-bank CAS constraint between a pair of requests depends on the direction of the requests themselves: for a W followed by a R, $tWL + tB + tWTR$; for a R followed by a W, $tRTW$; and for two requests of the same direction, $tCCD$. Therefore, let variables $x^{CAS,WR}$ and $x^{CAS,RW}$ to indicate the number of W-to-R and R-to-W pairs. We then have:

$$\begin{aligned} L^{CAS} &\leq x^{CAS,WR} \cdot (tWL + tB + tWTR) + x^{CAS,RW} \cdot tRTW \\ &+ (x^{CAS} + R^{CAS}(i) + W^{CAS}(i) + R^{InterB}_{CAS} + W^{InterB}_{CAS} - x^{CAS,WR} - x^{CAS,RW}) \cdot tCCD \end{aligned} \tag{36}$$

To bound $x^{CAS,WR}$ and $x^{CAS,RW}$, we determine the maximum number of R and W requests for the first and second request in each pair. We note that interfering requests can be either; interfered critical requests of PE_i can only be the latter; and self-interfering requests of PE_i can only be the former. This yields:

$$R^{CAS,first} = R^{CAS}(i) + R^{Conf,c} + R^{Reorder,o} + R^{InterB}_{CAS} \tag{37}$$
$$R^{CAS,second} = R^{c}(i) + R^{o}(i) + R^{Conf,c} + R^{Reorder,o} + R^{InterB}_{CAS} \tag{38}$$
$$W^{CAS,first} = W^{CAS}(i) + W^{Conf,c} + W^{Reorder,o} + W^{InterB}_{CAS} \tag{39}$$
$$W^{CAS,second} = (1 - wb) \cdot (W^{c}(i) + W^{o}(i)) + W^{Conf,c} + W^{Reorder,o} + W^{InterB}_{CAS} \tag{40}$$
$$x^{CAS,WR} \leq W^{CAS,first} \wedge x^{CAS,WR} \leq R^{CAS,second} \tag{41}$$
$$x^{CAS,RW} \leq R^{CAS,first} \wedge x^{CAS,RW} \leq W^{CAS,second} \tag{42}$$
$$x^{CAS,WR} + x^{CAS,RW} \leq x^{CAS} + R^{CAS}(i) + W^{CAS}(i) + R^{InterB}_{CAS} + W^{InterB}_{CAS} \tag{43}$$

Total Cumulative Delay Bound. Finally, Δ is simply computed based on the sum of all terms computed so far:

$$\Delta = L^{Conf} + L^{ACT} + L^{CAS} - L^{self} \tag{44}$$

5 Interference Computation

Based on Equation 44, in Section 4 we have determined a bound Δ on the cumulative WCD suffered by requests of PE_i, assuming that the total number of requests per interfering component is known. We now seek to determine how many requests of each interfering PE contribute to each component. To this end, as shown in Table 3, for each interfering component we define a new set of variables with index (p) to represent the number of requests for that component that belong to PE_p. The total number of requests for each intra-

and inter-component is then equal to the sum over all cores: $\mathcal{V} = \sum_{\forall p \neq i} \mathcal{V}(p)$, where $\mathcal{V}$ is either $R^{Conf,c}$, $W^{Conf,c}$, $R^{Reorder,o}$, $W^{Reorder,o}$, $R_c^{InterB,c}$, $R_c^{InterB,o}$, $W_c^{InterB,c}$, $W_c^{InterB,o}$, R_o^{InterB}, or W_o^{InterB}.

We now proceed as follows. In Section 5.1, we first bound the number of per-PE interfering requests based on the total number of requests generated by each PE_p (job-driven bound). In Sections 5.2 – 5.5, we then bound the per-PE interfering requests based on the platform instance (request-driven bound). Note that for the write batching component, in Section 5.5 we will need to distinguish among three sets of requests for each PE, based on when the requests are generated: $W^{btch}(p), W^{before}(p)$ and $W^{after}(p)$. Furthermore, the write batching component includes requests of PE_i itself. Hence, the write batching component is bounded as follows:

$$W^{WB} = \sum_{\forall p} \left(W^{btch}(p) + W^{before}(p) + W^{after}(p) \right) \tag{45}$$

Finally, we show how to compute the WCD bound by solving a LP problem in Section 5.6.

5.1 Job-Driven Bounds

Recall from Section 3 that $R^o(p), R^c(p), W^o(p), W^c(p)$ denote the total number of R/W open/close requests for PE_p. We thus have:

$$R^{Conf,c}(p) + R_c^{InterB,c}(p) \leq R^c(p) \tag{46}$$

$$W^{Conf,c}(p) + W_c^{InterB,c}(p) \leq W^c(p) \tag{47}$$

$$R_c^{InterB,o}(p) + R^{Reorder,o}(p) \leq R^o(p) \tag{48}$$

$$W_c^{InterB,o}(p) + W^{Reorder,o}(p) \leq W^o(p) \tag{49}$$

$$R^{Conf,c}(p) + R_c^{InterB,c}(p) + R_c^{InterB,o}(p) + R^{Reorder,o}(p) + R_o^{InterB}(p) \leq R^c(p) + R^o(p) \tag{50}$$

$$W^{Conf,c}(p) + W_c^{InterB,c}(p) + W_c^{InterB,o}(p) + W^{Reorder,o}(p) + W_o^{InterB}(p) \leq W^c(p) + W^o(p) \tag{51}$$

$$W^{btch}(p) + W^{before}(p) + W^{after}(p) \leq W^c(p) \tag{52}$$

Equations 46-49 bound the number of requests of each type (open/close) and direction (R/W). Equations 50 bounds the number of read requests over all components; note that read inter-bank-open requests R_o^{InterB} can be either open or close. Similarly, Equation 51 bounds the write requests used in delay components when $wb = 0$; while Equation 52 bounds the write requests used in the write-batching delay for $wb = 1$.

5.2 Request-Driven Bounds: Conflict Requests

We introduce a constant n_p^{Conf} to denote the maximum number of conflict requests of PE_p that can interfere with one critical request of PE_i. Since conflict requests target the same bank as the critical request, n_p^{Conf} is zero if PE_p does not share any bank with PE_i. Otherwise, since conflict requests arrive before a critical request, we can set n_p^{Conf} equal to the maximum number of outstanding requests of PE_p, which is 1 if PE_p is in-order, and PR if out-of-order. Hence:

$$n_p^{Conf} = \begin{cases} \text{if } cr: \begin{cases} 0 & \text{if } PartAll \text{ or } PartCr \\ 1 & \text{if } NoPart \text{ and } (IOCr \text{ or } IO) \\ PR & \text{if } NoPart \text{ and } OOO \end{cases} \\ \text{if } ncr: \begin{cases} 0 & \text{if } PartAll \\ 1 & \text{if pr or } ((NoPart \text{ or } PartCr) \text{ and } IO) \\ PR & \text{if } (NoPart \text{ or } PartCr) \text{ and } (OOO \text{ or } IOCr) \end{cases} \end{cases} \tag{53}$$

Since conflict interfered requests must be close, the number of requests from core under analysis is bounded by $R^c(i) + (1 - wb) \cdot W^c(i)$, which yields the following constraint:

$$\forall p \neq i : R^{Conf,c}(p) + W^{Conf,c}(p) \leq n_p^{conf} \cdot (R^c(i) + (1 - wb) \cdot W^c(i)) \tag{54}$$

Finally, if $pr = 1$, at most one outstanding request of non-critical PEs can interfere with a critical request; hence we also obtain:

$$\text{if } pr\text{: } \sum_{\forall p \neq i, p \text{ ncr}} \Big(R^{Conf,c}(p) + W^{Conf,c}(p) \Big) \leq (R^c(i) + (1 - wb) \cdot W^c(i)) \tag{55}$$

5.3 Request-Driven Bounds: Reorder Requests

Reorder requests target the same bank as requests of PE_i. Hence, if PE_p and PE_i do not share any bank, the number of reorder requests of PE_p is zero. Similarly, if $pr = 1$, requests of non-critical PEs cannot be reordered ahead of critical requests, since PE_p has higher priority. Accordingly:

$$\text{if } PartAll \text{ or } PartCr, \forall p \neq i, p \text{ cr} : R^{Reorder,o}(p) = W^{Reorder,o}(p) = 0 \tag{56}$$

$$\text{if } PartAll \text{ or } pr, \forall p \neq i, p \text{ ncr} : R^{Reorder,o}(p) = W^{Reorder,o}(p) = 0 \tag{57}$$

Furthermore, if $thr = 1$, by definition no more than N_{thr} requests can be reordered ahead of each close request of PE_i. Hence it also holds:

$$\text{if } thr : R^{Reorder,o} + W^{Reorder,o} \leq N_{thr} \cdot \big(R^c(i) + (1 - wb) \cdot W^c(i) \big) \tag{58}$$

5.4 Request-driven bounds: inter-bank requests

Let $N_{reqs,c}$, $N_{reqs,o}$ be the number of requests that can be delayed by inter-bank-close and inter-bank-open requests, as introduced in Section 4.1:

$$N_{reqs,c} = R^c(i) + (1 - wb) \cdot W^c(i) + R^{Conf,c} + W^{Conf,c} \tag{59}$$

$$N_{reqs,o} = R^o(i) + (1 - wb) \cdot W^o(i) + R^{Reorder,o} + W^{Reorder,o} \tag{60}$$

We first bound the number of requests generated by PE_p that can interfere on each of the $N_{reqs,c}$ close requests. Due to the assumption of RR arbitration among banks, the number of interfering inter-bank requests is limited by the number of banks in case where inter-bank reordering is not allowed ($breorder = 0$) or write batching is deployed ($wb = 1$), since it cancels the effects of inter-bank reordering [14]. Since PE_p can only access NB_p banks by definition, it must hold:

$$\text{if } (wb = 1 \text{ or } breorder = 0): \quad \forall p \neq i : R_c^{InterB,o}(p) + R_c^{InterB,c}(p) + W_c^{InterB,o}(p) + W_c^{InterB,c}(p) \leq NB_p \cdot N_{reqs,c} \tag{61}$$

Similarly, we can bound the interference caused by all critical (other than PE_i) and non-critical PEs based on the total number of banks they can access. Note that by definition, inter-bank interfering requests target a different bank than the request they interfere upon. Hence, inter-bank requests of critical PEs can target $N_{cr} - 1$ banks, while inter-bank requests of any PEs can target $N_B - 1$ banks:

$$\text{if } (wb = 1 \text{ or } breorder = 0): \quad \sum_{\forall p \neq i, p \text{ cr}} \Big(R_c^{InterB,o}(p) + R_c^{InterB,c}(p) + W_c^{InterB,o}(p) + W_c^{InterB,c}(p) \Big) \leq (N_{cr} - 1) \cdot N_{reqs,c} \tag{62}$$

$$\text{if } (wb{=}1 \text{ or } breorder{=}0): \left(R_c^{InterB,o} + R_c^{InterB,c} + W_c^{InterB,o} + W_c^{InterB,c} \right) \leq (N_B - 1) \cdot N_{reqs,c} \tag{63}$$

Finally, similarly to the constraint for conflict requests in Equation 55, if the MC uses a priority scheme, then the number of interfering requests from all non-critical PEs is in worst-case one for each of the N_{reqs} interfered requests:

$$\text{if } pr \text{ and } (wb = 1 \text{ or } breorder = 0): \sum_{\forall p \neq i, p \text{ ncr}} \left(R_c^{InterB,o}(p) + R_c^{InterB,c}(p) + W_c^{InterB,o}(p) + W_c^{InterB,c}(p) \right) \leq N_{reqs,c} \tag{64}$$

We now consider inter-bank-open requests. All derived constraints depend on the RR arbitration and bank assignment; hence, Equations 61-64 also apply to the number of interfering inter-bank-open requests $R_o^{InterB}(p) + W_o^{InterB}(p)$, except that we consider $N_{reqs,o}$ in place of $N_{reqs,c}$.

5.5 Request-driven bounds: write-batching requests

If $wb = 1$, the $R^o(i) + R^c(i)$ critical read requests of PE_i can suffer interference from write batches created by writes of either PE_i or other PEs. Since the MC gives priority to read requests over write batches, in the worst case a critical R request can be delayed by a single batch of W_{btch} write requests started before the R arrives. Hence, if we let $W^{btch}(p)$ to denote the number of interfering write requests of PE_p that arrive while no critical request is active (arrived but not completed), we have:

$$\sum_{\forall p} W^{btch}(p) \leq W_{btch} \cdot (R^o(i) + R^c(i)) \tag{65}$$

However, after a critical request arrives but before it completes, further writes that arrive in the system may fill the write buffer, forcing additional batches to be processed. Therefore, we next consider the number of interfering write requests that arrive while a critical request is active. Recall that the system has a write-back write-allocate last-level cache. Accordingly, a write request can only be generated in conjunction with a read request; hence, we will reason about the maximum number of read requests that can be generated while a critical request is active. In particular, we use W^{after} to denote the number of write requests corresponding to reads that arrive while a critical request is active, and are executed after the critical request (but their corresponding writes can be executed before that critical request due to the batching scheme); and W^{before} to denote the number of write requests corresponding to read requests that arrive while a critical request is active, and are executed before it. We start by bounding W^{after}. Similar to the conflict interference in Section 5.2, we introduce a constant n_p^{after} to denote the maximum number of requests that can arrive while the critical request is active and are executed after it. Hence, it can be computed by Equation 66, and $W^{after}(p)$ can be accordingly computed by Equation 67.

$$n_p^{after} = \begin{cases} 1 & \text{if } IO \text{ or } (IOCr \text{ and } p \text{ } cr) \\ PR & \text{if } OOO \text{ or } (IOCr \text{ and } p \text{ } ncr) \end{cases} \tag{66}$$

$$\forall p \neq i : W^{after}(p) \leq n_p^{after} \cdot (R^o(i) + R^c(i)) \tag{67}$$

We now consider the W^{before} requests. First, if $pr = 1$, each critical request can suffer interference from a maximum of one W^{before} request from all the non-critical PEs, therefore:

$$\text{if pr: } \sum_{\forall p, p \text{ ncr}} W^{before}(p) \leq R^o(i) + R^c(i) \tag{68}$$

Second, if interfering PE_p does not share banks with PE_i, then $W^{before}(p)$ can be only due to the inter-bank RR arbitration among banks. Recall that PE_p is assigned NB_p banks, the total number of banks assigned to critical cores other than the bank targeted by the request under analysis is $N_{cr} - 1$, and the total number of banks assigned to all cores other than the bank targeted by the request under analysis is $NB - 1$. As a result, the following three conditions hold from the RR arbitration:

$$\forall p \neq i : \text{if } (PartAll) \text{ or } (PartCr \text{ and } p \text{ is cr}): W^{before}(p) \leq NB_p \cdot (R^o(i) + R^c(i)) \tag{69}$$

$$\text{if } PartAll \text{ or } PartCr: \sum_{\forall p \neq i \wedge p \text{ cr}} W^{before}(p) \leq (N_{cr} - 1) \cdot (R^o(i) + R^c(i)) \tag{70}$$

$$\text{if } PartAll: \sum_{\forall p \neq i} W^{before}(p) \leq (NB - 1) \cdot (R^o(i) + R^c(i)) \tag{71}$$

Finally, if no partitioning is deployed, we also have the FR-FCFS reordering. Thus, each request from the core under analysis can be interfered by N_{thr} (if threshold is deployed) due to intra-bank FR-FCFS reordering, while each of these requests can also be delayed by $NB - 1$ requests from RR inter-bank arbitration. Additional $NB - 1$ requests can interfere with the request under analysis itself. This gives a total of $(N_{thr} + 1) \cdot (NB - 1)$:

$$\text{if } thr = 1: \sum_{\forall p \neq i} W^{before}(p) \leq (N_{thr} + 1) \cdot (NB - 1) \cdot (R^o(i) + R^c(i)) \tag{72}$$

5.6 Optimization Problem

Consider the variables in Table 3; by definition, numbers of requests and constraints are positive integers, and the same holds for delay terms since we measure them in clock cycles. Furthermore, all constraints introduced in Sections 3-5 are linear in such variables. Hence, we could compute an upper bound on Δ by solving an integer LP problem, with the optimization objective of maximizing Equation 44. In practice, we consider a linear relaxation of the same problem, where all variables are treated as reals; by construction, the resulting LP problem still yields a valid bound on Δ. The number of variables and constraints is proportional to P; hence the complexity of solving the linear programming problem is polynomial in the number of PEs.

6 Evaluation

Simulation Environment. We use MacSim [19], a heterogeneous multi-processor simulator integrated with DRAMSim2 [32]. MacSim models x86 architecture and supports IO and OOO PEs. It also allows the configuration of the maximum number of pending requests through managing the number of entries in MSHR registers. MacSim has a frontend that includes the virtual-to-physical mapping. This enables us to implement partitioning without running a complete OS. We implement the three partitioning schemes discussed in Section 3. DRAMSim2 [28] is a cycle-accurate DRAM simulator, which we extend to also support priority assignment amongst PEs as well as write batching. We implement the optimization framework in Matlab and it finishes within few seconds for all experiments using a machine with a quad-core i7 processor and 8GB DRAM and is running Linux.

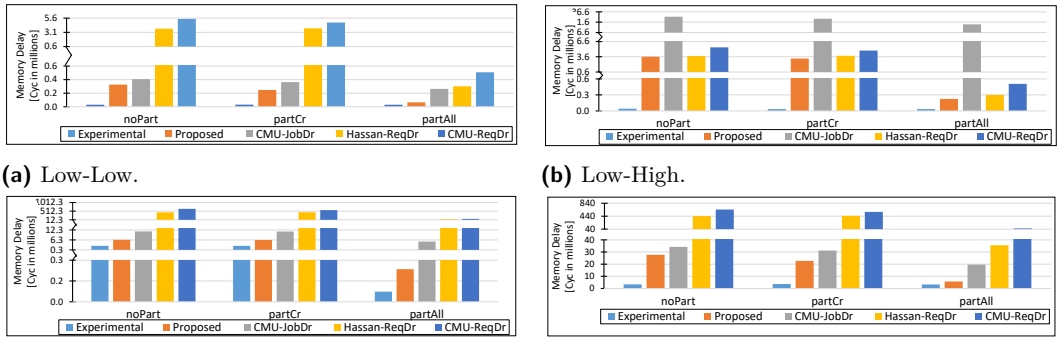
Table 4 Evaluation Setup.

(a) Benchmarks.

High				Low			
BM	#Reads	#Writes	total	BM	#Reads	#Writes	total
matrix	280000	38428	318428	rspeed	2000	482	2479
a2time	166000	21751	187751	pntrch	2000	479	2478
aifftr	101000	77234	178234	basefp	2000	478	2478

(b) Configuration parameters.

P	4	Pcr	2	$Pncr$	2
N_{thr}	8	W_{bch}	16	PR	4
N_B	8				
N_{Bp}	8 ($noPart$), 2 ($PartAll$), or 4/8 ($PartCr$ and p is cr/ncr)				
N_{Bcr}	8 ($noPart$ or $PartCr$), 4 ($PartAll$)				
N_{Bncr}	8 ($noPart$ or $PartCr$), 4 ($PartAll$)				



(a) Low-Low. **(b)** Low-High.

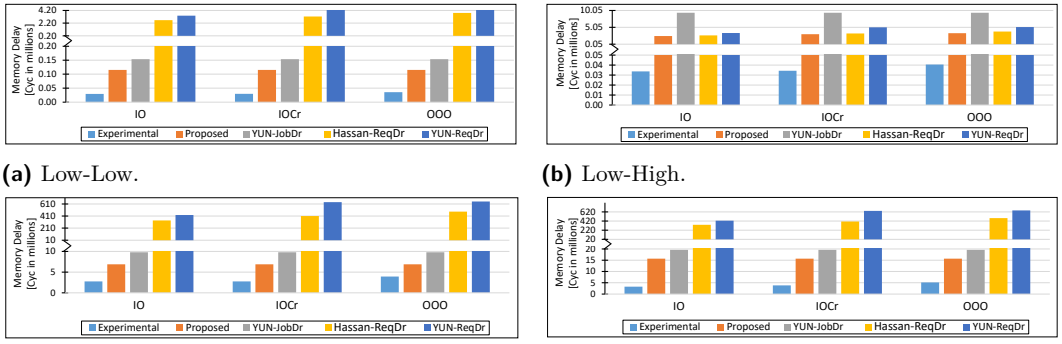
(c) High-Low. **(d)** High-High.

Figure 3 Results for configurations that are considered by CMU [20].

Benchmarks. We use benchmarks from the EEMBC-auto suite [29], which include representative applications from the embedded automotive domain. Recall from Section 5 that the maximum number of interfering requests, and thus the memory delay incurred by the PE under analysis, depend both on the number of memory requests initiated by this PE as well as the number of requests issued by the competing PEs. Therefore, we construct experiments that capture different scenarios. Towards doing so, we classify the used benchmarks into two categories: *High* and *Low* as shown in Table 4a. The *High* (*Low*) benchmarks are those that issue a large (a small) number of memory requests.

Experiments Setup. We compare the proposed analysis with five state-of-the-art approaches; two of them are job-driven analyses: CMU-JobDr [20] and YUN-JobDr [37]; and three are request-driven analyses: CMU-ReqDr [20], YUN-ReqDr [37], and Hassan-ReqDr [14]. We also compare against the experimental WCD observed from the simulator, denoted as Experimental. We use a system composed of four cores: two in-order critical and the other are OOO non-critical ones. Table 4b lists all values of used parameters. Since both CMU and YUN do not support mixed criticalities, for their analysis all tasks are considered to have the same criticality. In addition, since they also cover only certain system configurations, we compare against these solutions under all configurations they support. To evaluate each analysis under different interference scenarios, we run different experiments using different mix of the benchmarks in Table 4a. Namely, we evaluate with four different scenarios: Low-Low, Low-High, High-Low, and High-High, where the first term refers to the task under analysis and the second term refers to interfering tasks. For instance, in a Low-High scenario, the interfered task is chosen to be the *rspeed* benchmark, which is in the Low category in Table 4a, while the interfering tasks are *matrix, a2time*, and *aifftr* from the High category.

1) System Configurations Supported By CMU. Both job- and request-driven CMU's analyses [20] can be applied to platform instances with in-order pipelines, all cores have same priority, and the memory controller deploys a FR-FCFS threshold but does not deploy write



(a) Low-Low. **(b)** Low-High.

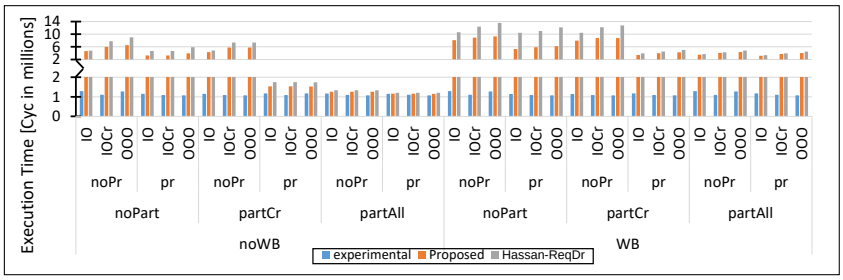
(c) High-Low. **(d)** High-High.

Figure 4 Results for configurations that are considered by YUN [37].

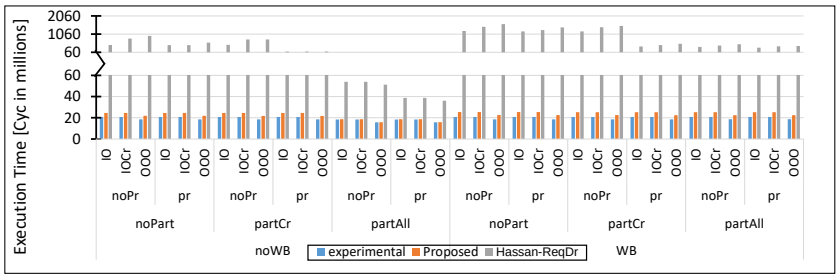
batching. However, the analysis can cover different bank partitioning scenarios. Accordingly, these are the instances we used in our experiments when comparing against those approaches. Figure 3 delineates the total memory delay suffered for different considered approaches under different interference scenarios. We make the following observations. 1) As aforestated, the request- and job-driven approaches are incomparable: neither approach is better than the other under all scenarios. Although CMU-JobDr provides tighter delay bounds than request-driven ones (both CMU-ReqDr and Hassan-ReqDr) in Figures 3a, 3c, and 3d, the request driven approaches have better bounds for the Low-High scenario in Figure 3b. Since request-driven analysis considers only the number of requests from the core under analysis, when this number is relatively small compared to the total number of competing requests, this analysis provides tighter bounds. This is the case for the Low-High scenario. For other scenarios, the number of requests of the core under analysis is relatively large and leads to the larger delay bounds of the request-driven analyses. 2) The proposed analysis provides the tightest bounds across all scenarios. For the Low-High scenario, Proposed provides up to a 34% tighter bound than the second best approach, which is Hassan-ReqDr (*PartAll* in Figure 3b). For all other scenarios, Proposed provides at least 24% (*noPart* in Figure 3a) and up to 22.6× (*PartAll* in Figure 3c) tighter bound than CMU-JobDr, which is the second best approach in all these scenarios.

2) System Configurations Supported By YUN. The platform instances covered in [37] (for both YUN-JobDr and YUN-ReqDr) are partitioning banks across cores (*PartAll*), all cores have same priority, and the memory controller deploys both FR-FCFS threshold and write batching. Although [37] only evaluates OOO cores, we find that the analysis is extensible to any core pipeline (by managing maximum number of pending requests from each core). Therefore, we experiment with different pipelining configurations and show the results in Figure 4. 1) YUN-JobDr provides tighter bounds than request driven analyses (YUN-ReqDr and Hassan-ReqDr) for all interference scenarios except for Low-High. 2) The Proposed approach still provides the tightest bounds across all scenarios. Proposed provides at least 25% (IO in Figure 4d) and up to 42% (OOO in Figure 4c) better bounds compared to YUN-JobDr (next best approach) in the Low-Low, High-Low, and High-High scenarios. In the Low-High scenario in Figure 4b, it provides up to 15% better bounds than the second best option of Hassan-ReqDr.

3) System Configurations Supported By Hassan-ReqDr. We now compare the proposed analysis with the Hassan-ReqDr analysis for all the supported platform instances discussed in Section 3. We compared both approaches for all interference scenarios; however, for



(a) Low-High.



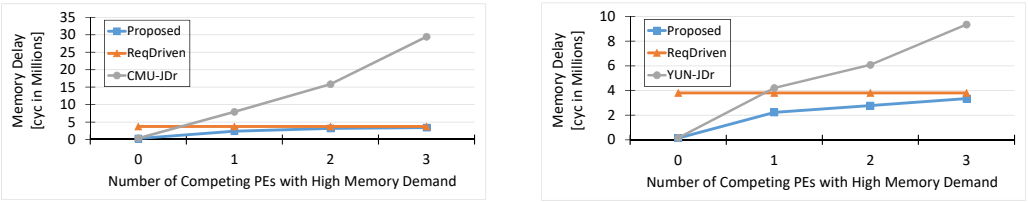
(b) High-Low.

Figure 5 Comparison of the total response time with Hassan-ReqDr [14] for all configurations.

Table 5 Proposed's WCD (in cycles) for the 63 platform instances that are declared unbounded under Hassan-ReqDr (satisfying condition in Equation 73). We found WCD to depend only on WB and paritioning values. Values are for the Low-High interference scenario.

Partitioning	noWB	WB	Partitioning	noWB	WB	Partitioning	noWB	WB
PartAll	15706330	24540906	*PartCr*	24560480	24540906	*noPart*	24560480	24540906

space considerations, in Figure 5, we only show results for the Low-High and High-Low scenarios, which best illustrate the main lessons we want to highlight. 1) Proposed provides tighter bound than Hassan-ReqDr for all platform instances. 2) For the Low-High scenario (Figure 5a), bounds of both solutions are very close. This is because for his scenario, the number of requests from the core under analysis is relatively small compared to the total number of competing requests. Therefore, request-driven analysis (Sections 5.2 – 5.5) provides tighter bounds than job-driven analysis (Section 5.1). However, Proposed still outperforms Hassan-ReqDr thanks to the leveraged knowledge about the running tasks. As a result, in Figure 5a, it provides up to 98% (instance WB-*noPart*-pr-IO) and 24% on average tighter bounds across all platform instances. 3) For the High-Low scenario (Figure 5b), we observe a large gap between Proposed and Hassan-ReqDr. In Figure 5b, Proposed provides up to 71× and 18× on average tighter bound across all configurations. Two main reasons are behind such significant gap: no partitioning (*noPart*), and write batching (WB). Both features, if considered, forces Hassan-ReqDr to consider a pathological worst-case scenario that is overly pessimistic. For *noPart*, Hassan-ReqDr considers every request of the core under analysis to have the worst-case intra-bank (conflict and reorder) interference from competing cores. Similarly for WB, Hassan-ReqDr assumes that every read request will suffer a worst-case write batching delay even if there are not enough number of competing requests to cause this much interference for every single request from the core under analysis. On the other hand, by leveraging the job-driven analysis and considering the number of competing requests, Proposed provides tighter bounds.



(a) A comparison for a configuration with no WB, no partitioning, IO pipeline, and no priority.

(b) A comparison for a configuration with WB, *PartAll*, OOO pipeline, and no priority.

Figure 6 Varying number of "High" competing cores.

4) Configurations Unbounded by Hassan-ReqDr. In [14], the authors considered 144 platform instances. The Hassan-ReqDr analysis bounded 81 of them, while 63 instances were proven to be unbounded under this analysis. We identify those 63 instances by the following condition:

$$\Big(breorder{=}1 \text{ and } wb{=}0\Big) \text{ or } \Big(thr{=}0 \text{ and } \big(part{=}noPart \text{ or } (part{=}PartCr \text{ and } pr{=}0)\big)\Big) \quad (73)$$

Leveraging the job-driven analysis, the Proposed approach is able to bound all these cases using information about memory requests of competing tasks (Section 5.1). Table 5 shows the obtained bounds for the Low-High scenario.

5) Hybrid Analysis under different Interference Severity. To further show the benefit of the proposed hybrid analysis compared to the state-of-the-art request- and job-driven analyses, we investigate with different number of competing tasks with high memory demand (High from Table 4a). In this set of experiments, we use *rspeed* (Low) benchmark as the one under analysis and vary the number of high competing cores. The total number of cores in the experiment is four. For instance, 2 in the x-axis of Figure 6 indicates that in addition to the core under analysis, two cores are running benchmarks from the High category, while one core is running a benchmark from the Low category. From Figure 6, it is clear that the effectiveness of the request- vs job-driven analysis is dependent on the relative ratio between the number of requests of the core under analysis and the number of requests from competing cores. On the other hand, the proposed hybrid analysis is able to achieve better bound compared to both approaches for all cases.

7 Conclusions

We propose a novel approach to bound interference delays due to contention upon accessing off-chip DRAMs in heterogeneous COTS MPSoCs. The proposed hybrid framework blends both request- and job-driven analyses to provide tighter bounds than those determined by each analysis separately and then taking the minimum of both. The framework also leverages information about the memory behavior of running task such as number of read and write requests, which are usually available from statically analyzing each task in isolation. We evaluate the proposed approach across a wide set of COTS platform instances, where it outperforms existing state-of-the-art analyses (both request- and job-driven).

References

1 Ankit Agrawal, Gerhard Fohler, Johannes Freitag, Jan Nowotsch, Sascha Uhrig, and Michael Paulitsch. Contention-aware dynamic memory bandwidth isolation with predictability in COTS multicores: An avionics case study. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2017.

2 Ankit Agrawal, Renato Mancuso, Rodolfo Pellizzoni, and Gerhard Fohler. Analysis of dynamic memory bandwidth regulation in multi-core real-time systems. In *IEEE Real-Time Systems Symposium (RTSS)*, 2018.

3 Balasubramanya Bhat and Frank Mueller. Making DRAM refresh predictable. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2010.

4 Vamsi Boppana, Sagheer Ahmad, Ilya Ganusov, Vinod Kathail, Vidya Rajagopalan, and Ralph Wittig. UltraScale+ MPSoC and FPGA families. In *IEEE Hot Chips Symposium (HCS)*, 2015.

5 Roman Bourgade, Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Accurate analysis of memory latencies for WCET estimation. In *International Conference on Real-Time and Network Systems (RTNS)*, 2008.

6 Mauricio Calle and Ravi Ramaswami. Multi-bank scheduling to improve performance on tree accesses in a DRAM based random access memory subsystem, January 2005. US Patent 6,839,797.

7 Leonardo Ecco, Sebastian Tobuschat, Selma Saidi, and Rolf Ernst. A Mixed Critical Memory Controller Using Bank Privatization and Fixed Priority Scheduling. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2014.

8 Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, and Lothar Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *ACM International Conference on Embedded Software (EMSOFT)*, 2013.

9 Danlu Guo, Mohamed Hassan, Rodolfo Pellizzoni, and Hiren Patel. A Comparative Study of Predictable DRAM Controllers. *ACM Transaction on Embedded Computer Systems (TECS)*, 2018.

10 Danlu Guo and Rodolfo Pellizzoni. A request bundling dram controller for mixed-criticality systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

11 Sebastian Hahn, Michael Jacobs, and Jan Reineke. Enabling compositionality for multicore timing analysis. In *International conference on real-time networks and systems (RTNS)*, 2016.

12 Mohamed Hassan. Heterogeneous MPSoCs for Mixed Criticality Systems: Challenges and Opportunities. *IEEE Design & Test*, 2017.

13 Mohamed Hassan, Hiren Patel, and Rodolfo Pellizzoni. A Framework for Scheduling DRAM Memory Accesses for Multi-Core Mixed-time Critical Systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

14 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 (TCAD)*, 2018.

15 Intel. External memory interface handbook volume 2: Design guidelines, 2017.

16 Bruce Jacob, Spencer Ng, and David Wang. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

17 Javier Jalle, Eduardo Quinones, Jaume Abella, Luca Fossati, Marco Zulianello, and Francisco J Cazorla. A dual-criticality memory controller (DCmc): Proposal and evaluation of a space case study. In *IEEE Real-Time Systems Symposium (RTSS)*, 2014.

18 DDR3 SDRAM JEDEC. JEDEC jesd79-3b, 2008.

19 H Kim, J Lee, N Lakshminarayana, J Lim, and T Pho. Macsim: Simulator for heterogeneous architecture, 2012.

20 Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Raj Rajkumar. Bounding memory interference delay in COTS-based multi-core

systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

21 N. Kim, B. Ward, M. Chisholm, J. Anderson, and F.D. Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. *Real-Time Systems*, 2017.

22 Haohan Li and Sanjoy Baruah. Global mixed-criticality scheduling on multiprocessors. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

23 Yonghui Li, Benny Akesson, and Kees Goossens. Dynamic Command Scheduling for Real-Time Memory Controllers. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

24 Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 146–160. IEEE, 2007.

25 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.

26 Xing Pan, Yasaswini Gownivaripalli, and Frank Mueller. Tintmalloc: Reducing memory access divergence via controller-aware coloring. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

27 Risat Mahmud Pathan. Schedulability analysis of mixed-criticality systems on multiprocessors. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

28 Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for memory interference in multicore systems. In *IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2010.

29 Jason Poovey. Characterization of the EEMBC benchmark suite. *North Carolina State University*, 2007.

30 Qualcomm. Qualcomm snapdragon 600e processor apq8064e recommended memory controller and device settings application note, 2016.

31 Scott Rixner, William J Dally, Ujval J Kapasi, Peter Mattson, and John D Owens. Memory access scheduling. *ACM SIGARCH Computer Architecture News*, 28(2):128–138, 2000.

32 Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. DRAMSim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters (CAL)*, 2011.

33 Jeffrey Stuecheli, Dimitris Kaseridis, Hillery C Hunter, and Lizy K John. Elastic refresh: Techniques to mitigate refresh penalties in high density memory. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010.

34 Prathap Kumar Valsan and Heechul Yun. MEDUSA: a predictable and high-performance DRAM controller for multicore based embedded systems. In *IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, 2015.

35 Zheng Pei Wu, Rodolfo Pellizzoni, and Danlu Guo. A Composable Worst Case Latency Analysis for Multi-Rank DRAM Devices under Open Row Policy. *Real-Time Systems*, 2016.

36 Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

37 Heechul Yun, Rodolfo Pellizzon, and Prathap Kumar Valsan. Parallelism-aware memory interference delay analysis for COTS multicore systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2015.

38 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 *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

39 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory bandwidth management for efficient performance isolation in multicore platforms. *IEEE Transactions on Computers (TC)*, 2016.

smARTflight: An Environmentally-Aware Adaptive Real-Time Flight Management System

Anam Farrukh
Department of Computer Science, Boston University, MA, USA
afarrukh@bu.edu

Richard West
Department of Computer Science, Boston University, MA, USA
richwest@bu.edu

Abstract

Multi-rotor drones require real-time sensor data processing and control to maintain flight stability, which is made more challenging by external disturbances such as wind. In this paper we introduce smARTflight: an environmentally-aware *adaptive real-time* flight management system. smARTflight adapts the execution frequencies of flight control tasks according to timing and safety-critical constraints, in response to transient fluctuations of a drone's attitude. In contrast to current state-of-the-art methods, smARTflight's criticality-aware scheduler reduces the latency to return to a steady-state target attitude. The system also improves the overall control accuracy and lowers the frequency of adjustments to motor speeds to conserve power. A comparative case-study with a well-known autopilot shows that smARTflight reduces unnecessary control loop executions under stable conditions, while reducing response time latency by as much as 60% in a given axis of rotation when subjected to a 15° step attitude disturbance.

2012 ACM Subject Classification Computer systems organization → Firmware

Keywords and phrases adaptive real-time systems, safety criticality, flight controller, multi-rotor drones, environmental awareness

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.24

Funding This work is supported in part by the National Science Foundation (NSF) under Grant #1527050. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

1 Introduction

Multi-rotor drones and copters are increasingly being used in cyber-physical applications that require unmanned aerial vehicles (UAVs). Their growing popularity is largely attributed to their flexibility, low costs, ability to hover, and ease of maneuverability. Robust hardware designs complemented with agile software stacks make them prime candidates for use in ground-breaking applications such as aerial photography [9, 23], remote package delivery [15], territorial exploration and inspection [25], infrastructure mapping [17,19], search and rescue [4, 18], and many others. The vast array of promising use-cases requires UAVs to navigate different environments, which pose significant challenges to flight stability. In turn, more sophisticated flight management systems are needed to dynamically compensate for adverse environmental conditions, such as wind disturbances.

At the heart of all multi-rotor aerial vehicles lies the *autopilot*. It is a flight controller firmware or software that combines sensor data processing with attitude [1] estimation and rotor speed adjustments to maintain a target trajectory. Flight management on typical multi-

[1] Attitude refers to the 3D orientation of an object relative to the Earth's stationary reference frame.

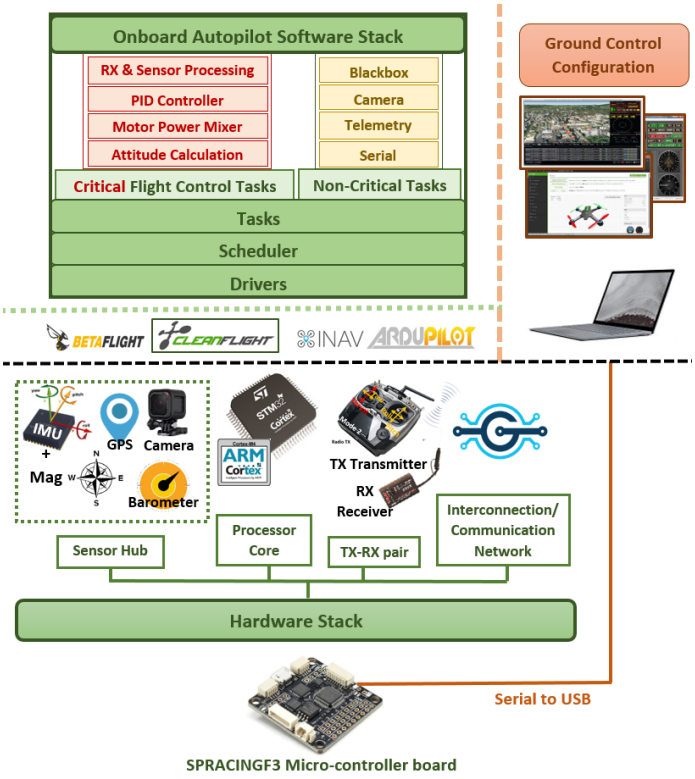


Figure 1 Hardware and software components of an autopilot firmware. Critical and non-critical tasks are dispatched for execution by the underlying scheduler. Configuration applications typically run on ground computers for initial setup of various parameters such as mission parameters, loop times, PID gains, and filter cutoff frequencies.

rotor drones employs classical linear mechanisms that are energy inefficient, non-deterministic in time, and prone to instability. Current autopilots lack the ability to dynamically adapt flight behavior in response to external disturbances. Consequently varying environmental conditions such as wind significantly impact the ability of a drone to maintain flight along a target path. Manual intervention is oftentimes required to correct for situations that would otherwise lead to crashes or inability to achieve flight objectives.

The key issues directly impacting drone performance include: 1) lack of adaptable and timing predictable flight control, 2) inability to reactively restore stable flight in a precise manner, 3) lack of low latency attitude recovery with fast response times, and 4) inefficient usage of limited battery power. In light of these challenges, this paper presents smARTflight: an environmentally-aware, Adaptive and Real-Time flight management system. smARTflight leverages sensor data processing and existing flight control functionality of the autopilot to autonomously counteract adverse effects of environmental disturbances on the drone's attitude. This is achieved by smartly adapting the execution rates of critical flight controller tasks, while guaranteeing their real-time and safety-critical scheduling constraints.

Current state-of-the-art autopilot systems combine safety-critical flight control functionality with non-safety-critical tasks such as camera data processing for on-screen displays (OSDs), telemetry data transmission, and blackbox data logging onto a common hardware platform. Fig. 1 characterizes an example software stack managed by a scheduler, along with the essential hardware modules common to real-world copter autopilot architectures.

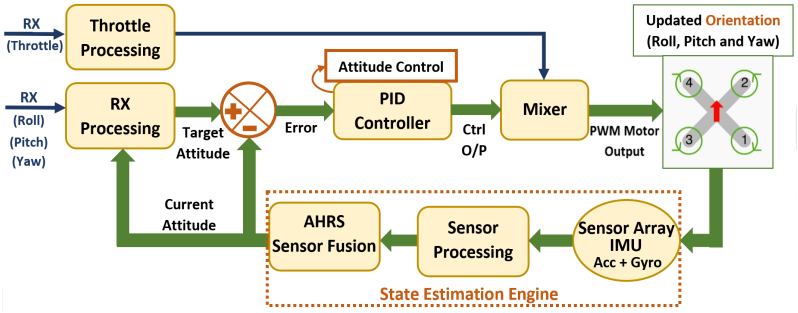


Figure 2 Components of a flight control loop in an autopilot.

The critical flight control tasks in charge of low-level attitude stabilization are often tightly coupled within a closed loop. These tasks are the core constituents of the widely used linear negative feedback control technique, depicted in Fig. 2. The loop involves high frequency sampling for acquisition and processing of data from multiple sensors. It then employs a series of filters and complex sensor fusion algorithms to estimate the current orientation, or attitude of the drone, as part of the state estimation engine. This data is compared with the target attitude received via the RX module, and the error is fed to the PID (Proportional, Integral and Derivative) controller. The three regulators transform the error between the actual and desired angular states into control signals for the electronic speed controllers (ESCs). Controller output commands mixed with the input throttle data are then used to adjust individual motor speeds and rotor rotations per minute (RPMs). This compensates for actual versus desired maneuvers of the copter. A combination of differential angular velocities results in an applied net torque about the center of gravity of the copter, causing the system to undergo Euler angle rotations involving roll, pitch and yaw, with respect to the Earth's reference.

To aid flight control, the system is equipped with: 1) a network of sensors (e.g. accelerometers and gyroscopes collectively known as the Inertial Measurement Unit (IMU), magnetometers, barometers, sonars, cameras, GPS and so forth), 2) a complete power train with motors, rotor propellers and ESCs, and 3) a power distribution and regulation sub-system.

Collectively the entire system ensures stable and accurate flight control under steady-state conditions. However as previously mentioned, current autopilot system designs are highly sensitive to external changes in flight dynamics [11, 24] and thus fall short when subjected to transient attitude disturbances. smARTflight's rate-adaptive, criticality-aware real-time scheduling strategy overcomes the stability challenge by augmenting existing flight controllers with enhanced environmental awareness. It improves flight stability and performance by adopting a modular and structured approach to achieve deterministic, timing predictable and adaptable flight control.

The rate-adaptation policy is built upon a key insight that performance of a flight controller is directly related to the rate of execution of its critical flight control loop [10]. When the drone experiences frequent changes in its orientation, a high frequency sampling rate of the sensors captures the most recent data representing the state of the drone. This allows the underlying PID controller to consequently correct for the current attitude error in all three navigation dimensions, namely roll, pitch and yaw.

The controller is therefore able to closely monitor the instantaneous forces acting on the copter at any point in time. If the external forces on the copter vary significantly then a correspondingly higher frequency of updates to the motor speeds leads to a much finer

granularity and accuracy of attitude corrections and control. Alternatively, infrequent sensor data processing and state estimation leads to inaccurate attitude predictions, resulting in potentially incorrect motor speed adjustments to maintain a target flight trajectory.

In contrast, stable weather conditions impose little to no anomalous attitude variations, which presents an opportunity to reduce the rate of executions of the control loop. This allows efficient management of compute resources making them available for other non-critical mission-level functionality such as object detection and tracking [31], camera data processing for obstacle avoidance [12] and possible way-point navigation. Brushless DC motors responsible for generating the necessary lift force to maintain flight are connected directly to the main battery power and are the primary energy consumers on the drone. A reduction in the required updates to motor speeds therefore ensures energy efficient flight control.

In summary, this paper does not focus on any specific flight control algorithm for multi-rotor UAVs, nor does it propose gain scheduling techniques for PID based controllers. Instead, we modify the rates of execution of existing control tasks with their default tuning parameters to enhance stability in the presence of disturbances. This work lays the foundation to empower autopilot software stacks with criticality-aware rate adaptation and real-time execution behavior.

Our flight management framework: 1) identifies tasks as safety-critical and non-safety-critical within a well-known autopilot system, 2) identifies operating frequencies for individual tasks according to the system's current criticality mode, 3) autonomously reasons about the varying external conditions to dynamically adapt task execution rates in a principled manner, 4) ensures real-time management and accounting of processor cycles, and 5) guarantees hard-set execution time bounds for all flight controller tasks, to ensure flight success in the presence of timing uncertainties. smARTflight compensates for instantaneous changes in the environment within predictable time bounds, ensuring low latency responsiveness to critical external events, preventing crashes and expediting recovery from anomalous attitude shifts. Compared to static-rate autopilots, the one adopted by smARTflight avoids unnecessary control loop executions in comparatively stable weather conditions, to save battery power and free computational resources for additional tasks.

The rest of the paper is organized as follows: Section 2 describes the background to the *Cleanflight* (CF) [7] flight control firmware used in this work. Included are the details of the vanilla scheduling policy and the resulting shortfalls of the state-of-art algorithm. Section 3 describes smARTflight's execution model, system and task criticality semantics and the mode-change policy. Section 4 details the experimental setup and evaluation of smARTflight's performance against the vanilla Cleanflight system. Finally Section 5 discusses related work and Section 6 concludes the paper.

2 Background

Autopilots. The open-source community of multi-rotor flight controllers features a rich set of projects tailored to either autonomous way-point navigation, such as Ardupilot [3], PX4 [22] and iNav [14] or first-person-view (FPV) drone racing such as Betaflight [5] and Cleanflight [7]. Despite their differences, the autopilots host more or less similar flight control logic as the standard feedback loop depicted in Fig. 2.

smARTflight works on the principle of general applicability across all autopilot firmwares by extending existing control logic with run-time adaptability of task execution rates. As an example implementation, we retrofit one of the most popular configurable autopilots,

Cleanflight (CF), with smARTflight's scheduling architecture. Designed specifically for racing quadcopters, CF maintains a competitive edge over other popular open-source flight controllers, in terms of flight efficiency, functional reliability and controller performance. Combined with a functionally robust and minimalistic flight control stack it proves to be an ideal autopilot platform to show smARTflight's performance benefits on cost-effective resource constrained embedded hardware.

Despite its advantages, Cleanflight is inflexible in the way it operates. Flight control tasks are defined with static time periods that only act as soft time bounds. The underlying scheduler is also based on a non-preemptive, best-effort scheduling policy. The lack of timing predictability is one of the key contributors to variable task dispatch times, which often manifests as a job starting earlier than expected or finishing later than required. Delay variations between the release times of jobs for the same task become a major hindrance to predictable adaptation within the system. The absence of strict timing constraints thus makes task execution non-deterministic leading to a negative impact on the safety and robustness of flight control. To avoid catastrophic failure under external disturbances, task times must therefore be precisely controlled [13]. We verify this inherent uncertainty in task execution times through real-world experiments with Vanilla CF as Phase-I of our evaluations (Section 4). The next section provides a description of the Vanilla CF implementation.

2.1 Cleanflight Tasks

Table 1 List of essential Cleanflight Tasks: (**bold** indicates critical tasks involved in low-level flight control).

Task Name	Time Period (μs) Cleanflight/ smARTflight (LO)	Execution Frequency (Hz)	Static Priority (Vanilla CF)	Criticality (smARTflight)	Description
TASK_SYSTEM	100,000	10	Med-High	LO	Report system statistics
TASK_BAT_VOLT	20,000	50	Medium	LO	Sample battery voltage
TASK_GYROPID (Looptime)	**4,000 / 2,000 / 1,000**	**250 / 500 / 1,000**	**Real-Time** (highest)	**HI**	**Sample Gyroscope + PID-based motor control**
TASK_ACCEL	**1,000**	**1,000**	**Medium**	**HI**	**Sample Accelerometer data**
TASK_ATTITUDE	**10,000**	**100**	**Medium**	**HI**	**Calculate current attitude**
TASK_RX	20,000	50	High	LO	Process receiver commands
TASK_SERIAL	10,000	100	Low	LO	Serial communication with the ground computer

Cleanflight (v.2.3.1) features 31 tasks in total, of which more than half constitute optional add-on functionality. The critical flight control functionality is distributed over a set of 3 tasks: TASK_GYROPID, TASK_ACCEL and TASK_ATTITUDE. The core control loop (Fig. 2) consists of: 1) a receiver task (TASK_RX) that processes reference inputs for the target roll, pitch and yaw attitudes, 2) separate PID controllers for each axis of rotation that adjust the output response based on the current attitude error (P gain), an accumulation of past errors (I gain), and the rate of change in error (D gain), 3) a mixer component that determines the magnitude of the thrust force applied to each motor, 4) several sensor processing tasks (TASK_ACCEL and TASK_GYROPID), and 5) a quaternion-based Attitude and Heading Reference Sub-system (AHRS: TASK_ATTITUDE) that combines sensor data using Madgwick & Mahony's complementary filter algorithm [29, 32] to compute the current attitude of the copter.

Throttle processing is incorporated within TASK_RX. A radio or ground control station transmits commands wirelessly, which are then received by the hardware RX module on the drone. TASK_RX varies in its importance to the overall flight control mechanism, depending on the update frequency of target attitudes often dictated by the drone application. For our experiments in Section 4, we consider TASK_RX a low importance and, hence, low-criticality task since our target remains fixed throughout the flight of the drone. The essential tasks are listed in Table 1 along with some low-criticality book-keeping functions. The table also records `static priorities` and `time periods` that are used by the Vanilla CF scheduler to determine task dispatch order at runtime.

We note that TASK_GYROPID incorporates a chain of sub-tasks, enumerated earlier as item points 2), 3) and 4) of the core control loop. This task sequentially samples gyroscope data, executes the PID controller algorithm and updates motor output commands. TASK_GYROPID thus forms a *fast* loop that allows a user-configurable rate of execution known as the base *looptime* in CF terminology. Looptime therefore represents the time period for one iteration of the fast control loop. Other critical tasks in the main control loop, namely (TASK_ACCEL and TASK_ATTITUDE), work at non-configurable frequencies that are fixed integral multiples of the base looptime.

2.2 Vanilla Scheduler

Algorithm 1 details Vanilla CF's non-preemptive scheduling policy. The scheduler maintains a fixed ready queue of tasks in decreasing order of static priorities from Table 1. Tasks are scheduled from highest to lowest dynamic priority, which is calculated at run-time for each task as a product of its static priority and the elapsed time since last execution (task age-cycles) (Lines 6–7). The queue is traversed on every scheduler invocation and the task with the highest dynamic priority is dispatched for execution according to Lines 9–11. The chosen task either has a real-time static priority, or it has a lower static priority and has aged for at least two consecutive time periods.

Algorithm 1 Vanilla Cleanflight scheduler.

Require: *task* parameters: $lastExeTime$, $staticPeriod$, $staticPrio$
Require: rtTaskRunnable
1: **procedure** SCHEDULE
2: $curTime = get\_time\_micro()$
3: $selTask =$ **nil** and $selTaskDynamicPrio = 0$
4:
5: **for all** *tasks* in *taskQueue* **do**
6: update $task \rightarrow ageCycles = \frac{curTime - task \rightarrow lastExeTime}{task \rightarrow staticPeriod}$
7: calculate $task \rightarrow dynPrio = dynamic\_prio(task)$
8: /* Task with highest dynamic priority is selected */
9: **if** $task \rightarrow dynPrio > selTaskDynamicPrio$ **then**
10: **if** $task \rightarrow staticPrio ==$ **Real-Time or**
11: {!rtTaskRunnable **and** $task \rightarrow ageCycles > 1$} **then**
12: $selTask = task$
13: $selTaskDynamicPrio = task \rightarrow dynPrio$
14: **end if**
15: **end if**
16: **end for**
17: **if** selTask **then**
18: execute selected task function: $selTask \rightarrow taskFunc()$
19: Update $task \rightarrow lastExeTime$
20: Reset $selTask \rightarrow ageCycles$ and $selTask \rightarrow dynPrio$
21: **end if**
22: **end procedure**

We note that in Vanilla CF's context "real-time" does not impose any temporal constraints on a task but is used instead to represent the highest static priority. The dispatched task runs to completion and only cooperatively yields control back to the scheduler at the end of its execution. Depending on the execution time of a task, the time between consecutive scheduler invocations may vary considerably.

3 smARTflight Execution Model

This section formalizes smARTflight's system and task model and details the execution semantics.

Motivation

Higher execution rates offer finer granularity of control thereby reducing the convergence time for a drone to asymptotically settle to its steady-state target attitude. With smARTflight, we are motivated to maximize the benefits associated with a high execution frequency of the main flight control loop in adverse environmental conditions, while avoiding unnecessary over-provisioning in comparatively calm conditions. smARTflight therefore adapts individual rates of all the critical flight control tasks in addition to the fast loop's looptime. To this end, we introduce an explicit notion of task and system safety criticality into Cleanflight. A description of the real-time task and system model is presented next.

3.1 Task Model

The system is modeled as a set of real-time periodic tasks, $\{\tau_1, \tau_2, \ldots, \tau_n\}$, which are scheduled according to an extension of the *Liu & Layland* model [6]. In our system, each task, τ_i, is parameterized by a 5-tuple $\{C_i,\ \big[T_i(\text{LO}), T_i(\text{HI})\big],\ \big[D_i(\text{LO}), D_i(\text{HI})\big],\ L_i,\ \big[p_i(\text{LO}), p_i(\text{HI})\big]\}$, with each term defined as follows:

- C_i: *worst-case computation time, or budget.* The computational logic and structure of a task remains unaltered. This implies that the execution time also remains more or less the same across multiple job instances of the same task. Cleanflight is a closed system with a fixed total number of tasks. We determine the run-time budget by profiling the system online at every system start-up. This budget value is then used to calculate per task utilization ($U_i = \frac{C_i}{T_i}$) which is then subsequently used to compute processor utilization (U_{sys}).
  C_i is computed pessimistically as a upper bound of the task's actual computation time by integrating all possible interrupt overheads that may be charged to the task's runtime budget due to I/O and memory requests, in addition to the scheduler overhead within the base time.
- $\vec{T_i} = \big[T_i(\text{LO}), T_i(\text{HI})\big]$: *a vector of time periods.* Each task τ_i has a corresponding period T_i for each criticality level in the system, where $L_{sys} = \{\text{LO}, \text{HI}\}$ is the set of system criticality levels. Tasks explicitly modify their time periods across system mode changes. System modes are discussed in Section 3.2. Each time period is a multiplicative inverse of the corresponding task rate (R_i) : $T_i(L_{sys}) = \frac{1}{R_i(L_{sys})}$.
- $\vec{D_i} = \big[D_i(\text{LO}), D_i(\text{HI})\big]$: *a vector of deadlines.* A job's deadline is relative to its release instance. Each deadline occurs $T_i(L_{sys})$ time units after the job's arrival time, implying $D_i(L_{sys}) = T_i(L_{sys})$.
- $L_i = \{\text{LO}, \text{HI}\}$: *task criticality level.* A task is assigned static criticality as described in Table 1. In this paper, we consider only two task criticality levels. However, smARTflight is able to support more than two levels when finer-grained task rate adaptations are

required across system mode changes, to compensate for environmental factors. This would allow the system to exhibit a more graceful transitional response to varying exogenous conditions.

- $\vec{p_i} = \big[p_i(\text{LO}), p_i(\text{HI})\big]$: *a vector of task priorities* assigned under the *Rate-Monotonic* priority assignment algorithm (RMS) [6]. Tasks with higher rates and, hence, shorter time periods, are assigned higher priorities than tasks with longer periods. Tasks therefore have different priorities in different system modes.

All Cleanflight tasks are modified to be preemptible in smARTflight. Thus unlike Vanilla CF, their execution is interleaved. We present our scheduling framework that supports this task model in Section 3.3.

Task criticality (L_i) is defined as a measure of the task's functional importance to the overall flight control operation. HI criticality is associated with tasks that must operate correctly within the real-time temporal bounds of their budget, C_i, and period, T_i, in order to maintain stable flight and avoid crashing the drone. All flight-control tasks shown in **bold** in Table 1 are assigned to this level. In contrast, tasks that have minimal impact to the runtime flight control functionality are assigned a LO criticality. Examples of such tasks include blackbox logging, camera data capture, and serial transmission.

Task criticality allows us to directly associate one of the two rate-adaptation behaviors, *rate increase (↑)* or *decrease (↓)*, with each task across the two system execution modes. A HI criticality task's frequency increases on a LO → HI mode transition of the system. Inversely, a LO criticality task's frequency decreases. This counteracts the increase in execution rate for HI criticality tasks by acting as a protection mechanism against potential system overload situations. However, smARTflight optionally allows LO criticality tasks to retain their current rate of execution on a LO → HI mode transition, if real-time task schedulability is maintained according to the RMS utilization bound. Notwithstanding, all task rates are reset back to the original LO mode values when the system transitions from HI → LO mode. Table 2 summarizes the relationship between task rates in terms of the time periods (T_i) for both LO and HI criticality tasks in each system mode.

Table 2 Relationship between task time periods (T_i) for both LO and HI criticality tasks in each system mode (L_{sys}).

LO Criticality Tasks	**HI** Criticality Tasks
$T_i(L_{sys} = \textbf{LO}) \leq T_i(L_{sys} = \textbf{HI})$	$T_i(L_{sys} = \textbf{LO}) > T_i(L_{sys} = \textbf{HI})$

3.2 System Model

For the purposes of this paper, the system is characterized by two distinct steady-state execution modes, or system criticality levels, referred to as LO and HI. System criticality captures the direct influence of external disturbances on the attitude of the drone. Each mode is therefore defined in terms of the captured environmental dynamics and the corresponding effects on the stability of flight.

On every iteration of the control loop, environmental data, as reported by the navigation sensors, is sampled and processed to compute an updated value for the drone's current attitude in each axis of rotation: roll, pitch and yaw. The output of the state-estimation engine is then used to trigger a particular system execution mode at millisecond granularity. Each mode is activated as a complementary response to variation in attitude when compared against the corresponding angular thresholds in all three dimensions. Since each axis is

independently subjected to environmental influence, we identify three independent Euler angle thresholds that represent upper bounds on the maximum tolerable deflection from the copter's target attitude along that axis. If the fluctuation goes beyond the maximum bound in *any* one of the axes, the system switches to HI mode. The system reverts back to LO mode only if the intensity of variations falls below the predefined threshold in *all* three axes. This ensures low latency response times in all dimensions. A typical scenario is the absence or presence of high winds that directly translates to calm (LO) versus adverse (HI) environmental conditions.

The flight controller tasks are characterized by different execution rates, as a function of system mode and individual task criticality. Tasks gracefully adapt their rates across mode-switches as previously described in our task model. The system mode acts as a flag to trigger an increase or decrease of the task execution rates, which in-turn ensures a timely and adaptable flight control response to the changing environment. The system always starts in LO mode, with subsequent LO → HI and HI → LO transitions occurring as a result of environmental conditions.

Mode Transition Protocol

Mode change requests are modeled as asynchronous events within the system. These are flagged in the control loop iteration following their arrival. smARTflight's version of TASK_ATTITUDE registers the mode change request with minimal delay, by comparing angle thresholds with current attitude values for each axis. The system is then able to react to changing conditions with very low mode-switching delay, according to the following mode transition definitions:

- LO → HI: Transition begins with the arrival of the mode change event and ends when all the HI criticality tasks have increased their rates of execution, or equivalently decreased their time periods, from $T_{i|\{L_i=\text{HI}\}}(\text{LO})$ to $T_{i|\{L_i=\text{HI}\}}(\text{HI})$. The algorithm waits for every HI criticality task to complete its LO mode execution before updating its time period, thereby preserving the timing properties of the task across the system mode change. This policy adopts a graceful transition of rates, thus maintaining task schedulability in real-time. After every switch, the processor's utilization (U_{sys}) is re-calculated and compared against the maximum allowed utilization. On exceeding the bound, excess utilization is compensated by decreasing the rates of execution, or equivalently increasing the time periods, of all the LO criticality tasks from $T_{i|\{L_i=\text{LO}\}}(\text{LO})$ to $T_{i|\{L_i=\text{LO}\}}(\text{HI})$. This is done in consecutive iterations until the updated taskset regains a feasible schedule. Increasing a LO criticality task's period only serves to raise the likelihood of regaining a feasible schedule. Consequently, the time periods for LO criticality tasks are increased without waiting for their prior LO mode executions to complete. The current job execution is thus carried over from LO to HI mode without disruption.
- HI → LO: Follows by symmetry of argument presented above.

Both system mode transitions are represented in algorithmic form in Algorithm 2. A new mode change request is only serviced if the system is in either of the two steady-state system modes. This ensures non-overlapping and graceful transitions between task rate parameters. To avoid frequent oscillations between LO and HI states, consecutive mode changes are temporally spaced out in an artificial manner. The amount of delay is configured by the programmer as part of the system tuning process. We next discuss smARTflight's adaptive real-time scheduling policy that replaces Cleanflight's vanilla scheduler in our example implementation.

Algorithm 2 smARTflight's rate-adaptation policy.

Require: taskQueue[] with tasks arranged from high → low priority/rate
Require: *current_task*
Require: $T\_decrease$ and $T\_increase$
Require: U_{bound}^{RMS}
1: **if** LO → HI **then**
2: **if** current_task → criticality == HI & *current_task* yielded **then**
3: $T_i^{\text{HI}}(L_{sys} = \text{HI}) = T_i^{\text{HI}}(L_{sys} = \text{LO}) - T\_decrease$
4: Insert task in taskQueue[] at new position based on updated rate
5: Update task and system utilization: U_i^{HI} & U_{sys}
6: **end if**
7: **while** $U_{sys} > U_{bound}^{RMS}$ **do**
8: **for all** LO tasks ∈ taskQueue[] **do**
9: $T_i^{\text{LO}}(L_{sys} = \text{HI}) = T_i^{\text{LO}}(L_{sys} = \text{LO}) + T\_increase$
10: Update task utilization : U_i^{LO}
11: Update $U_{sys}(L_{sys} = \text{HI})$
12: **end for**
13: **end while**
14: **else if** HI → LO **then**
15: **for all** tasks ∈ taskQueue[] **do**
16: Restore T_i(LO) ▷ mirrored logic from LO→HI
17: Insert task in taskQueue[] at old position
18: Update task utilization: U_i
19: **end for**
20: Recalculate $U_{sys}(L_{sys} = \text{LO})$ and check against U_{bound}^{RMS}
21: **end if**

3.3 Rate-Adaptive Real-Time Scheduling Algorithm

smARTflight extends Cleanflight's task model with real-time constraints to ensure predictable execution. In accordance with the task model presented in Section 3.1, each periodic task (τ_i) generates an infinite sequence of jobs at run-time. Successive jobs are spaced $T_i(L_{sys})$ time units apart and execute for at-most C_i time units before completing by their deadline at the end of their period. We leverage *Liu & Layland's* real-time Rate Monotonic Scheduling (RMS) algorithm [6] to dispatch tasks for execution according to their statically assigned priorities, based on their respective rates.

Schedulability of a taskset under RMS depends on the utilization bound test, i.e., tasks are guaranteed to meet their deadlines if the CPU utilization (U_{sys}) is below the RMS bound. The scheduler is invoked at each time quantum boundary, whose interval is determined as a function of all tasks' rates. Compliance with the bound is checked and the ready-queue is inspected. The task with the highest rate (priority) is dispatched for execution if current-time ≥ task release time. This real-time variant of Cleanflight is termed RMS CF in our experiments.

Rate Adaptation Scheduling Policy

We extend smARTflight's RMS scheduler with additional functionality that: 1) adapts task rates across mode changes according to the model presented in the previous section, 2) dynamically updates the ready-queue of the real-time scheduler and reprograms the interval timer, and 3) re-configures task dispatch behavior at run-time. The criticality-aware scheduler updates priorities and rearranges tasks in the ready queue according to the assigned execution rates. Per-task utilization is calculated as a modified ratio between the budget and current time period: $U_i = \frac{C_i}{T_i(L_{sys})}$. The RMS schedulability test is performed during each system mode transition window (Algorithm 2, Line 7) as an additional check to trigger CPU utilization adjustments in case of transient system overload. This is achieved by modifying rates of execution of LO criticality tasks according to the required system utilization.

4 Evaluation

We deploy a real testbed to validate our proposed framework and conduct experiments with a high-performance racing quadcopter drone. Details of our hardware setup common to all experiments are presented in the next section. We conduct our analysis in *three* distinct phases. **Phase I** (Section 4.2) investigates Vanilla CF's lack of timing guarantees as a consequence of imprecise task execution and non-real-time scheduling logic. This paves the way for RMS CF. In **Phase I-I** (Section 4.3), we determine the effect of statically varying execution frequencies of critical flight control tasks on the response time to achieve a target hover attitude. Performance benefits of RMS CF when compared with Vanilla CF are shown in **Phase II** (Section 4.4). Finally, we test smARTflight in **Phase III** (Section 4.5) and demonstrate the benefits of the rate adaptive approach over both Vanilla and RMS CF.

4.1 Experimental Setup

Quadcopter Hardware

We use a custom-buit QAV250mm quadcopter (Fig. 3) for our hardware-in-the-loop simulations and experiments. Current to the four brushless DC motors is regulated by the Electronic Speed Controllers (ESCs) according to analog Pulse Width Modulation (PWM) signals sent by the autopilot.

We flash the autopilot firmware on the popular SPRACINGF3 [8] Acro flight controller board, featuring an STM32F303 microcontroller with an ARM Cortex®-M4 core clocked at 72MHz. The board features a 6 degrees-of-freedom IMU (16-bit 3-axis gyroscope/accelerometer), general-purpose IO ports for communication with the radio receiver and motors, and 8MB flash storage for flight logs.

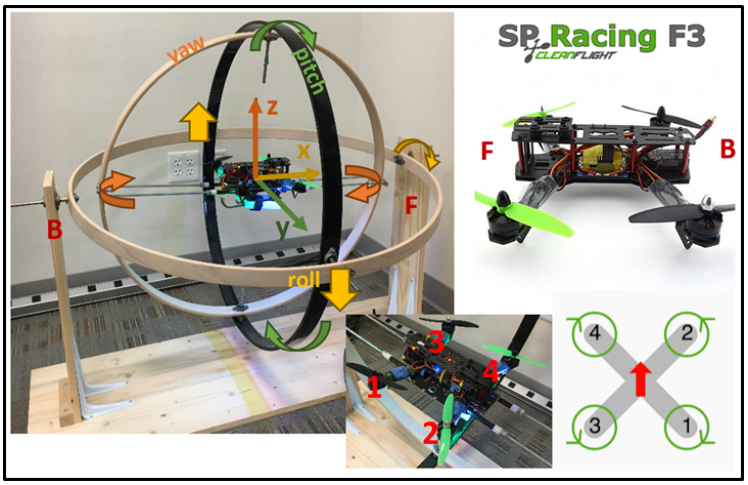


Figure 3 BirdCage testbed for QAV250mm quadcopter with an SPRACINGF3 flight controller board. The orientation of the motors enumerated and configured within Cleanflight [7] is shown on the bottom right.

The drone's carbon-fiber frame and light-weight components ensure aerodynamic efficiency and flexibility of flight. The overall mechanical structure gives the quad a competitive edge to be used in drone racing applications. It thus makes for an ideal platform to focus on improving flight controller performance.

Autopilots

As stated earlier, smARTflight builds upon existing autopilot firmware. Any improvements in flight performance compared to a vanilla autopilot implementation aim to show the benefits of real-time, rate-adaptive task execution. For our case-study in this work, we thus consider three autopilot firmwares: 1) *Vanilla CF*, comprising a default non-real-time scheduler, 2) *RMS CF*, featuring a real-time, fixed-priority rate-monotonic scheduler (RMS), and 3) *smARTflight*, which has a criticality-aware rate-adaptive policy extension to RMS.

Each autopilot is pre-configured to run a subset of the most essential tasks from Cleanflight, listed in Table 1. We stripped away unnecessary functionality from the autopilot to reduce the memory footprint for our embedded board. In addition to all flight controller tasks, we include some functionality for book-keeping, configuration, and collecting system statistics. In smARTflight, these auxiliary tasks are categorized as LO criticality, while all flight controller tasks are classified as HI criticality (bold font in Table 1).

BirdCage

For our real-world experiments, we pivot the quadcopter at the center of a custom-made mechanical gyroscope called the *BirdCage* (Fig. 3). Three rotating gimbals are mounted orthogonal to one another, allowing the drone to rotate freely about its roll, pitch and yaw axes. We perform controlled and reproducible step attitude disturbances emulating wind effects by displacing the axial rings a constant angle. The rig's design alters physical dynamics of the drone thus posing some additional challenges:

- *Torque about the center of gravity due to the inner most ring* – We compensate for the added torque by using four EMAX 2300kV brushless DC motors (MT2204). Each motor exerts a maximum thrust of ~ 400 grams under no-load conditions. This amounts to a total of 1.6kg of total upward thrust and is enough to counterbalance the downward force of the drone and the inner ring combined.
- *Pendulum effect* – Displacing the roll and pitch axial rings from the target (hover) attitude instills gravitational potential energy in the corresponding ring. This leads to simple harmonic motion of the drone in the displaced axis. The erroneous oscillations are sampled by the sensors and fed to the PID controller as attitude error in need of correction. At higher execution rates, the effect becomes more pronounced as the motor speeds are adjusted more frequently to counter the momentum gained by the pendulum. This causes the copter to continuously over-correct its attitude. We compensate for this in our experiments by calibrating the sensors to consider a higher gravitational potential point as its target hover, and by displacing the roll and pitch axial rings to the lowest potential point instead. This removes the pendulum effect from the system, significantly reducing the number of oscillations.

Metrics and Settings

To study the performance of our system against both versions of Cleanflight, we record attitude variation profiles of the copter over time, in response to step input disturbances. We calculate *response time* to achieve a steady-state target attitude by computing the difference between the time the copter is subjected to an initial step disturbance along a particular axis of rotation and when it stabilizes within $\pm 5°$ of the target. The $\pm 5°$ steady-state error band compensates for sensor imprecision and calibration, inaccuracies in the drone hardware and

granularity of the motor outputs. It also restricts the maximum tolerable worst-case offset in achieving the target attitude. We additionally analyze the *absolute error* accumulated over the entire course of the attitude adjustment of the drone to understand how smARTflight's rate adaptation policy impacts accuracy of flight control.

To draw conclusions about power usage and energy efficiency, we sample PWM commands sent to the motors and compute the minimum, maximum and average duty-cycle of one of the motors on the drone. Duty-cycle is represented as a percentage of high (on) time of the signal over the time period (reciprocal of motor update frequency). It is directly proportional to the power applied to the motors. In particular, we consider values for the bottom-left motor: motor-3 (Fig. 3) as it is involved to a high degree in both roll and pitch corrections. Commands sent by the flight controller to the other three motors are either roughly equivalent or less than motor-3's duty-cycle in all test cases.

For all experiments, we consider hover ($0° \pm 5°$ tilt with respect to the stationary frame reference) to be our stable steady-state target in both roll and pitch axes. The flight mode is set to self-level as opposed to manual rate-mode. We note that smARTflight's adaptive control is independent of any autopilot flight-mode. Our setup removes human input, thus isolating all benefits achieved with smARTflight alone. PID controller constants are initially tuned to achieve a desired control response in either axis and kept fixed across all experiments for comparison.

We conduct experiments along two axes: roll (**Exp Roll-Left**) and pitch (**Exp Pitch-Back**). Cleanflight's flight control logic for yaw is inherently limited in its capability. Due to a lack of support for self-level mode, all attitude corrections require manual input. We, therefore cannot show smARTflight's improvements in the yaw dimension. Nevertheless, implementation of smARTflight optimizes performance in all three rotational dimensions, when the underlying flight control logic does not impose any limitations of its own.

We vary critical task rates and repeat each experiment at-least 3 times. For each run, the roll and pitch rings of the BirdCage are displaced to a maximum angle of 15° (step disturbance to the system). The copter is then allowed to stabilize to target hover and blackbox data is recorded. We plot the step response profiles of the attitude adjustments over time and determine steady-state response time values for each run. Based on the average response time across runs, we choose the profile with the minimum variance, to be the representative result for a particular experiment.

4.2 Phase I: Vanilla CF

The lack of timing guarantees in the Vanilla CF scheduling policy leads to variations in task runtime frequencies. We investigate the effect of different looptimes (refer to Table 3) on the rate of execution of the low priority receiver task: TASK_RX (statically set rate = 50Hz). The motor update frequency for the PWM protocol is kept fixed at a maximum of 500Hz. We instrument Vanilla CF to toggle GPIO pins on the STM32F303 board at every motor and receiver update. The signal trace is viewed on an oscilloscope over time and measurements of runtime frequency for receiver updates, averaged over ≈450 iterations of the control loop, are reported in Table 3.

The average rate of execution of the lower priority task, TASK_RX, varies between the statically set 50Hz to a maximum rate of 11kHz depending upon the configured looptime. We also observe runtime jitter on the oscilloscope traces between consecutive task release instances. This shows a high degree of dependence between runtime execution frequencies of low and high priority tasks in the system. For a deterministic and predictable system, task executions must comply with strict time bounds. We thus replace the vanilla scheduler with a real-time scheduling policy in RMS CF.

Table 3 Task rates for Vanilla CF simulations.

Static Rates			Measured Runtime Rates	
Looptime (Hz)	Motor Rate (Hz)	RX Rate (Hz)	RX Rate (Hz)	Motor Rate (Hz)
500	500	50	11.1k	485.0
1k	500	50	11.1k	476.6
4k	500	50	6.46k	381.6
8k	500	50	49.8 ≈ 50Hz	376.2

Table 4 All possible task rate parameters for exploratory experiments with **Vanilla CF**. Note: 25Hz for TASK_ATTITUDE does not always lead to attitude stabilization. It is thus an invalid setting and not considered for other looptimes.

Critical Tasks	Default Rates (Hz)			Custom Execution Rates (Hz)									
GYROPID/**Looptime**	1000	500	250	1000			500			250			
ACCEL	1000			1000			500			250			
ATTITUDE	100			200	100	50	200	100	50	25	200	100	50
Roll: Avg. Response Times (s)	13.5	18.5	21.5	14	13.5	21.5	33	16.5	20	33	33	32.5	26.5

4.3 Phase I-I: Performance Analysis of Vanilla CF with Different Looptimes

We next conduct a set of exploratory experiments on the BirdCage with Vanilla CF to determine two possible sets of feasible task execution rates to be used in smARTflight; one per mode of the system. The two requirements are: 1) the *highest frequency* looptime that leads to the best response time performance when impacted with external attitude disturbances. This is upper bounded by the maximum sensor sampling rate and the motor update protocol; 2) the *lowest frequency* looptime that allows the copter to maintain accurate stable flight, while sparing any unnecessary processing and freeing up as many system resources as possible.

We focus on varying the execution rates of three highly safety critical tasks of the flight control loop: TASK_GYROPID, TASK_ACCEL and TASK_ATTITUDE, collectively represented under the looptime value. Sets of all possible task rates considered are listed in Table 4. For the *first set of experiments* we vary only the TASK_GYROPID rate: 250Hz, 500Hz and 1000Hz. TASK_ACCEL and TASK_ATTITUDE are fixed at their default rates of 1000Hz and 100Hz, respectively. This sets the baseline performance for Vanilla CF. We measure average response time for each of the three rate specifications and report them under the "Default Rates" column in Table 4. Without any modifications to Vanilla CF, accurate attitude is attained with the lowest response time of 13.5s at a looptime frequency of 1000Hz.

For the next batch of experiments, we vary TASK_ACCEL and TASK_ATTITUDE rates in addition to the looptime. The different rate parameters, along with the corresponding average response time results are summarized in Table 4. We shortlist the best average response times for each of the three looptime specifications (highlighted in the table) and compare their step-response attitude profiles in Fig. 4. Lowest response time is achieved with looptime=1000Hz and the highest with looptime=250Hz.

At 250Hz, we observe sharp corrections in the roll attitude both in the short and long term, which are manifested as thicker trace lines and abrupt variations in the response time profile (Fig. 4). In contrast, at 1000Hz, the trace is much smoother. This is evidence of a finer granularity of attitude control at higher rates of execution. At lower rates, the flight controller uses stale roll values to compute motor commands as consecutive attitude updates are spaced further apart in time. This results in a larger magnitude of required corrections on the next iteration of the flight control loop. It also leads to slower response times in achieving the target.

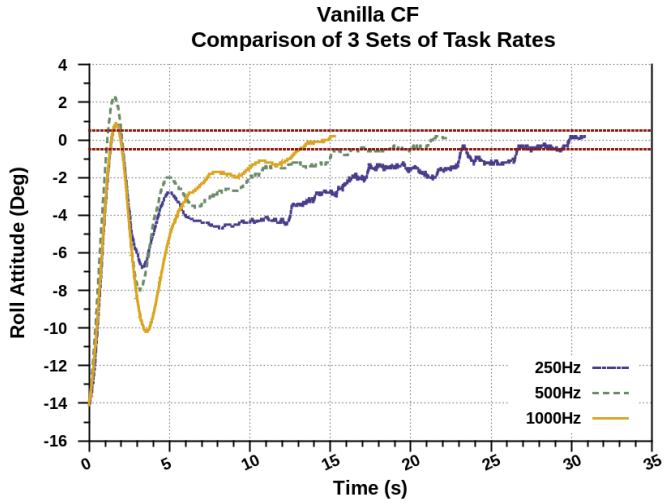


Figure 4 Exp: Roll-Left, Comparison between the best three task rate specifications of **Vanilla CF**. Lowest response time=*13.5s* with looptime=1000 Hz and maximum response time=*26.5s* with 250 Hz. Looptime=500Hz gives a response-time of *15.5s*.

We note that accuracy and response time of the flight controller is dependent on the correctness and frequency of attitude calculations. In particular, the ratio between sensor and attitude task frequencies determines the accuracy. For each of the three highlighted sets of rates from Table 4, the accuracy ratio is 10, 5 and 5, respectively. This implies that the steady-state final attitude value for looptime=250Hz is relatively less accurate than for 1000Hz. Since stable environments lead to little or no variations in sensor data, lower looptimes can still maintain accurate attitude values. This presents an opportunity to spare unnecessary loop executions that can free up CPU resources without compromising on flight stability. However, practical limitations don't allow a value less than 250Hz, below which the copter cannot maintain flight. We thus choose 250Hz as the lowest frequency looptime.

Table 5 Exp: Pitch-Back, Comparison between two rate specifications of **Vanilla CF**: Looptime=1000 Hz & 250 Hz. Low response time is achieved with a higher rate of execution of the control loop.

Looptime	**1000 Hz**	**250 Hz**
Avg. Response Time (s)	10	37

In contrast, rapidly changing environmental dynamics require the flight controller to execute at the maximum feasible frequency, which counteracts external disturbances in the attitude in a timely and accurate manner. We thus choose 1000Hz as the highest frequency looptime. The pitch axis steady-state response time results with the two shortlisted task rate sets are presented in Table 5. Response times for pitch are proportional to roll, as expected, for each looptime specification.

4.4 Phase II: Comparison between Vanilla and RMS CF

Fig. 5 reports the average response times for both roll and pitch experiments, repeated with RMS CF using the rate parameters from the last section. RMS CF follows a similar trend

as Vanilla CF, with the best case response time of 9s in the pitch axis using a looptime of 1000Hz. We present a comparison of attitude profiles between both Cleanflight systems in Fig. 6a and Fig. 6b.

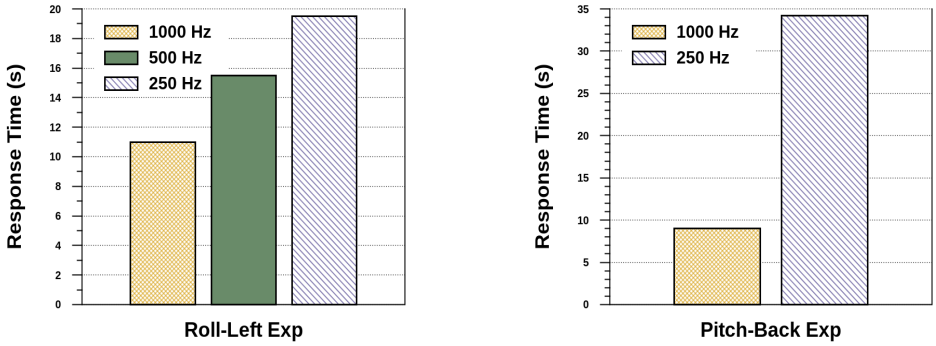
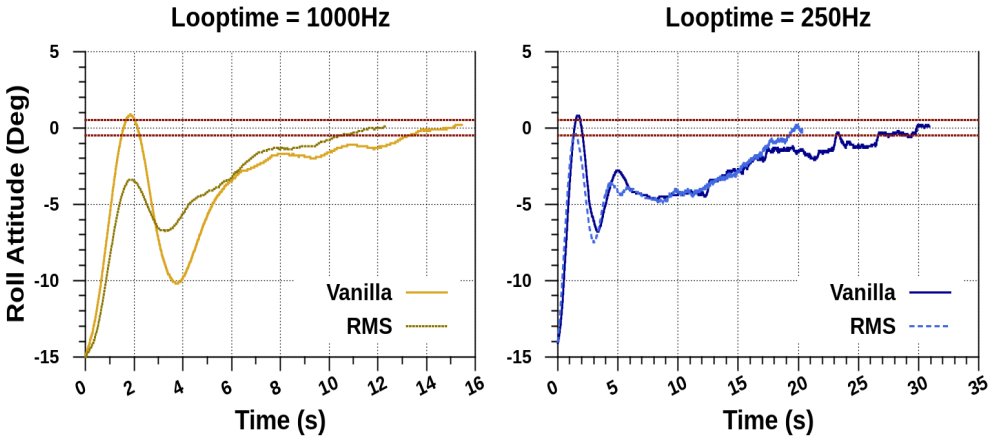
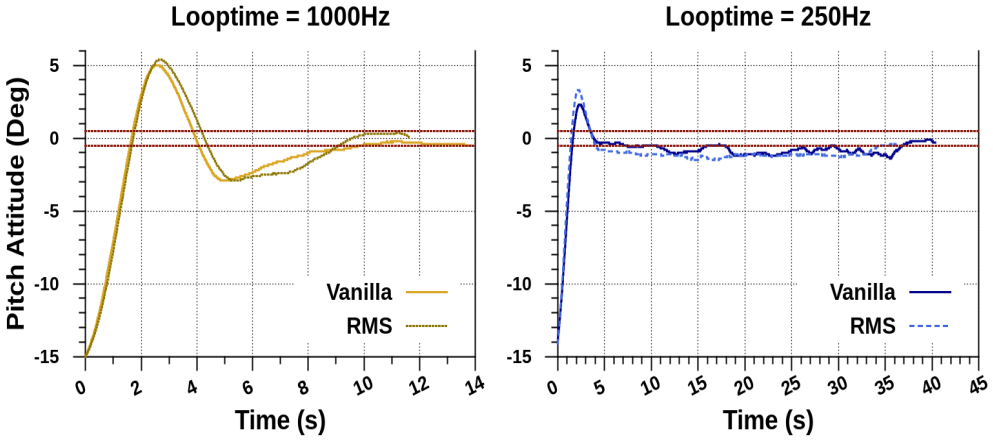


Figure 5 RMS CF: Response Times for **Roll Left** (*left*) and **Pitch Back** (*right*). A comparison between different looptimes shows best response times of 11s (9s) at looptime=1000Hz for Roll (Pitch).



(a) Exp: Roll-Left: RMS CF reduces response time by 18.5% (*left*) & 26.4% (*right*).



(b) Exp: Pitch-Back: RMS CF reduces response time by 10% (*left*) & 7.6% (*right*).

Figure 6 Comparison of attitude variation profiles for **RMS** & **Vanilla** CF.

Our results confirm that replacing the vanilla scheduler with a static priority real-time scheduling policy yields better response times. In contrast to Vanilla CF, interleaved task executions and enforcement of strict deadlines ensures timing predictability. This results in low-latency response to changes in the environment. Unlike Vanilla CF, lower priority tasks do not have to wait in the scheduler queue for assignment of a dynamic priority. Thus each task executes once every time period. The RMS scheduler dispatches tasks periodically, removing unnecessary dispatching delays and variations in scheduling latency, guaranteeing deterministic behavior for the entire system.

4.5 Phase III: Performance of smARTflight

We set smARTflight's task rates to correspond to looptime=250Hz and 1000Hz, for LO and HI mode, respectively. smARTflight provides statically configurable attitude thresholds about the target attitude (*target* ± *threshold*) in each axis of control: roll, pitch and yaw. This allows us to independently tune flight behavior in the corresponding axis, by trading responsiveness of the drone to changes in the environment (HI mode) against energy efficient utilization of system resources and better power consumption (LO mode).

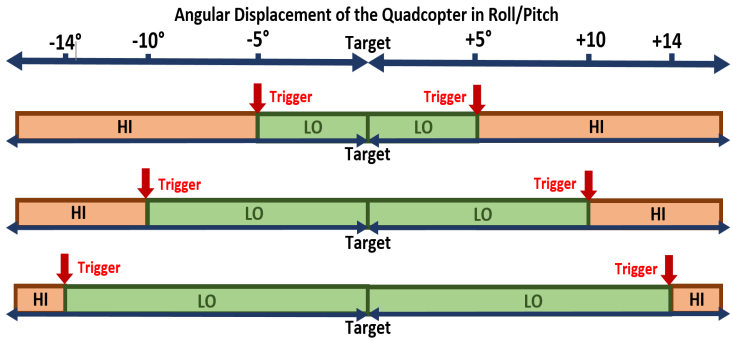


Figure 7 smARTflight: Attitude threshold displacements from the target i.e. target ± {5°,10°,14°} to trigger mode change in roll & pitch axes.

Fig. 7 gives a pictorial representation of LO→HI and HI→LO mode-switches at different angle thresholds, depending upon the drone's attitude displacement from the target. Three distinct values are considered: ±5°, ±10° and ±14° relative to our target of 0° with a maximum of 15° step attitude disturbance. The length of the horizontal bars in the diagram indicates the amount of time the flight controller system operates in one mode relative to the other. We study the influence of these thresholds on the response time performance of our copter, and compare results for both roll and pitch in Fig. 8.

The best response time performance of 9s in the roll axis, and 4s in the pitch axis is achieved at a threshold of ±**10**° and ±**14**°, respectively. The threshold difference between roll and pitch is purely an artifact of the mechanical structure of the drone, because similar flight control logic is used for both axes.

Legacy autopilots like Vanilla CF allow for mixer configuration and PID tuning to appropriately compensate for: (1) distribution of the overall mass along the two axes, and (2) the motors not being equidistant from one another. The final mix of outputs from the PID controller and the throttle commands (Fig. 2) controls the power to the motors. Thus, tuning either mixer or PID values directly affects the net applied thrust along an axis of rotation, which in-turn influences the drone's response time. To achieve optimal performance for each Cleanflight system, we therefore employ the legacy tuning method.

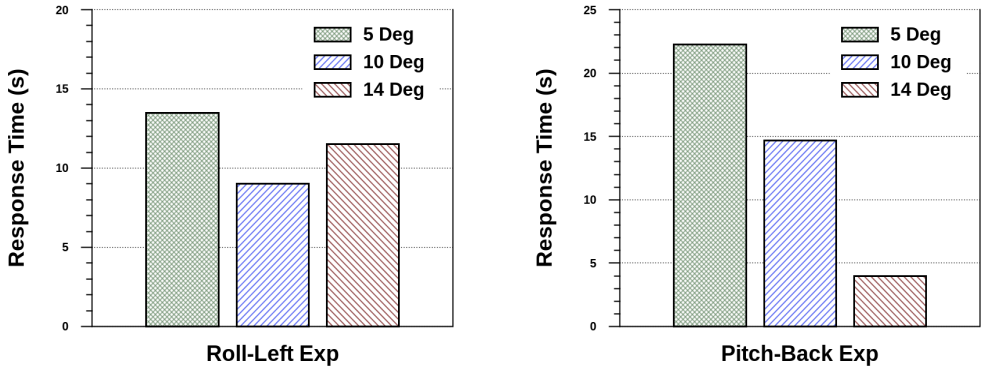
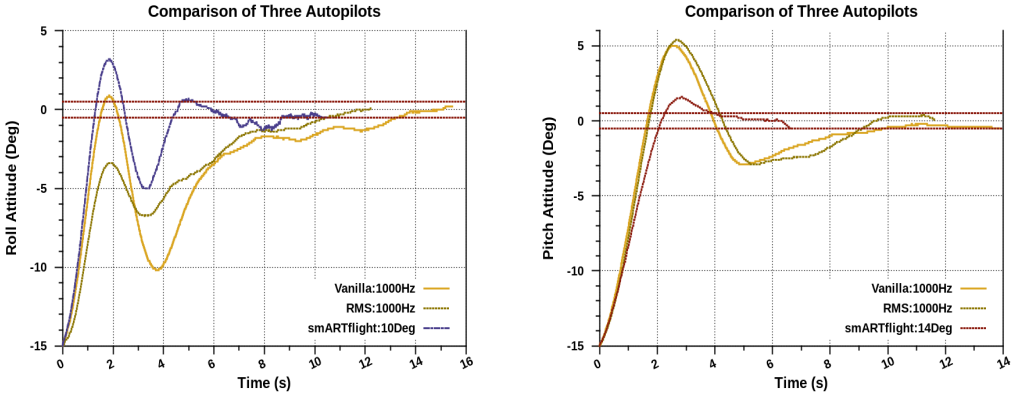


Figure 8 smARTflight: Response times for **Roll Left** (*left*) and **Pitch Back** (*right*). A comparison between 3 attitude thresholds: **5°, 10°,14°**. Best response time of 9s (4s) is achieved with roll (pitch) at 10° (14°).

However, results from Fig. 8 show that with smARTflight, "attitude-threshold" instead provides a *single* tuning knob for improving performance. Thresholds define the attitude boundary between LO and HI system modes, which adapt flight controller behavior. Threshold variations thus allows for performance control at system run-time. For all comparisons with smARTflight, we keep PID controller constants fixed at their optimal values across all autopilots, and used the same standard mixer settings from earlier experiments with Vanilla and RMS CF.



(a) Exp: Roll-Left: **smARTflight** (0°± **10°**) reduces response time by 33.3% (18.2%) compared to Vanilla & RMS CF.

(b) Exp: Pitch-Back: **smARTflight** (0° ± **14°**) reduces response time by 60% (55.6%) compared to Vanilla & RMS CF.

Figure 9 smARTflight versus **Vanilla** & **RMS CF** (looptime=1000Hz).

Figs. 9a–9b show average case attitude adjustments over time for all three autopilots. To compare smARTflight against the optimal performance achieved with Cleanflight systems, we set the looptimes for Vanilla and RMS CF to 1000Hz. We also tune smARTflight with threshold values of 10° in the roll and 14° in the pitch axis, to yield best case response time performance. With the right thresholds, smARTflight significantly improves the drone's response in recovering from an initial step disturbance. The improvements range from a minimum 33% to a maximum of 60% reduction in response time against Vanilla CF.

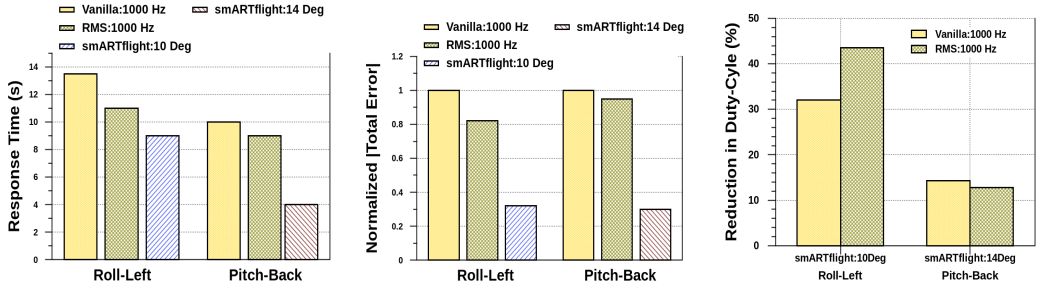


Figure 10 Response time to reach target hover attitude [Steady-State Response] (*left*); Cumulative attitude adjustment error from the step disturbance [Transient Response] (*center*); Percentage reduction in average duty-cycle for Motor-3 [$\propto$ Power Usage] (*right*).

Transient response characteristics of all flight controllers are represented as the cumulative absolute error. We normalize this error against that of Vanilla CF for a clear comparison. Results of our offline analysis are presented in Fig. 10. smARTflight's benefits clearly supersede both Cleanflight systems. By controlling task rates and switching system modes at appropriate times, smARTflight ensures minimum flight response times, and reduces the total absolute error by at-least 68% compared to Vanilla CF.

In addition, smARTflight allows direct control over task utilization. Once the system stabilizes below the attitude threshold, flight control tasks do not need to run as fast. According to our looptime specifications, LO mode execution rates reduce the motor update frequency by a factor of 4 when compared against HI mode. Since motors are prime energy consumers on the drone, a reduced motor update frequency results in less power usage.

Fig. 10 (*right*) shows the percentage reduction in average duty-cycle for motor-3 using smARTflight, compared to Vanilla and RMS CF, for pitch and roll axes. We present minimum, maximum and average duty-cycle for all three autopilots in Table 6. A particular point to note is that a higher frequency of real-time task executions within RMS CF comes at a cost of increased duty-cycle for the motors, compared to Vanilla CF. With real-time constraints, system idle time is reduced and updates to the motors happen periodically. This is in contrast to Vanilla CF where the system idles for longer periods and motors are not updated at a fixed rate (Table 3). We observe that smARTflight reduces power usage against both Cleanflight systems by consolidating real-time benefits with system mode-switches.

Table 6 Average PWM duty-cycle for Motor-3 (Bottom-Left Motor) across all autopilots.

Roll-Left Experiment			
Autopilot	**Motor Percentage Duty-Cycle**		
	Min	**Max**	**Average**
Vanilla:1000Hz	23.5%	38.3%	29.9%
RMS:1000Hz	30.7%	41.2%	35.9%
smARTflight:10°	20.2%	29.8%	20.3%
Pitch-Back Experiment			
Vanilla:1000Hz	16.5%	31.5%	22.3%
RMS:1000Hz	18.1%	27.1%	21.9%
smARTflight:14°	16%	24.6%	19.1%

5 Related Work

Bregu *et.al.*'s work on reactive control for aerial drones [10] adapts task execution rates according to environmental triggers. Unlike smARTflight, reactive control does not ensure task timing predictability, and only allows for adaptation of the average control rate associated with the fast-loop. smARTflight instead ensures deterministic flight control operation with independent rate adaptations for *all* critical flight control tasks within the loop. Furthermore, smARTflight provides well-defined system modes, time bounds and rate transitioning semantics. Unlike reactive control, smARTflight does not impose any limitations on the tuning parameters of the PID controller but provides an additional threshold tuning handle to enhance responsiveness of the system to external environmental triggers.

Mixed Criticality Systems (MCS) [30] have gained special attention over the past decade with widespread applicability in the automotive and avionic domains. Flight controllers for aerial drones present a prime example of the coexistence of multiple functions with varying degrees of importance on resource-constrained embedded platforms. smARTflight's criticality-aware adaptation model is influenced by research in the mixed-criticality domain [1].

Baruah *et.al.*'s work on Adaptive Mixed Criticality (AMC) scheduling [27] allows tasks to adapt their execution time budgets across system modes. AMC makes use of Audsley's priority assignment algorithm [20] as deadline monotonic priority assignment is shown to be sub-optimal for tasks with multiple execution time budgets [30]. By comparison, smARTflight adapts task execution rates rather than their budgets. This makes sense in the context of a flight management system, designed to operate in changeable environmental conditions. If environmental conditions affect the attitude of a drone at some changeable rate, then sensing and attitude control tasks must adjust their sampling and processing frequencies accordingly, if successful flight is to be achieved.

The original AMC model was later extended to also allow for changes in task rates [26] and task priorities [2, 28]. These works, however, rely on internal system triggers as opposed to external environmental factors. smARTflight exclusively relies on environmental dynamics to affect the system's state. Crespo [16] and Pedro [21] conducted a detailed response time analysis for mode changes in uni-processor systems. We derive smARTflight's unique mode transition protocol from a detailed study of the theoretical analysis presented in these works.

6 Conclusions & Future Work

This work presents smARTflight, a novel and principled timing predictable, rate-adaptive flight management system for multi-copter drones. smARTflight dynamically configures execution frequencies of sensor data processing and flight control tasks in response to external disturbances such as wind. The system extends existing flight controllers with criticality-aware real-time scheduling and enhanced environmental awareness, to improve overall flight performance and stability.

We define safety-critical task and system model semantics and identify conditions to trigger mode-switches between higher and lower criticality levels based on external factors. As a proof of concept, we identify critical and non-critical tasks within the popular Cleanflight flight controller and replace the traditional best-effort scheduling algorithm with smARTflight's adaptive rate-monotonic policy. Empirical comparisons with Vanilla Cleanflight show significant improvements in flight accuracy and stability with lower response time latency and better energy usage. Our study therefore validates smARTflight's capability to smartly manage available system resources, and quickly correct for transient attitude variations (e.g. due to wind disturbances) with lower power consumption.

Future work will investigate active power management strategies, to increase the flying range of multi-rotor drones. Our aim is to incorporate smARTflight into an autonomous flight management system for multicore flight controllers, using complementary OS components for reconfigurable missions. Advanced OS software components will empower smARTflight with driver and library support for 3D cameras and wireless communication devices, and object tracking and avoidance algorithms.

References

1. A. Burns and R. I. Davis. A Survey of Research into Mixed Criticality Systems. In *ACM Computing Surveys (CSUR)*, pages 1–37, January 2018.
2. A. Burns and S. K. Baruah. Towards a More Practical Model for Mixed Criticality Systems. In *In Proceedings of the Ist Workshop on Mixed Criticality Systems (WMC), RTSS*, pages 1–6, 2013.
3. Ardupilot. Home. URL: `goo.gl/x2CHyM`.
4. BBC News. Disaster Drones: How Robot Teams can Help in a Crisis. URL: `goo.gl/6efliV`.
5. Betaflight. Home. URL: `https://betaflight.com/`.
6. C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. In *Journal of the ACM (JACM)*, pages 46–61, 1973.
7. Cleanflight. Home. URL: `goo.gl/uCGmr4`.
8. D. Clifton. SPRACINGF3 Flight Controller Manual (Revision 4), 2015. URL: `https://bit.ly/2Mx9dRV`.
9. Da-Jiang Innovations Science and Technology Co. DJI. URL: `http://dji.com/`.
10. E. Bregu, N. Casamassima, D. Cantoni, L. Mottola, and K. Whitehouse. Reactive Control of Autonomous Drones. In *In Proceedings of the 14th Annual International Conference on Mobile Systems, Applications and Services (MobiSys'16)*, pages 207–219, June 2016.
11. E. Ibarra and P. Castillo. "Nonlinear super twisting algorithm for UAV attitude stabilization. In *In Proceddings of 2017 International Conference on Unmanned Aircraft Systems (ICUAS*, June 2017.
12. F. Corrigan. 12 Top Collision Avoidance Drones And Obstacle Detection Explained., January 2020. https://www.dronezon.com/learn-about-drones-quadcopters/top-drones-with-obstacle-detection-collision-avoidance-sensors-explained/.
13. G. Y. Immanuel and E. Johnson. State-based Scheduling of Real-Time UAV Flight Control Avionics Tasks. In *InfoTech at Aerospace: Advancing Contemporary Aerospace Technologies and Their Integration*, pages 945–951, 2005.
14. iNav. Home. URL: `https://github.com/iNavFlight/inav/wiki`.
15. Inc. Amazon.com. Amazon Prime Air. URL: `https://www.amazon.com/b?ie=UTF8{&}node=8037720011`.
16. J. Real and A. Crespo. Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal. In *Journal of Real-Time Systems*, volume 26, pages 161–197, 2004.
17. K. C. Peng, L. Feng, K. C. Peng, Y. C. Hseeh, T. H. Yang, S. H. Hsiung, Y. D. Tsai, and C. Kuo. Unmanned Aerial Vehicle for Infrastructure Inspection with Image Processing for Quantification of Measurement and Formation of Facade Map. In *In Proceedings of the 2017 IEEE International Conference on Applied System Innovation, IEEE-ICASI*, 2017.
18. K. P. Valavanis. *Advances in Unmanned Aerial Vehicles*. Springer Science and business Media, 2008.
19. K. Slowey. More Evidence that Drones Could and Should play Major Role in Infrastructure Inspections., February 2019. URL: `https://www.constructiondive.com/news/more-evidence-that-drones-could-and-should-play-major-role-in-infrastructur/547684/`.
20. N. C. Audsley. On Priority Assignment in Fixed Priority Scheduling. *Information Processing Letters*, 79(1):39–44, 2001.

21 P. Pedro and A. Burns. Schedulability Analysis for Mode Changes in Flexible Real-Time Systems. In *In 10th Euromicro Workshop on Real-Time Systems (ECRTS)*, pages 172–179, 1998.

22 PX4. Home. http://px4.io/.

23 R. Braun and S. Garlington. Drone Photography in Whale Research, December 2018. URL: `https://djiphotoacademy.com/drone-photography-for-whale-research/`.

24 S. Islam, M. Faraz, R. K. Ashour, G. Cai, J. Dias, and L. Seneviratne. Adaptive Sliding Mode Control Design for Quadrotor Unmanned Aerial Vehicle. In *International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 34–39, 2015.

25 S. Jordan, J. Moore, S. Hovet, J. Box, J. Perry, D. Lewis K. Kirsche, and Z. T. H. Tse. State-of-the-art Technologies for UAV Inspections. In *IET Radar, Sonar & navigation*, volume 12, pages 151–164, 2017.

26 S. K. Baruah. Schedulability Analysis of Mixed-Criticality Systems with Multiple Frequency Specifications. In *In Proceedings of the International Conference on Embedded Software (EMSOFT)*, 2016.

27 S. K. Baruah, A. Burns, and R. I. Davis. Response-time Analysis for Mixed Criticality Systems. In *In Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS)*, pages 34–43, 2011.

28 A. Burns S. K. Baruah and R. I. Davis. An Extended Fixed Priority Scheme for Mixed Criticality Systems. *Proc. ReTiMiCS, RTCSA*, pages 18–24, 2013.

29 S. O. H. Madgwick., A. J. L. Harrison, and R. Vaidyanathan. Estimation of IMU and MARG Orientation using a Gradient Descent Algorithm., 2011.

30 S. Vestal. Preemptive Scheduling of Multi-Criticality Systems with Varying Degrees of Execution Time Assurance. In *In Proceedings of the 28th IEEE Real-Time Systems Symposium*, pages 239–243, 2007.

31 W. Yang, M. Chun, G. Jang, J. Baek, and S. Kim. A study on Smart Drone using Quadcopter and Object Tracking Techniques. In *In Proceedings of IEEE 4th International Conference on Computer Applications and Information Processing Technology*, pages 1–5, March 2018.

32 X-IO Technologies. Open Source IMU and AHRS algorithms. URL: `https://x-io.co.uk/open-source-imu-and-ahrs-algorithms/`.