29th Euromicro Conference on Real-Time Systems

ECRTS 2017, June 28–30, 2017, Dubrovnik, Croatia

Edited by

Marko Bertogna

Editor

Marko Bertogna
University of Modena
Modena
Italy
marko.bertogna@unimore.it

ACM Classification 1998
C.3 Real-Time and Embedded Systems, C.4 Performance of Systems, D.3.4 Processors, D.4.1 Scheduling, D.4.7 Real-Time Systems and Embedded Systems

ISBN 978-3-95977-037-8

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-95977-037-8.

Publication date
June 2017

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available online at http://dnb.d-nb.de.

Digital Object Identifier: 10.4230/LIPIcs.ECRTS.2017.0

LIPIcs – Leibniz International Proceedings in Informatics

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

ISSN 1868-8969

http://www.dagstuhl.de/lipics

Contents

29th Euromicro Conference on Real-Time Systems (ECRTS 2017).
Editor: Marko Bertogna
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Session 4: Automotive Systems

Session 5: Multi-Core Scheduling

Session 6: Probabilistic and Weakly-Hard Models

Session 7: Mixed Criticality

Session 8: Energy- and Security-Aware Scheduling

Session 9: Outstanding Papers

Preface

Message from the Chairs

It is our pleasure to welcome you to the **29th Euromicro Conference on Real-Time Systems** (ECRTS 2017) in Dubrovnik, Croatia. ECRTS is the premier European venue for presenting research in the broad area of real-time and embedded systems. Along with RTSS and RTAS, ECRTS ranks as one of the three top international conferences on real-time systems.

We received **79 submissions** this year with authors from 22 countries, 36 (46%) from outside Europe. Each paper was reviewed by at least three active researchers in our community. Then, the program committee met in person and discussed each paper. From that discussion, 26 high-quality submissions were selected for publication and a 30 minute presentation. Three of these papers have been recognized as **Outstanding Papers** and will be presented in a dedicated session. Among the three, one is going to be selected as the Best Paper, based on the scientific contribution and the presentation clarity. Authors of accepted papers had the possibility to submit a replication package. An artifact evaluation committee validated the artifacts and included a seal of approval for those who passed the replication test. Six of the papers in the proceedings are marked with this seal.

In addition to full-length paper sessions, a **Work-In-Progress session** for short papers has been organized. Papers submitted to the Work-In-Progress session were evaluated separately by a second committee, chaired by **Patrick Meumeu Yomsi**, and are not part of these published proceedings.

The first conference days feature an opening plenary session with industrial keynote speakers. The keynotes are a great occation to identify important, unsolved, challenges faced by real-time systems practitioners. The first keynote talk will be given by **Giulio Corradi**, from Xilinx, presenting architectural insights to address the predictability and safety challenges brought by next-generation system on chip and hardware programmable heterogeneous architectures. The second will be given by **Peter Zijlstra**, from Intel, outlining the current state of Real-Time scheduling supported by the Linux kernel, and highlighting current short comings and proposed ways of addressing them.

ECRTS 2017 is the first real-time conference introducing an **Open Access** publication model in collaboration with LIPIcs – Leibniz International Proceedings in Informatics established in cooperation with Schloss Dagstuhl, Leibniz Center for Informatics. Paper selection procedures and quality control remained unchanged, but from this year on accepted papers are made available to the public without cost.

The day before the main conference is dedicated to five outstanding international workshops: the workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), the workshop on Real-Time Networks (RTN), the Real-Time Scheduling Open Problems Seminar (RTSOPS), the workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS), and the workshop on Worst-Case Execution Time Analysis (WCET).

ECRTS 2017 was made possible by the hard work of many people. We are especially grateful for the contributions of the following people: the **Program Committee** and reviewers, who are listed in subsequent pages; Patrick Meumeu Yomsi as Work-In-Progress

29th Euromicro Conference on Real-Time Systems (ECRTS 2017).
Editor: Marko Bertogna
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Chair; Marcus Völp and Heechul Yun, as **OSPERT** Workshop Chairs; Jean-Luc Scharbarg and Mathieu Jan as **RTN** Workshop Chairs; Vincent Nelis and Thidapat (Tam) Chantem as **RTSOPS** Workshop Chairs; Sophie Quinton and Arne Hamann as **WATERS** Workshop Chairs; Jan Reineke as **WCET** Workshop Chair; Sebastian Altmeyer and Martina Maggio as Artifact Evaluation co-Chairs. A special thanks to Marc Herbstritt of Dagstuhl Publishing and Björn Brandenburg as Publication Chair, who did an outstanding job in collecting and properly formatting the camera-ready versions of the papers appearing in these proceedings, and to Gerhard Fohler for his unwavering support and contributions as the Euromicro Real-Time Technical Committee Chair.

Congratulations to all of the authors for their exceptional work. ECRTS 2017 would not exist without the contributions of the authors that submitted their work for review and critique. We are very pleased with the quality, depth, and breadth of this year's technical program. We hope you enjoy yourself at ECRTS 2017!

Martina Maggio
General Chair, ECRTS 2017

Marko Bertogna
Program Chair, ECRTS 2017

Committees

General Chair

Martina Maggio, Lund University, Sweden

Program Chair

Marko Bertogna, University of Modena, Italy

Real-Time Technical Committee Chair

Gerhard Fohler, TU Kaiserslautern, Germany

Publications Chair

Björn B. Brandenburg, Max Planck Institute for Software Systems (MPI-SWS), Germany

Artifact Evaluation Chairs

Martina Maggio, Lund University, Sweden
Sebastian Altmeyer, University of Amsterdam, Netherlands

Program Committee

Sebastian Altmeyer, University of Amsterdam, Netherlands
Benny Åkesson, TNO-ESI, Netherlands
Karl-Erik Årzén, Univerity Lund, Sweden
Patricia Balbastre, Universitat Politècnica de València, Spain
Sanjoy Baruah, University of North Carolina at Chapel Hill, USA
Andrea Bastoni, SYSGO AG, Germany
Robert I. Davis, University of York, UK & INRIA-Paris, France
Jean-Dominique Decotignie, EPFL/CSEM, Switzerland
Marco Di Natale, Scuola Superiore S. Anna, Italy
Johan Eker, Ericsson Research, Sweden
Rolf Ernst, TU Braunschweig, Germany
Gerhard Fohler, TU Kaiserslautern, Germany
Christian Fraboul, IRIT - ENSEEIHT, Toulouse, France
Steve Goddard, University of Nebraska-Lincoln, USA
Nan Guan, Hong Kong Polytechnic University, HK SAR, China
Arne Hamann, Robert Bosch Gmbh, Germany
Robert Kaiser, Hochschule RheinMain, Germany
Shinpei Kato, University Nagoya, Japan
George Lima, Federal University of Bahia, Brazil
Daniel Lohmann, FAU Erlangen-Nürnberg, Germany
Martina Maggio, Lund University, Sweden
Claire Maiza, INP/Verimag, Grenoble, France

29th Euromicro Conference on Real-Time Systems (ECRTS 2017).
Editor: Marko Bertogna
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Julio Luis Medina, University of Cantabria, Spain
Vincent Nelis, CISTER, ISEP, Portugal
Geoffrey Nelissen, CISTER, ISEP, Portugal
Claire Pagetti, ONERA, France
Michael Paulitsch, Thales, Austria
Rodolfo Pellizzoni, University of Waterloo, Canada
Isabelle Puaut, University of Rennes I / IRISA, France
Peter Puschner, Vienna University of Technology, Austria
Sophie Quinton, INRIA-Grenoble Rhône-Alpes, France
Jan Reineke, Universität des Saarlandes, Germany
Christine Rochange, IRIT, University of Toulouse, France
Marcus Völp, University of Luxembourg, Luxembourg

Additional Reviewers

Leonie Ahrendts
Matthias Beckert
Frédéric Boniol
Alan Burns
Thomas Carle
José Carlos Palencia
Hugues Cassé
Bekim Cilku
Jacques Combaz
Javier Coronel
Alfons Crespo
Dakshina Dasari
Christian Dietrich
Leonardo Ecco
Christian Eichler
Nathan Fisher
Tom Fleming
Pascal Fradet
Florian Franzmann
Bernhard Froemel
Gautam Gala
David García Villaescusa
Michael González Harbour
Sebastian Hahn
Zain Hammadeh
Florian Heilmann
Robin Hofmann
Haris Isakovic
Tobias Klaus
Reinhold Kroeger
Kristin Krüger
Patrick Meumeu Yomsi
Mischa Möstl
Matthieu Moy
Rainer Müller
Mitra Nasri
Héctor Pérez Tijero
Quentin Perret
Eberle Rambo
Ralf Ramsauer
Denise Ratasich
Benjamin Rouxel
Selma Saidi
Luca Santinelli
Johannes Schlatow
Martin Schoeberl
Muhammad Soliman
Ali Syed
Marcus Thoss
Peter Ulbrich
Peter Waegemann
Saud Wasly
Wei Zhang
Alexander Zuepke

Artifact Evaluators

Marcus Thelander Andrén
Antoine Bertout
Hugo Daigmorte
Zain Hammadeh
Tobias Klaus
Sandro Pinto
Eberle Rambo
Hamza Rihani
Gautham Nayak Seetanadi
Youcheng Sun
Yang Xu
Shengqian Yang
Yuanbin Zhou

List of Authors

Jakaria Abdullah
Uppsala University, Sweden
jakaria.abdullah@it.uu.se

Jaume Abella
Barcelona Supercomputing Center (BSC-CNS), Spain
jaume.abella@bsc.es

Ankit Agrawal
Technische Universität Kaiserslautern, Germany
agrawal@eit.uni-kl.de

Benny Akesson
TNO Eindhoven, Netherlands
benny.akesson@tno.nl

Gatherer Alan
America Wireless Access Laboratory, Huawei Technologies Co. Ltd, USA
alan.gatherer@huawei.com

Jim Anderson
The University of North Carolina at Chapel Hill, USA
anderson@cs.unc.edu

Jan Andersson
Cobham Gaisler, Sweden
jan@gaisler.com

Neil Audsley
University of York, United Kingdom
neil.audsley@york.ac.uk

Muhammad Ali Awan
CISTER Research Centre, Portugal
muaan@isep.ipp.pt

Alen Bardizbanyan
Cobham Gaisler, Sweden
alen.bardizbanyan@gaisler.com

Sanjoy Baruah
The University of North Carolina at Chapel Hill, USA
baruah@cs.unc.edu

Ashikahmed Bhuiyan
Missouri University of Science and Technology, USA
abvn2@mst.edu

Alessandro Biondi
Scuola Superiore Sant'Anna, Italy
alessandro.biondi@sssup.it

Tobias Blaß
Saarland University, Germany
s9toblas@stud.uni-saarland.de

Konstantinos Bletsas
CISTER/INESC-TEC, ISEP, Portugal
ksbs@isep.ipp.pt

Rakesh Bobba
Oregon State University, USA
rakesh.bobba@oregonstate.edu

Giorgio Buttazzo
Scuola Superiore Sant'Anna, Italy
giorgio@sssup.it

Jorge Cabral
University of Minho
jcabral@dei.uminho.pt, Portugal

Marco Caccamo
University of Illinois at Urbana-Champaign, USA
mcaccamo@cs.uiuc.edu

Daniel Casini
Scuola Superiore Sant'Anna, Italy
d.casini@sssup.it

Francisco Cazorla
Barcelona Supercomputing Center and IIIA-CSIC, Spain
francisco.cazorla@bsc.es

Kevin Chappuis
Virtual Open Systems, France
k.chappuis@virtualopensystems.com

Jian-Jia Chen
TU Dortmund, Germany
jian-jia.chen@cs.uni-dortmund.de

29th Euromicro Conference on Real-Time Systems (ECRTS 2017).
Editor: Marko Bertogna
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Fabrice Cros
Airbus Defense and Space, France
fabrice.cros@airbus.com

Unmesh D. Bordoloi
General Motors, Germany
unmesh.bordoloi@gmail.com

Nicolas Dagieu
Virtual Open Systems, France
n.dagieu@virtualopensystems.com

Dakshina Dasari
Robert Bosch GmbH, Germany
dakshina.dasari@de.bosch.com

Robert Davis
University of York, United Kingdom
rob.davis@york.ac.uk

Zheng Dong
University of Texas at Dallas, USA
zheng@utdallas.edu

Sandeep D'souza
Carnegie Mellon University, USA
sandeepd@andrew.cmu.edu

Pontus Ekberg
Uppsala University, Sweden
pontus.ekberg@it.uu.se

Rolf Ernst
TU Braunschweig, Germany
r.ernst@tu-bs.de

Heiko Falk
Hamburg University of Technology, Germany
heiko.falk@tuhh.de

Gerhard Fohler
Technische Universität Kaiserlautern, Germany
fohler@eit.uni-kl.de

Eugen Frasch
Ulm University, Germany
eugen.frasch@uni-ulm.de

Johannes Freitag
Airbus Innovations, Germany
johannes.freitag@airbus.com

Tiago Gomes
University of Minho, Portugal
tgomes@dei.uminho.pt

Nan Guan
Hong Kong Polytechnic University, Hong Kong, Special Administrative Region of China
csguannan@comp.polyu.edu.hk

Zhishan Guo
Missouri University of Science and Technology, USA
guozh@mst.edu

Zain A. H. Hammadeh
TU Braunschweig, Germany
hammadeh@ida.ing.tu-bs.de

Sebastian Hahn
Saarland University, Germany
sebastian.hahn@cs.uni-saarland.de

Arne Hamann
Robert Bosch GmbH, Germany
arne.hamann@de.bosch.com

Damien Hardy
University of Rennes 1/IRISA, France
dhardy@irisa.fr

Monowar Hasan
University of Illinois at Urbana-Champaign, USA
mhasan11@illinois.edu

Franz J. Hauck
Ulm University, Germany
franz.hauck@uni-ulm.de

Rafik Henia
Thales Research & Technology, France
rafik.henia@thalesgroup.com

Carles Hernandez
Barcelona Supercomputing Center, Spain
carles.hernandez@bsc.es

Wen-Hung Huang
TU Dortmund, Germany
wen-hung.huang@tu-dortmund.de

Zhe Jiang
University of York, United Kingdom
zj577@york.ac.uk

Prachi Joshi
Virginia Tech, USA
prachi@vt.edu

Simon Kramer
Robert Bosch GmbH, Germany
simon.kramer2@de.bosch.com

Cong Liu
University of Texas at Dallas, USA
cxl137330@utdallas.edu

Pierre Lucas
Virtual Open Systems, France
p.lucas@virtualopensystems.com

Arno Luppold
Hamburg University of Technology, Germany
arno.luppold@tuhh.de

Renato Mancuso
University of Illinois at Urbana-Champaign, USA
rmancus2@illinois.edu

Lee McFearin
America Wireless Access Laboratory, Huawei Technologies Co. Ltd, USA
lee.mcfearin@huawei.com

Sibin Mohan
University of Illinois at Urbana-Champaign, USA
sibin@illinois.edu

Morteza Mohaqeqi
Uppsala University, Sweden
morteza.mohaqeqi@it.uu.se

Frank Mueller
North Carolina State University, USA
mueller@cs.ncsu.edu

Viet Anh Nguyen
University of Rennes 1 / IRISA, France
anh.nguyen@irisa.fr

Vladimir Nikolov
Ulm University, Germany
vladimir.nikolov@uni-ulm.de

Jan Nowotsch
Airbus Innovations, Germany
jan.nowotsch@gmail.com

Dominic Oehlert
Hamburg University of Technology, Germany
dominic.oehlert@tuhh.de

Marco Panunzio
Thales Alenia Space, France
marco.panunzio.ext@gmail.com

Michele Paolino
Virtual Open Systems, France
m.paolino@virtualopensystems.com

Risat Mahmud Pathan
Chalmer University of Technology, Sweden
risat@chalmers.se

Michael Paulitsch
Thales Vienna, Austria
michael.paulitsch@thalesgroup.com

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

Jorge Pereira
University of Minho, Portugal
jorge.m.pereira@algoritmi.uminho.pt

Sandro Pinto
University of Minho, Portugal
sandro.pinto@dei.uminho.pt

Michael Pressler
Robert Bosch GmbH, Germany
michael.pressler@de.bosch.com

Isabelle Puaut
University of Rennes 1 / IRISA, France
puaut@irisa.fr

Tao Qian
North Carolina State University, USA
tqian2@ncsu.edu

Sophie Quinton
Inria, France
sophie.quinton@inria.fr

Daniel Raho
Virtual Open Systems
s.raho@virtualopensystems.com

Raj Rajkumar
Carnegie Mellon, USA
raj@ece.cmu.edu

Eberle A. Rambo
TU Braunschweig, Germany
rambo@ida.ing.tu-bs.de

S. S. Ravi
Virginia Tech, USA
ssravi@vbi.vt.edu

Jan Reineke
Saarland University, Germany
reineke@cs.uni-saarland.de

Laurent Rioux
Thales Research & Technology, France
laurent.rioux@thalesgroup.com

Abusayeed Saifullah
Missouri University of Science and Technology, USA
saifullah@wayne.edu

Soheil Samii
General Motors Research & Development, USA
soheil.samii@gm.com

Sandeep Shukla
IIT Kanpur, India
sandeeps@cse.iitk.ac.in

Abhishek Singh
The University of North Carolina at Chapel Hill, USA
abh@cs.unc.edu

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

Pedro Souto
University of Porto, Portugal
pfs@fe.up.pt

Adriano Tavares
University of Minho, Portugal
atavares@dei.uminho.pt

Neriman Tokcan
University of Illinois at Urbana-Champaign, USA
tokcan2@illinois.edu

Eduardo Tovar
CISTER/INESC-TEC, Portugal
emt@isep.ipp.pt

Sascha Uhrig
Airbus Innovations, Germany
sascha.uhrig@airbus.com

Georg von der Brüggen
TU Dortmund University, Germany
georg.von-der-brueggen@tu-dortmund.de

Franck Wartel
Airbus Defense and Space, France
franck.wartel@airbus.com

Stefan Wesner
Ulm University
stefan.wesner@uni-ulm.de

Falk Wurst
Robert Bosch GmbH, Germany
falk.wurst@de.bosch.com

Yufeng Xin
RENCI, The University of North Carolina at Chapel Hill, USA
yxin@renci.org

Haoyi Xiong
Missouri University of Science and Technology
xiongha@mst.edu

Peter Yan
America Wireless Access Laboratory, Huawei Technologies Co.Ltd, USA
peter.yifey.yan@huawei.com

Wang Yi
Uppsala University, Sweden
yi@it.uu.se

Haibo Zeng
Virginia Tech, USA
haibo.zeng@gmail.com

Bus-Aware Static Instruction SPM Allocation for Multicore Hard Real-Time Systems*

Dominic Oehlert[1], Arno Luppold[2], and Heiko Falk[3]

1 **Hamburg University of Technology, Hamburg, Germany**
`dominic.oehlert@tuhh.de`
2 **Hamburg University of Technology, Hamburg, Germany**
`arno.luppold@tuhh.de`
3 **Hamburg University of Technology, Hamburg, Germany**
`heiko.falk@tuhh.de`

Abstract

Over the past years, multicore systems emerged into the domain of hard real-time systems. These systems introduce common buses and shared memories which heavily influence the timing behavior. We show that existing WCET optimizations may lead to suboptimal results when applied to multicore setups. Additionally we provide both a genetic and a precise Integer Linear Programming (ILP)-based static instruction scratchpad memory allocation optimization which are capable of exploiting multicore properties, resulting in a WCET reduction of 26% in average compared with a bus-unaware optimization. Furthermore, we show that our ILP-based optimization's average runtime is distinctively lower in comparison to the genetic approach. Although limiting the number of tasks per core to one and partially exploiting private instruction SPMs, we cover the most crucial elements of a multicore setup: the interconnection and shared resources.

1998 ACM Subject Classification C.3 Special-Purpose and Application-Based Systems, D.3.4 Processors, G.1.6 Optimization

Keywords and phrases compiler, optimization, WCET, real-time, multicore

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.1

1 Introduction

The Worst-Case Execution Time (WCET) of a program is defined as the worst possible time the program needs from the start until the end of its execution. In hard real-time systems where a task must provably finish its execution within a given amount of time, reducing the WCET is crucial to the correct behavior of the system.

Within the last couple of years, increasing computational requirements led to the introduction of multicore systems into the world of hard real-time systems. The drawback of these systems is the much more complex timing analysis due to shared memories which are accessed over common bus systems. Due to these common buses, the WCET of a program is heavily influenced by the implemented bus scheduling policy. These effects have to be taken into account during the analysis in order to determine a safe, yet tight WCET. Recent scientific works [3, 11] tackle the precise analysis of these systems, but optimization techniques have yet to catch up to these new challenges.

* This work received funding from Deutsche Forschungsgemeinschaft (DFG) under grant FA 1017/1-2. This work was partially supported by COST Action IC1202: Timing Analysis On Code-Level (TACLe).

29th Euromicro Conference on Real-Time Systems (ECRTS 2017).
Editor: Marko Bertogna; Article No. 1; pp. 1:1–1:22

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

Over the last years, memory optimizations, especially those featuring a fast but small Scratchpad Memory (SPM) have proven to be powerful tools to selectively optimize the WCET of a program for singlecore systems [5, 18]. We therefore exemplary use a state of the art ILP-based static WCET-aware instruction SPM allocation for singlecore systems to show that those optimizations may yield suboptimal results when applied to multicore systems without taking common memory buses into account. This even holds for relatively simple bus access algorithms like TDMA with equally-sized fixed slot lengths.

To counter these issues, we extend the ILP model by a precise bus model for a TDMA schedule with fixed slot lengths and show to be able to specifically optimize programs for multicore systems. Our experiments show WCET reductions of up to 90% percent over the bus-unaware SPM allocation. The presented approach covers precisely the potential blocking times of an access to a shared memory due to the TDMA schedule. As reference for the assessment of the quality of the ILP-based model, we additionally propose a bus-aware SPM allocation based on a genetic algorithm.

This paper is outlined as follows: Section 2 gives an overview over the related work. In Section 3, we give an overview of the used multicore architecture. Section 4 introduces a motivating example, illustrating the necessity to consider bus-related effects during WCET-driven optimizations for a multicore platform. In Section 5, we present the used base ILP model, the bus-aware extensions, as well as an overview of the nomenclature and certain preliminaries. Section 6 presents the evolutionary algorithm used for an instruction SPM allocation on a multicore platform. The bus-aware extensions to the ILP model and the evolutionary-based approach form the contributions of this paper. The evaluation of the presented approaches is shown in Section 7. Section 8 concludes this paper and gives an outlook on possible future work.

2 Related Work

Regarding hard real-time multicore systems, numerous parameters can be taken into account in order to increase its performance. Kelter et al. [10] presented WCET-aware scheduling optimizations for multicore systems. They showed a WCET-aware optimization centered on the schedule parameters of a system, e.g., the bus scheduling policy, number of bus slots or priorities of slots. Besides, they also presented an optimization featuring a WCET-oriented instruction reordering. For both approaches, evolutionary algorithms were exploited.

Suhendra and Mitra [17] presented the effects of locking and partitioning caches inside a multicore architecture, regarding the worst-case performance. They examined the timing profits of locking or partitioning a shared L2 instruction cache, based on task or core level. Both, dynamic and static cache locking, were discussed.

The optimization of programs in hard real-time systems using scratchpad memory allocation has been discussed in several publications. An ILP-based optimization for a WCET-aware data scratchpad memory (SPM) allocation for a singlecore architecture was presented by Suhendra et al. [18]. Based upon this structure, an adapted version for instruction memory allocation was introduced by Falk and Kleinsorge [5].

Liu and Zhang [15] presented different multicore architectures featuring multilevel scratchpad memories. They also demonstrated an ILP-based optimization to decrease the WCET of a program by allocating certain parts of a program to the different SPMs available. Static and dynamic SPM allocations were discussed, while also an evaluation concerning the worst-case energy consumption was given. However, bus- or multicore-related factors like

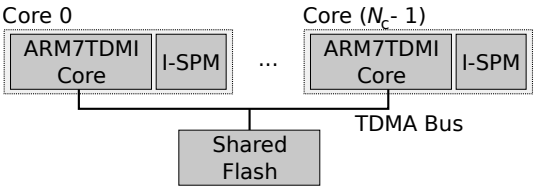


Figure 1 Overview of the proposed multicore architecture.

bus communication latencies are neglected in the presented timing models, thus lowering the accuracy of the overall model.

Kim et al. [13] presented a WCET-aware approach for dynamic code management on SPMs that focused on software-managed multicores. They proposed a multicore architecture with private SPMs, in which every main memory access is forced to go through the SPM, which issues a direct memory access (DMA). Based on this system, an ILP-based and a heuristic technique to reduce the WCET of a program were presented. Also here, the interconnection network between the SPMs and the main memory is neglected in terms of timing, thus degrading the accuracy of the presented model.

Kafshdooz and Ejlali [9] presented an ILP- and heuristic-based approach in order to reduce the WCET of a program, exploiting dynamic SPM allocation in a multicore-multiprocess system. Each bus access was assumed to take the worst latency possible, leading to a heavy over-approximation.

Chattopadhyay and Roychoudhury [2] introduced a static scratchpad allocation on a multicore system, that features bus-awareness. They presented a heuristic approach to reduce the worst-case response time (WCRT) of a multiprocessor program, based on a bus-aware WCRT-analysis. This analysis result was iteratively used to find a proper SPM allocation.

In contrast to the discussed approaches, we present an ILP-model which features a precise bus-awareness without the requirement to rely on the worst-case timing for each access. Besides, we demonstrate an evolutionary-based approach in order to classify the figure of merit of the presented model.

3 Multicore Architecture

This section presents the architecture used throughout this paper which is illustrated in Figure 1. The system consists of N_c parallel homogeneous cores with private instruction SPMs, a TDMA scheduled bus and a shared Flash memory which is connected to the bus. An SPM typically consists of a static RAM which is placed closely to the processor, leading to significantly lower access time in comparison to, e.g., Flash memories, yet limiting their overall capacity. We assume one task per core. Due to the homogeneity of the cores and the TDMA scheduled bus, the mapping of tasks to a core is not covered in this paper. The ARM7TDMI core was used for evaluation purposes only. The presented ILP-model is, however, generally applicable to other multicore architectures based on in-order processors. To improve the predictability, all caches of the cores are disabled. The instruction SPM of each core is private and can only be accessed via the attached core. Hence, no bus access is necessary during a scratchpad memory access. The shared Flash memory has to be accessed via the bus. The access delay of the Flash memory (excluding possible stalls before a bus grant) is considerably higher (approx. factor 6) in comparison to the access delay of an SPM. The bus is assumed to be TDMA scheduled, while the TDMA schedule consists of N_c slots. Each core's time slot length, during which it exclusively can access the bus, can be adjusted

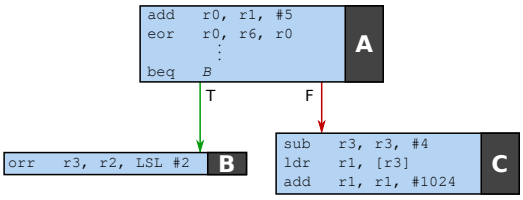


Figure 2 Exemplary CFG I.

Table 1 WCET (in cycles) for the exemplary program.

Basic Block	Flash	SPM
A	390	20
B	96	1
C	114	9

individually. It is assumed that all cores are globally synchronous. The execution of each core's task starts at a common point in time, which is assumed to be the first slot in the TDMA schedule.

The ARM7TDMI architecture features a 3 staged pipeline, fetches each instruction piecewise and supports a very basic form of branch „prediction“ (always not taken, even if unconditional).

4 Motivating Example

In order to demonstrate the necessity to consider the bus-related effects while performing an SPM allocation, we are assuming a small exemplary program, represented by the control flow graph (CFG) shown in Figure 2.

Basic block (BB) A is not shown fully here due to space limitations and consists of 20 instructions in total. The second instruction (`ldr`) of basic block C is assumed to access the `.data` section which is placed inside the shared Flash memory. As an exemplary system, we assume the architecture presented in Section 3 with 4 parallel cores and a TDMA bus with equally-sized slot lengths (each slot can accommodate 5 accesses to the shared Flash memory). One Flash memory access is assumed to take 6 cycles. Furthermore, we assume that each private SPM has a size of 20 Bytes. We assume the program to be executed on core 0, which owns the first bus slot of every bus period inside the schedule.

Following the ILP-based static instruction SPM allocation for a singlecore system [5], the WCET of the program is analyzed twice, once with all BBs placed inside the shared Flash memory and once with all BBs placed inside the (virtually enlarged) private SPM of core 0. These analyses return the WCET per BB denoted in Table 1. The BBs allocated to the SPM are loaded into the SPM once prior to the actual execution of the program. Therefore, the time required to initially load the BBs into the SPM can be neglected.

Since only basic blocks A and C are part of the worst case execution path (WCEP), but the SPM has a limitation of only 20 Bytes (5 instructions), the ILP solver will decide to place BB C into the SPM, resulting in an expected WCET reduction of 84 cycles. This reduction already includes jump correction costs of 21 cycles. These additional cycles stem from jump correction code, which has to be inserted subsequent to the SPM allocation decisions in order to restore a working control flow. Since BB C is placed in a different memory region, it is not subsequent (by means of physical addresses) to BB A anymore. Hence, an explicit jump (most likely an indirect one to cope with the physical address offsets of the memory regions) to C is required to be inserted at the end of BB A. However, a final WCET analysis with the proposed SPM allocation done reveals a WCET of 511 cycles, resulting in an actually worse timing after the optimization than before (504 cycles).

The cause of this misprediction is based on the bus-related latencies. Figure 3a shows the WCET of the program in regard to the bus schedule in case BB A and C are allocated to the shared Flash memory. The individual bus slots are shown along the x-axis, with one bus

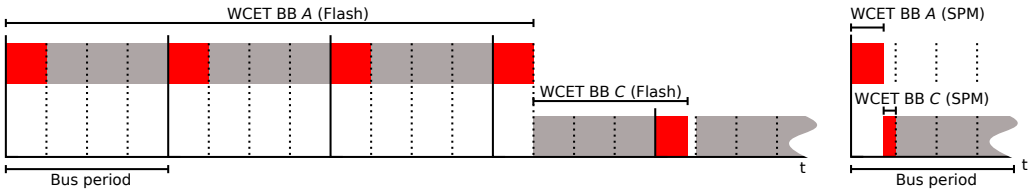


Figure 3 WCETs of BB A and C in regard to the bus schedule: (a) allocated to Flash, (b) allocated to SPM.

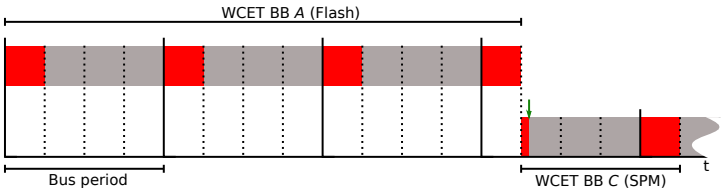


Figure 4 WCETs of BB A and C in regard to the bus schedule, „optimized“ Allocation.

period consisting of 4 bus slots. Core 0 (on which the program is allocated) owns the first bus slot of every period. The areas marked in red show the actual execution of instructions. Basic block A can be executed in exactly four slots, while BB C starts at the beginning of the second slot of a bus period, hence the processor has to stall until another fetch can be done. When the bus grant is regained, BB C can be executed within one bus slot.

Figure 3b shows the WCET for the second step, when all BBs have been allocated into the private SPM. Due to the severely lower access times to the SPM, basic blocks A and B can now be completely executed in the first bus slot.

Based upon these timings, the ILP solver assigns BB C to the SPM. The actual WCET analysis of the allocated program in regard to the bus schedule can be seen in Figure 4. Basic block A is executed in the same manner as seen in Figure 3a.

However, in difference to Figure 3b, the execution of basic block C now starts at the beginning of the second bus slot. This is caused by the fact that the execution time of the preceding basic block A is different to the timing analyzed when the whole program was placed inside the SPM. The execution of BB C can be started, since the instructions are placed inside the private SPM, but has to be stalled during the second instruction (`ldr`, depicted with an arrow) until the `.data` section can be accessed. Additionally, jump correction code has to be inserted to create a memory region crossing jump from basic block A to C. These circumstances lead to a drastically higher WCET for basic block C than expected, which again leads to a higher WCET in total.

This example shows the crucial sensitivity of a program's timing in regard to the underlying interconnection network. Neglecting bus-related timings strongly decreases the accuracy of the SPM allocation optimization, thus easily resulting in an underestimated WCET. We see that even though the presented optimization methods work fine for singlecore platforms, it does not give an accurate result when being applied to a multicore platform. This is due to the fact that the ILP model does not have a required notion of history to consider timing effects which are induced by preceding allocation decisions.

5 ILP Model

In the following, we will present an ILP model which is able to predict the timing behavior of the used bus architecture, thus enabling a WCET-centered instruction SPM allocation on multicore platforms, avoiding mispredictions as shown in the previous section. First, we will give a short overview of the notational conventions and ILP formulations used throughout this paper. Subsequently, we will shortly introduce the base ILP model which our work builds upon. Eventually, we are presenting our bus-aware extensions, which are enabling a precise prediction of bus-related timing behaviors.

5.1 Notational Conventions

In the following, lower case italic Latin letters like i will be used for ILP variables. Upper case italic Latin letters like A represent constants inside the ILP model. Letters in bold, e.g., $\boldsymbol{o}$, depict intervals of bus offsets. Lastly, lower case italic Greek letters like ν are used to denote a certain basic block.

Table 2 contains all ILP variables used in this paper, while Table 3 contains other miscellaneous symbols and their description used in this paper.

5.2 Mathematical Preliminaries and ILP Formulations

In this section, we will present certain mathematical preliminaries and ILP formulations in a general way which are used throughout the paper.

Modulo Function

We are using the definition of a modulo function described by Knuth [14]:

$$m = x \bmod y = x - \left\lfloor \frac{x}{y} \right\rfloor \cdot y \text{ , if } y \neq 0 \tag{1}$$

where x and y are any integer numbers. The resulting value m has the same sign as the divisor y. We reformulate (1) to the following equations:

$$x < (q+1) \cdot y \tag{2}$$
$$x \geq y \cdot q \tag{3}$$
$$m = x - q \cdot y \tag{4}$$

Equations (2) and (3) implement the floored division with q holding the result of $\lfloor \frac{x}{y} \rfloor$. Variable y is assumed to be always non-zero. The variable m is set to the modulo result by Equation (4). In case we assume the divisor y to be constant, equations (2) - (4) are linear and can be used inside an ILP formulation.

Conditional Assignment of ILP Variables

The following conditional assignment is given:

$$a = \begin{cases} u & \text{if } c = 1, \\ v & \text{else.} \end{cases} \tag{5}$$

Table 2 ILP decision variables used

Symbol	Description
d_ν	Additional number of stall cycles preceding an explicit data access during the execution of block ν in comparison to the analysis results.
$l_{\nu,\mu}$	Additional number of cycles needed due to bus stalling during the execution of jump correction code from block ν to μ in comparison to the analyzed interval. This variable does not contain the cycles required for the execution of the code, solely the cycles needed to gain the first grant.
$\boldsymbol{o}_\nu^{\text{In}}$	The incoming bus offset interval at ν.
$\boldsymbol{o}_{\nu,\mu,\text{W}}^{\text{Out}}$	Interval variable representing the outgoing bus offset interval at BB ν *with* considering potential jump correction to its successor BB μ.
$\boldsymbol{o}_{\nu,\text{WO}}^{\text{Out}}$	Interval variable representing the outgoing bus offset interval at BB ν *without* considering a potential jump correction.
$o_{\nu,\text{low}}$, $o_{\nu,\text{high}}$	The lower and upper elements of the corresponding bus offset interval $\boldsymbol{o}_\nu$.
r_ν^{Data}	The number of cycles required to receive a bus grant to access shared data at the BB ν.
r_ν^{Jump}	The number of cycles required to receive a bus grant block during the execution of jump correction code at BB ν.
w_ν	The accumulated WCET starting at BB ν.
x_ν	Binary variable representing whether BB ν is assigned to the SPM (x_ν=1) or not.

Table 3 Miscellaneous symbols used

Symbol	Description
$\boldsymbol{A}_{\nu,\text{Flash}}^{\text{In}}$, ($\boldsymbol{A}_{\nu,\text{SPM}}^{\text{In}}$)	Incoming bus offset interval at BB ν if the whole program is allocated to Flash (SPM).
$\boldsymbol{A}_{\nu,\text{Flash}}^{\text{Out}}$, ($\boldsymbol{A}_{\nu,\text{SPM}}^{\text{Out}}$)	Outgoing bus offset interval at BB ν if the whole program is allocated to Flash (SPM).
$C_{\nu,\text{Flash}}$,($C_{\nu,\text{SPM}}$)	WCET of 1 execution of BB ν when allocated to the Flash memory (SPM).
F_{SPM}	Access delay of the SPM.
F_{Flash}	Access delay of the Flash memory (bus grant acquired).
G_ν	Expected timing gain if a BB ν is assigned to the SPM in comparison to a Flash allocation.
H_ν	Binary constant set to 1 in case BB ν contains an instruction which potentially accesses the `.data` section
I	A core ID (0 ... N_c-1).
$J_{\mu,\nu,\text{SPM}}$	Cycles needed for the execution of jump correction code from BB ν (in SPM) to μ (in Flash), neglecting the required pipeline refill.
N_c	Number of cores.
P	Total bus period in cycles.
$Q_{\text{Flash}\rightarrow\text{SPM}}$	Bus offset after the execution of jump correction code in case the source BB is placed inside the Flash memory and its target BB resides in the SPM.
R_ν^{Data}	Required number of stall cycles to receive the bus grant for the shared memory access at block ν, accounted by the analysis.
S_ν	Code size of basic block ν.
S_{SPM}	Total size of a private SPM.
$\boldsymbol{T}_{\nu,\text{SPM}}$	Execution time window of BB ν as a result of an BCET/WCET analysis when the whole program is allocated to the SPM.
ν, μ	Indexes representing BBs.

with a, u, v and c being ILP variables, whereas the condition variable c is restricted to Boolean values. We are expressing equation (5) as a set of inequations in order to formulate if-then-else structures inside an ILP model.

$$a \geq u - (1 - c) \cdot M \tag{6}$$
$$a \leq u + (1 - c) \cdot M \tag{7}$$
$$a \geq v - c \cdot M \tag{8}$$
$$a \leq v + c \cdot M \tag{9}$$

We are using the so-called big-M method, where M is a sufficiently large constant. Equations (6) and (7) fix variable a to the value of u in case $c = 1$, otherwise a is only constrained to satisfy $u - M \leq a \leq u + M$. Analogously, equations (8) and (9) force the a to the value of v in case $c = 0$, while in the opposite case a is solely constrained to $v - M \leq a \leq v + M$.

Min/Max Function

Given are the following two functions:

$$\max(x, y) \tag{10}$$
$$\min(x, y) \tag{11}$$

with x and y being ILP variables. In order to express these two functions in terms of ILP constraints, we first create an ILP variable c restricted to Boolean values, used as a condition.

$$y \leq x + c \cdot M \tag{12}$$
$$x \leq y + (1 - c) \cdot M \tag{13}$$

M is a sufficiently large constant. Equations (12) and (13) set c to 1 in case $y > x$, otherwise c is forced to 0. Based upon this, the functions $\max(x, y)$ and $\min(x, y)$ can be represented using the following case statement:

$$\max(x, y) = \begin{cases} y & \text{if } c = 1, \\ x & \text{else.} \end{cases} \tag{14}$$

$$\min(x, y) = \begin{cases} x & \text{if } c = 1, \\ y & \text{else.} \end{cases} \tag{15}$$

The case statement used in equations (14) and (15) can be represented using the conditional assignment of ILP variables as shown before.

5.3 Base Model

The optimization introduced by Falk and Kleinsorge [5] uses an ILP model which is based on the model presented by Suhendra et al. [18]. We are using the exemplary program shown in Figure 5 to demonstrate the model. This model will be extended in Section 5.4 to enable bus-awareness.

Before the actual generation of the ILP model, the program needs to be analyzed twice in terms of WCET. Conventional WCET analyzers like AbsInt aiT [1] are not able to analyze bus-related latencies in a multicore setup, thus leading to an insufficient timing accuracy. However, analysis methods like those proposed by Kelter et al. [12] or Chattopadhyay et

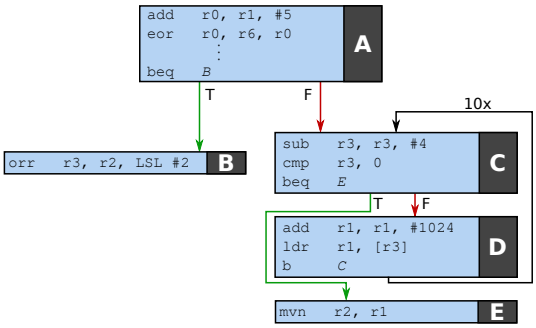


Figure 5 Exemplary CFG II.

al. [3] can be used to analyze multicore architectures including bus-related effects. The first analysis run is done with all basic blocks assigned to the Flash memory, while the second analysis is executed with all BBs assigned to a (virtually enlarged) SPM. The analysis runs are executed using multicore-enabled analyzing methods, so the analyzed timings include bus-related timings. The results of these analysis runs can be used to extract the net WCET per basic block ν, namely $C_{\nu,\text{Flash}}$ and $C_{\nu,\text{SPM}}$, denoting to which memory basic block ν was assigned to. Potential bus-related timings are included in the WCET of the basic blocks. Using these net WCETs, a timing gain G can be defined per basic block:

$$G_\nu = C_{\nu,\text{Flash}} - C_{\nu,\text{SPM}} \,. \tag{16}$$

G represents the timing profit in case a basic block is assigned to the SPM. Based on these timings, the WCET of the program can be modeled successively by introducing a variable w_ν which denotes the WCET of the path starting at basic block ν. We consider the timings of a block in terms of clock cycles, therefore integer variables are suitable to model the WCET of a basic block. The model is built up from the CFG's sink nodes.

$$w_B = C_{B,\text{Flash}} - x_B \cdot G_B \tag{17}$$
$$w_E = C_{E,\text{Flash}} - x_E \cdot G_E \tag{18}$$

x_ν is a Boolean decision variable and represents whether basic block ν is assigned to the SPM ($x_\nu = 1$) or not ($x_\nu = 0$). Subsequently, the control flow graph is traversed upwards. For each successor of a basic block, one individual constraint is added, containing its own net WCET and the corresponding successor's WCET. Regarding basic block A, this results in the following inequations:

$$w_A \geq C_{A,\text{Flash}} - x_A \cdot G_A + w_B \tag{19}$$
$$w_A \geq C_{A,\text{Flash}} - x_A \cdot G_A + w_{Loop} \tag{20}$$

The loop, which consists of basic blocks C and D, is modeled as a super-node. The partial WCET w_{Loop} starting at the entry of the loop is defined by its members, the loop bound and its successor.

$$w_{Loop} \geq c_{Loop} + C_{C,\text{Flash}} - x_C \cdot G_C + w_E \tag{21}$$
$$c_{Loop} \geq 10 \cdot w_{Entry} \tag{22}$$
$$w_{Entry} \geq C_{C,\text{Flash}} - x_C \cdot G_C + w_D \tag{23}$$
$$w_D \geq C_{D,\text{Flash}} - x_D \cdot G_D \tag{24}$$

$C_{C,\text{Flash}}$ is accounted for 11 times, since it is the head of the loop and is therefore executed one more time than the loop body. In order to restrict the number of basic blocks assigned to the scratchpad memory, an additional constraint has to be introduced:

$$S_{\text{SPM}} \geq x_A \cdot S_A + x_B \cdot S_B + x_C \cdot S_C + x_D \cdot S_D + x_E \cdot S_E \tag{25}$$

where S_{SPM} denotes the total size of the SPM and S_A the code size of basic block A. The overall WCET of the program can now be minimized by setting the objective function to minimize the WCET of the entry basic block, here A.

$$\min w_A \tag{26}$$

After assigning a block ν to the SPM, the control flow has to be repaired. This is required, since the program flow can be broken in several ways. In case the predecessor of ν did not contain an explicit jump to ν (e.g., see Figure 5: BB $A \rightarrow$ BB C), a new jump has to be introduced. However, even if an explicit jump was existing, it is likely to be not sufficient anymore, since the physical addresses of different memory regions presumably differ beyond the offset capabilities of a direct jump. Therefore, also the potential size and execution time of jump correction code has to be considered inside the ILP model using additional constraints. For the sake of simplicity and since this topic has already been discussed by Oehlert et al. [16], the additional ILP terms to consider such costs are omitted here. Nevertheless, the ILP model presented in this paper includes these constraints and considers their additional costs.

5.4 Bus-aware Extensions

As shown in Section 4, the WCETs extracted from the analysis runs of a BB are not safe anymore in case the current memory allocation of the program differs from the one used during analysis. More precisely, the WCET may differ if the temporal start of a basic block in regard to the bus schedule (the so-called *bus offset*) is different to the one analyzed. In regard to the program SPM allocation, this WCET change can be caused by two possibilities: If a basic block is allocated to the private SPM, but contains instructions which are explicitly accessing the shared memory, the bus offset during the access may be different, which may introduce additional waiting cycles. This is the case for data accessing instructions, since we assume the `.data` section to be placed inside the shared memory. In case the predecessor of a BB ν is assigned to the SPM (while ν resides in the shared memory), the bus offset at the execution start of ν may differ, which again may lead to a different WCET. Therefore, we extend the ILP model to predict these bus offsets and calculate bus-related penalties (or gains) based upon. We will retain the base model with its analyzed WCETs per block, but add these bus-related timing differences to it.

We assume all TDMA slots to be equally-sized and to be fixed to the length of exactly one Flash memory access delay F_{Flash}. This restriction enforces all possible accesses to the shared memory to be initiated during the first cycle of each core's bus slot. Thereby, all bus offsets at the beginning of a basic block placed inside the Flash are fixed and identical to the ones analysed. Accesses to the shared memory now serve as a kind of synchronization point. Since the bus offsets are known to be equal to the offsets during analysis, all WCETs of basic blocks placed inside the Flash can be safely extracted from the analysis again. In regard to the ARM7TDMI architecture used for evaluation purposes, this slot length restriction is acceptable in terms of timing since it fetches every instruction piecewise. Methods to relax this restriction while keeping the accuracy is part of our future work.

(a) Before sub basic block splitting **(b)** With sub basic block splitting applied

Figure 6 An exemplary BB containing an instruction with an explicit access to shared memory.

Additionally, all basic blocks which contain instructions with potential access to a shared memory are split up into sub basic blocks. These sub basic blocks either consist of *multiple* instructions which never access the shared data memory, or exactly *one* instruction which may then access the Flash. Example: An arbitrary BB Z is shown in Figure 6a. This basic block contains an instruction which accesses shared memory.

Figure 6b shows the sub basic blocks created from the former basic block Z. Sub basic block Z' now solely consists of the shared data memory accessing instruction.

This division of basic blocks is done in order to obtain the WCETs on a more detailed scale, since common WCET analyzers return the WCET per BB as the lowest granularity available. Using the timings of the sub basic blocks and the initial bus offset, the bus offset during the access can be predicted without further modifying existing analyzing techniques.

Bus Offset Calculation

For each sub basic block, the incoming and outgoing bus offsets are determined inside the ILP model. The incoming bus offset denotes the bus offset at the beginning of the execution of a basic block. In analogy to this, the outgoing bus offset describes the bus offset at the end of the execution of a basic block. Due to the different execution contexts of a basic block, it can have different execution times, varying between its Best-Case Execution Time (BCET) and its WCET. In this particular setup, these differing executions times are caused by pipeline effects or instructions with a possible varying execution time. Thus, the bus offset cannot be described as a scalar, but is rather an interval. This offset interval contains the lowest offset possible as well as the greatest. Inside the ILP model, an offset interval $\boldsymbol{o}$ is represented as two integer variables, o_{low} and o_{high}. The range of an offset variable like o_{low} or o_{high} is limited to the range of [0, P-1], where P is the total length of one bus period. In case of a wrap-around, i.e., $o_{\text{low}} > o_{\text{high}}$, the whole bus period is considered as a safe over-approximation. For every sub basic block ν, an offset interval $\boldsymbol{o}^{\text{In}}_{\nu}$ is added to the ILP model, representing the incoming bus offset interval of sub basic block ν. This offset interval is calculated as follows:

$$\boldsymbol{o}^{\text{In}}_{\nu} = \begin{cases} \boldsymbol{A}^{\text{In}}_{\nu,\text{Flash}} & \text{if } x_{\nu} = 0, \\ \bigcup_{\mu \in \text{Pred}(\nu)} \boldsymbol{o}^{\text{Out}}_{\mu,\nu,\text{W}} & \text{else.} \end{cases} \tag{27}$$

In case the Boolean SPM allocation variable x_{ν} is set to 0 (i.e., sub basic block ν would be assigned to the Flash memory), the incoming offset interval is equal to the interval extracted from the „all-in-Flash“ analysis. This is valid, since due to our restriction of slot lengths, we know that the actual execution start of a block placed inside the Flash memory can only happen at one single bus offset which we can extract from the analysis results. The potential

difference between the execution end of a preceding block and $\boldsymbol{o}_{\nu}^{\mathrm{In}}$ is later considered by adding a penalty timing to the preceding block.

If block ν is assigned to the SPM, $\boldsymbol{o}_{\nu}^{\mathrm{In}}$ has to be determined as the union of the outgoing bus intervals over all predecessors. The union over two given offset intervals $\boldsymbol{o_\nu}$ and $\boldsymbol{o_\mu}$ is defined using the following equations:

$$\boldsymbol{o_\nu} = (o_{\nu,\mathrm{low}}, o_{\nu,\mathrm{high}}) \tag{28}$$

$$\boldsymbol{o_\mu} = (o_{\mu,\mathrm{low}}, o_{\mu,\mathrm{high}}) \tag{29}$$

$$\boldsymbol{o_\nu} \cup \boldsymbol{o_\mu} = (\min(o_{\nu,\mathrm{low}}, o_{\mu,\mathrm{low}}), \max(o_{\nu,\mathrm{high}}, o_{\mu,\mathrm{high}})) \tag{30}$$

The min() and max() functions used in equation (30) are implemented as presented in Section 5.2. The interval $\boldsymbol{o}_{\mu,\nu,\mathrm{W}}^{\mathrm{Out}}$ represents the outgoing bus interval at sub basic block μ to successor ν, already including the effects of potential jump correction code.

Besides, for each successor of ν one offset interval $\boldsymbol{o}_{\nu,\mu,\mathrm{W}}^{\mathrm{Out}}$ has to be inserted into the ILP model (mind the switched indexes, since basic block ν is the source basic block in this case). This differentiation between successors is necessary, since e.g., a jump from the private SPM to the shared Flash memory requires an access to the bus to refill the pipeline, resulting in a contrasting bus offset in case the jump was not taken. It is defined as follows:

$$\boldsymbol{o}_{\nu,\mu,\mathrm{W}}^{\mathrm{Out}} = \begin{cases} \boldsymbol{o}_{\nu,\mathrm{WO}}^{\mathrm{Out}} & \text{if } x_\nu = x_\mu, \\ Q_{\mathrm{Flash}\to\mathrm{SPM}} & \text{if } x_\nu = \overline{x_\mu} = 0, \\ \boldsymbol{A}_{\mu,\mathrm{Flash}}^{\mathrm{In}} & \text{else.} \end{cases} \tag{31}$$

In case sub basic block ν and its successor μ are both placed inside the same memory, no additional jump correction is needed and the outgoing interval is identical to the offset interval not considering any jump correction, namely $\boldsymbol{o}_{\nu,\mathrm{WO}}^{\mathrm{Out}}$. If ν is placed inside the shared Flash memory and its successor μ in the SPM, an additional jump needs to be considered. Since the jump correction code will be placed inside the shared memory (where the ν resides), its bus offset at the end of the execution is always constant. This is due to the fact that the last instruction of a jump correction is always identical (an indirect jump). Because this instruction is placed inside the shared Flash memory, the fetching of it serves as a synchronization point. Therefore, the outgoing bus offset is constant under this circumstance and can be determined prior to the optimization. This offset is denoted as $Q_{\mathrm{Flash}\to\mathrm{SPM}}$.

In case sub basic block ν is assigned to the private SPM and its successor μ to the shared memory, a jump correction has to be done as well. Due to the fact that the successor is placed in the Flash memory, we can set the outgoing offset of ν to the analyzed incoming offset of μ (extracted from the „all-in-Flash“ analysis). During the execution of the final indirect jump as a part of the jump correction code, the processor needs to fetch the first instructions of block μ, which are placed in the shared memory. This is needed in order to refill the processor's pipeline, so the succeeding block does not start its execution with an empty pipeline (reminder: The ARM7TDMI architecture *always* assumes a jump not to be taken, so during the time the final indirect jump went into the execution phase, the pipeline was already filled with subsequent instructions from the SPM). Therefore, the outgoing offset of the jump correction code will be synchronized to the offset interval extracted from the analysis.

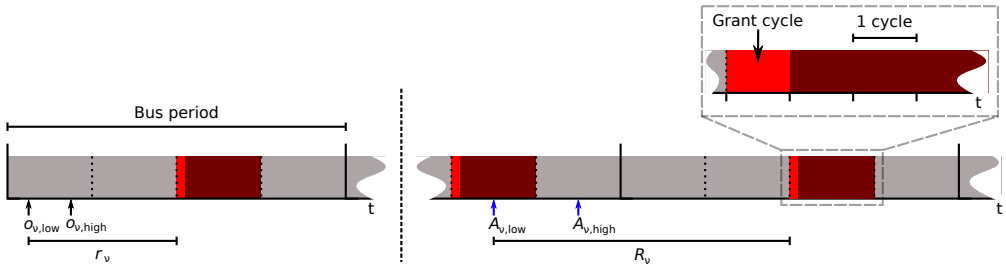


Figure 7 Two exemplary bus offset intervals $\boldsymbol{o}_\nu$ and $\boldsymbol{A}_\nu$ in regard to the bus schedule. The bus slot of the third core is highlighted, as well as the grant cycle during which an access can be issued.

Besides the outgoing bus offset interval including potential jump correction costs, a bus offset interval $\boldsymbol{o}^{\text{Out}}_{\nu,\text{WO}}$ is added to the ILP for each sub basic block ν, representing the outgoing bus offset without considering any jump correction. It can be determined as follows:

$$\boldsymbol{o}^{\text{Out}}_{\nu,\text{WO}} = \begin{cases} \boldsymbol{A}^{\text{Out}}_{\nu,\text{Flash}} & \text{if } x_\nu = 0, \\ \boldsymbol{A}^{\text{Out}}_{\nu,\text{SPM}} & \text{else if } H_\nu = 1, \\ (\boldsymbol{o}^{\text{In}}_\nu + \boldsymbol{T}_{\nu,\text{SPM}}) \bmod P & \text{else if } |\boldsymbol{T}_{\nu,\text{SPM}}| \leq P, \\ [0, P-1] & \text{else.} \end{cases} \tag{32}$$

In analogy to the incoming bus offset interval, the outgoing interval will be identical to the offset analyzed during the „all-in-Flash“ analysis run $\boldsymbol{A}^{\text{Out}}_{\nu,\text{Flash}}$ in case the sub basic block ν is placed inside the Flash memory. This is valid, since due to the slot lengths restriction we know that the bus offset during the start of the execution of block ν is fixed in this case. Therefore, its execution and also its outgoing bus offset interval will be always identical, independent from the temporal history.

If ν is assigned to the private SPM and has an explicit access to a shared memory region ($H_\nu = 1$), $\boldsymbol{o}^{\text{Out}}_{\nu,\text{WO}}$ will be set to the outgoing offset analysed during the „all-in-SPM“ analysis run. This is legitimate, since due to the sub basic block splitting, a sub basic block with a potential shared data memory access can only consist of this instruction itself. Therefore, the access can be regarded as a synchronization point, leading to the identical outgoing offset as resulted in the SPM analysis run.

Otherwise, the outgoing offset interval has to be calculated based on the incoming offset interval $\boldsymbol{o}^{\text{In}}_\nu$ and the analyzed execution time window (the difference between WCET and BCET) $\boldsymbol{T}_{\nu,\text{SPM}}$ when assigned to the SPM. However, in case the execution time window exceeds one whole bus period P, all possible bus offsets have to be considered.

Due to the nature of ILP, all possibilities of each ILP offset interval variable are calculated side by side and then chosen using a case distinction, implemented as presented in Section 5.2.

Bus-related Penalties

Based on the determined bus offset intervals per sub basic block, it is possible to predict the occurring timing related effects. For this purpose, we introduce two possible new ILP variables per sub basic block. The base penalty d_ν represents the additional cycles needed for the execution of a sub basic block ν due to an explicit access to a shared memory.

d_ν can also be negative if the different bus offset causes a better bus alignment in comparison to the previous analysis run. In this case, d_ν denotes a gain rather than a penalty. It is defined as follows:

$$d_\nu = \begin{cases} r_\nu^{\text{Data}} - R_\nu^{\text{Data}} & \text{if } x_\nu = H_\nu = 1, \\ 0 & \text{else.} \end{cases} \tag{33}$$

The ILP variable r_ν^{Data} describes the number of cycles needed at sub basic block ν to acquire a bus grant. It is required that ν is assigned to the private SPM and consists of an instruction which potentially accesses the shared memory through an explicit load/store instruction. Otherwise, r_ν^{Data} is always zero and does not need to be created.

Example: Figure 7 illustrates two exemplary bus offset intervals $\boldsymbol{o}_\nu$ and $\boldsymbol{A}_\nu$ in regard to the bus schedule. An arbitrary program is assumed to run on the third core of the system in its private SPM, possessing the third slot of each period which is highlighted. Since we restrict the initiation of a bus access to the first cycle of the corresponding slot, this cycle is highlighted in a lighter color as well. A sub basic block ν is assigned to the SPM and contains an access to the shared data memory. This access is tried to be issued at the shown bus offset interval $\boldsymbol{o}_\nu$. Considering the worst case, the processor has to stall at most r_ν cycles. Here, r_ν matches the ILP variable r_ν^{Data}, representing the greatest number of stalling cycles considering the current program allocation, until the bus grant is received.

The second offset interval $\boldsymbol{A}_\nu$ shown in the figure represents the bus offset interval extracted from the WCET analysis at the same sub basic block ν. Therefore, R_ν resembles the constant R_ν^{Data} from Equation (33), namely the number of cycles accounted by the WCET analyzer to gain the bus grant. Since the actual memory access delay is identical during both executions and already accounted, only the difference in stalling cycles until the bus grant is relevant to calculate. For this reason, the difference between r_ν^{Data} and R_ν^{Data} is calculated. Regarding Figure 7, the ILP-chosen allocation of blocks leads to a lower number of stall cycles needed at sub basic block ν in comparison to the „all-in-SPM“ allocation, since r_ν^{Data} is lower than R_ν^{Data}.

In order to define r_ν^{Data}, the number of cycles between the bus offset interval $\boldsymbol{o}_\nu^{\text{In}}$ and the next granted bus slot of the corresponding core is calculated.

$$a_{\text{low}} = \left(I \cdot F_{\text{Flash}} - o_{\nu,\text{low}}^{\text{In}}\right) \bmod P \tag{34}$$

$$a_{\text{high}} = \left(I \cdot F_{\text{Flash}} - o_{\nu,\text{high}}^{\text{In}}\right) \bmod P \tag{35}$$

$$r_\nu^{\text{Data}} = \begin{cases} P - 1 & \text{if } o_{\nu,\text{low}} \leq I \cdot F_{\text{Flash}} + 1 \leq o_{\nu,\text{high}}, \\ \max(a_{\text{low}}, a_{\text{high}}) & \text{else.} \end{cases} \tag{36}$$

The term $I \cdot F_{\text{Flash}}$ equals the bus offset of a core I at which a bus access is granted. The constant R_ν^{Data} is calculated in the same manner, but using the bus offsets extracted from the „all-in-SPM“ analysis run.

The second ILP variable introduced per sub basic block ν is $l_{\nu,\mu}$. This variable represents the potential additional bus waiting cycles during the execution of jump correction code. It is defined as follows:

$$l_{\nu,\mu} = \begin{cases} r_\nu^{\text{Jump}} & \text{if } x_\nu = \overline{x_\mu} = 1, \\ 0 & \text{else.} \end{cases} \tag{37}$$

Example: A sub basic block ν is assigned to the private SPM of the third core of a system, while its successor μ resides in the shared Flash memory. This circumstance requires the

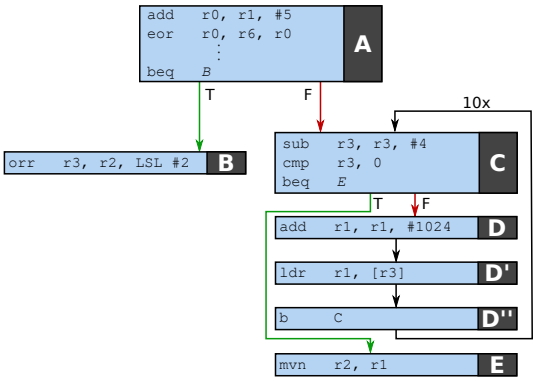


Figure 8 Exemplary CFG from Figure 5 with sub basic block splitting applied.

introduction of jump correction code subsequent to ν. During the end of the execution of the appended jump code, the first instructions of the succeeding block μ have to be fetched, so μ does not start with an empty pipeline. Since block μ is placed inside the shared Flash memory, the jump correction code will cause an additional bus access during the fetch of its leading instructions. Referring to Figure 7, $\boldsymbol{o}_\nu$ resembles the offset interval of sub basic block ν when such an access is tried to be issued. The variable r_ν^{Jump} then describes the greatest number of cycles which are needed, based on offset interval $\boldsymbol{o}_{\nu,\text{WO}}^{\text{Out}}$, to reach a valid bus slot.

The variable r_ν^{Jump} is determined in the same fashion as r_ν^{Data}, but utilizing the offset interval $\boldsymbol{o}_{\nu,\text{WO}}^{\text{Out}}$. Since these timing costs did not exist in the initial analysis runs, no subtractive term as in Equation (33) is added. All other timings of jump correction code are constant and can be calculated upfront. Those timings are already considered inside the jump correction costs of the base model.

Final ILP Model

The presented additions are integrated into the base ILP model. Prior to the initial WCET analysis runs, the sub base block splitting is applied.

Regarding the exemplary control flow graph shown in Figure 5, the additional variables d and l are added to the WCET constraints of the corresponding sub basic blocks. The CFG with splitting applied is shown in Figure 8.

$$w_A \geq C_{A,\text{Flash}} - x_A \cdot G_A + w_B + l_{A,B} \tag{38}$$
$$w_A \geq C_{A,\text{Flash}} - x_A \cdot G_A + w_{Loop} + l_{A,Loop} \tag{39}$$
$$w_B = C_{B,\text{Flash}} - x_B \cdot G_B \tag{40}$$
$$w_{Loop} \geq c_{Loop} + C_{C,\text{Flash}} - x_C \cdot G_C + w_E + l_{C,E} \tag{41}$$
$$c_{Loop} \geq 10 \cdot w_{Entry} \tag{42}$$
$$w_{Entry} \geq C_{C,\text{Flash}} - x_C \cdot G_C + w_D + l_{C,D} \tag{43}$$
$$w_D \geq C_{D,\text{Flash}} - x_D \cdot G_D + w_{D'} + l_{D,D'} \tag{44}$$
$$w_{D'} \geq C_{D',\text{Flash}} - x_{D'} \cdot G_{D'} + w_{D''} + l_{D',D''} + d_{D'} \tag{45}$$
$$w_{D''} \geq C_{D'',\text{Flash}} - x_{D''} \cdot G_{D''} + l_{D'',C} \tag{46}$$
$$w_E = C_{E,\text{Flash}} - x_E \cdot G_E \tag{47}$$
$$S_{\text{SPM}} \geq x_A \cdot S_A + x_B \cdot S_B + x_C \cdot S_C + ... + x_E \cdot S_E \tag{48}$$

The data access penalty d is only introduced to sub basic block D', since it is the only block with a potential access to a shared memory region. The objective function is kept from the base ILP. The constraints to consider the additional spatial costs of possible jump correction code are derived from the base ILP as well, yet not shown here to avoid unnecessary complications.

6 Evolutionary Algorithm

This section describes the genetic algorithm used as a reference for the ILP-based static bus-aware multicore SPM allocation optimization. The optimization is a classical genetic algorithm as described by Goldberg [7]. It starts with a set of individuals of which each holds a set of binary decision variables $x_{\nu,I}$ denoting whether basic block ν of core I will be assigned to SPM. We assume that there is no shared code between the cores.

Unless stated otherwise, a random selection is drawn from a uniform distribution. We create the initial set of N_{Ind} individuals as follows:

- The first individual is left with all basic blocks in Flash memory.
- For all other individuals, we virtually assign all blocks to the SPM and then randomly remove basic blocks of each core from SPM until the SPM memory is no longer overflowing.

For recombination of two individuals A and B, our tests showed good results with a simple one-point recombination with multi-bit mutation. We first randomly determine the core I to be crossed over. For the selected core, we randomly determine the position i at which the two individuals will be merged.

The new individual C will have the following new assignment:

$$C = A[0, i-1] \mid B[i, N_{B,I} - 1] \tag{49}$$

$N_{B,I}$ denotes the total amount of basic blocks contained by the task allocated to core I. The first i decision variables of the new individual C will be taken from individual A, while the second part is taken from individual B. Subsequently we randomly choose a number of maximum mutations M for the SPM assignment in the crossed over core I.

We then randomly select M basic blocks to mutate. Whether or not the assignments of these randomly selected basic blocks will be toggled is then again randomly determined for each with a user-definable probability.

Using the allocation determined by the new individual C, a jump correction is performed in order to repair the control flow graph. If this final SPM assignment including the inserted jump correction code fits into the physically available SPM, no repair is necessary. If this is not the case, either the number of basic blocks assigned to the SPM was too high, or the jump correction code overfilled the SPM boundaries. In this case, we again randomly remove blocks from the SPM and perform a jump correction respectively until the assignment is valid.

Finally, the new individual is analyzed using the WCET analyzing methods proposed by Kelter [11] to assess the new WCET. Because the WCET analysis automatically analyzes bus penalties and accounts for them in the task's worst execution timing behavior, the genetic optimization is inherently bus-aware.

For the next generation, the N_{Ind} fittest individuals are selected. The allocation of a program on one core does not interfere with the execution of a program on another core due to the TDMA schedule with fixed slot lengths. As a result, our fitness function can simply be chosen to minimize the sum over the WCETs of all tasks.

The optimization terminates if the WCET reduction of any core over 2 generations is smaller than a user-definable threshold ϵ, or a user-definable amount of time has gone by.

7 Evaluation

The presented bus-aware ILP-based instruction SPM allocation and the one based on evolutionary algorithms were implemented for a multicore ARM7TDMI architecture described in Section 3. We use the resulting WCET of the bus-unaware ILP-based instruction SPM allocation (described in Section 5) as a baseline. The access delay of the private SPM is assumed to be 1 cycle, while the access delay of the shared Flash memory is assumed as 6 cycles in case the bus grant is acquired. In reference to the bus slot length restriction described in Section 5.4, the bus schedule consists of equally-sized bus slots. The length of a slot is set to the access delay of the Flash memory F_{Flash} (6 cycles). The (sub) basic blocks assigned to the scratchpad memory are loaded into the memory prior to the actual execution of the program. Therefore, the initial cycles required to transfer the corresponding blocks into the SPM do not need to be considered in terms of the WCET of a program.

The MRTC benchmark suite [8] was used for evaluation purposes with annotated loop bounds from the TACLeBench project [4]. The `duff` benchmark was excluded from the set of benchmarks, since it contains an irregular loop which can not be modeled using our current ILP models. Besides, the benchmarks `petrinet` and `statemate` were excluded due to timeouts (execution time > 15h) during the bus-aware ILP-based optimization. However, the evolutionary approach was able to terminate in the given time limit for these benchmarks. The WCET analyses for the ARM7TDMI multicore platform were done by using methods described by Kelter [11].

For the instruction SPM allocation based on evolutionary algorithms, the parameters were carefully chosen to compromise between execution time and effectiveness:

- Initial population: 20
- Number of parents per generation: 20
- Number of offspring individuals: 20
- Maximum number of generations: 50
- Mutation probability: 0.2
- Multibit mutation
- Single-point crossover

All evaluations were performed on an Intel Xeon Server. ILPs were solved using Gurobi 7.0.1 using 20 threads. All benchmarks were compiled with the WCET-aware C compiler (WCC) [6] and the `-O2` flag applied which enables several ACET-oriented compiler optimizations. The private scratchpad memory size is set individually for each benchmark, adjusting it to 50% relative to the benchmark's code size. All optimizations were performed for a dualcore, quadcore and octacore target platform.

Figure 9 shows the WCET of each benchmark optimized using the bus-aware ILP-based allocation and based on evolutionary algorithms, relative to the optimized WCET using the bus-unaware ILP-based optimization. The programs were executed on one core of the dualcore platform. Since we are using a TDMA scheduling policy for the bus, it is irrelevant under which combinations the benchmarks are executed. The execution of a benchmark on one core does not influence the WCET of a benchmark being executed on another core, since the bus slot lengths are fixed for each platform. On average (geometric mean), the bus-aware ILP-based optimization results in a WCET 23% lower than the unaware ILP-based optimization. The benchmark `crc` yields the greatest reduction in terms of WCET with

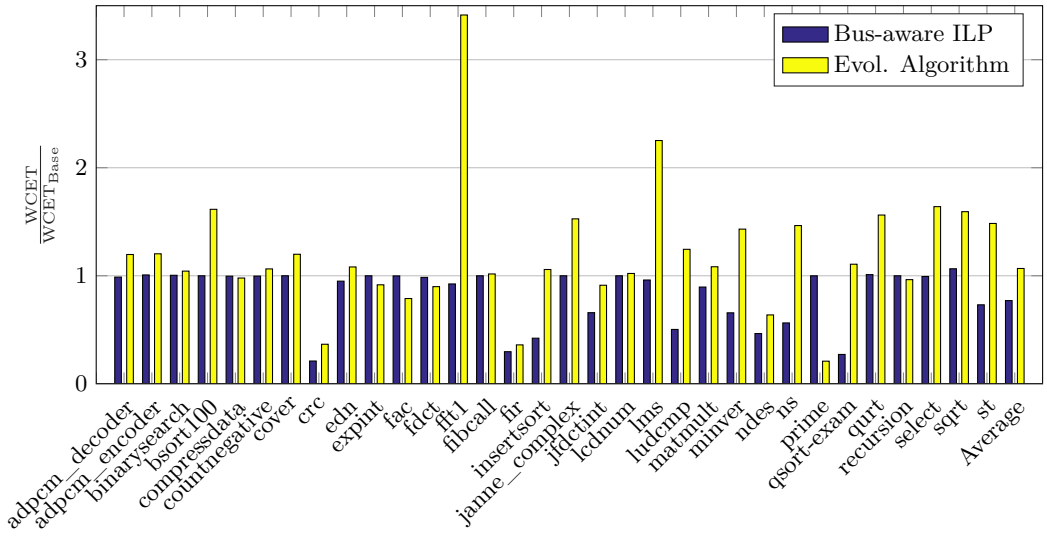


Figure 9 Relative optimized WCETs using the dualcore ARM7TDMI platform.

79%, while `sqrt` results in a 6% worse WCET. The bus-aware ILP-based optimization can lead to a worse allocation in comparison to the unaware model in certain cases, since the model potentially allocates blocks in a pessimistic way due to uncertainties related to bus effects. Because the bus-unaware ILP model does not consider potential bus effects, it can result in a lower WCET in case the bus-aware model is overly pessimistic.

The instruction SPM allocation based on evolutionary algorithms returns on average a WCET which is 7% greater than the baseline WCET. The benchmark with the greatest WCET reduction is `prime` with 79%, for which the bus-aware ILP optimization did not return any noticeable WCET reduction. This is mostly caused by the fact that the ILP model is not able to represent execution contexts, which leads to a harsh over-approximation regarding the total WCET. This behavior of the benchmark `prime` can be observed independent from the number of cores per system. On the contrary, the evolutionary algorithm is able to evaluate these effects inherently, since the used WCET analyzing methods consider execution contexts. However, the benchmark `fft1` results in a WCET which is 241% greater than the baseline WCET.

Figure 10 shows the results of the same experiment setup, but using a quadcore platform. The results resemble the experiment on a dualcore platform, yet the extremes are pushed further to either direction. In case of the bus-aware ILP-based optimization, still the `crc` benchmark results in the best reduction with 86%. Yet again, `sqrt` represents the worst result in this configuration with a 5% greater WCET. The bus-aware ILP optimization results in a 28% lower timing in comparison to the baseline WCET on average per benchmark when performed on a quadcore system.

On average, the evolutionary-based optimization leads to worse results in comparison to the dualcore experiment with an average WCET 12% higher than the baseline. With a WCET reduction of 72% compared against the baseline, the benchmark `fir` provides the best result using the evolutionary approach on a quadcore platform. Same as in the dualcore experiment, the `fft1` benchmark returns the worst result also in this configuration with a WCET 437% greater than the baseline WCET.

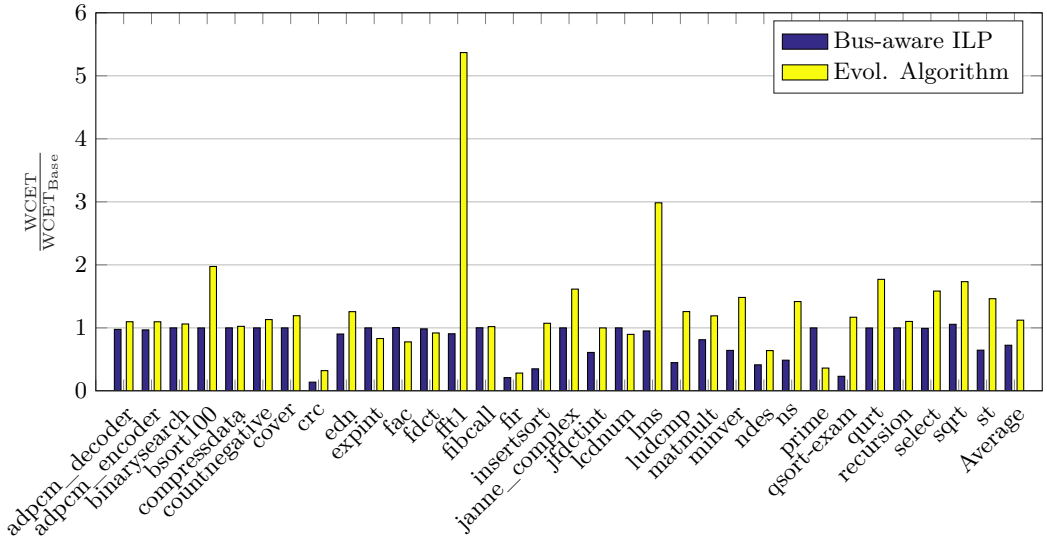


Figure 10 Relative optimized WCETs using the quadcore ARM7TDMI platform.

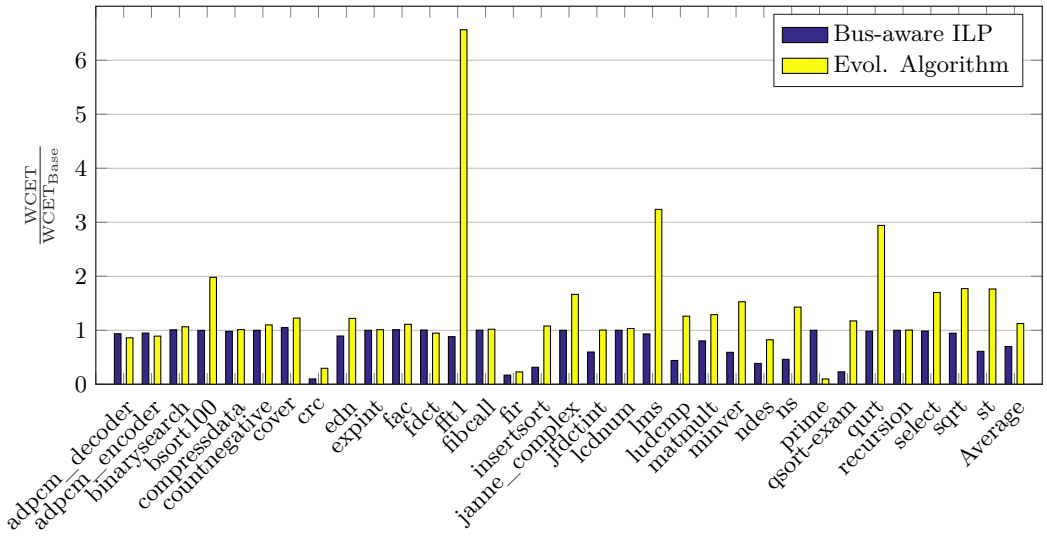


Figure 11 Relative optimized WCETs using the octacore ARM7TDMI platform.

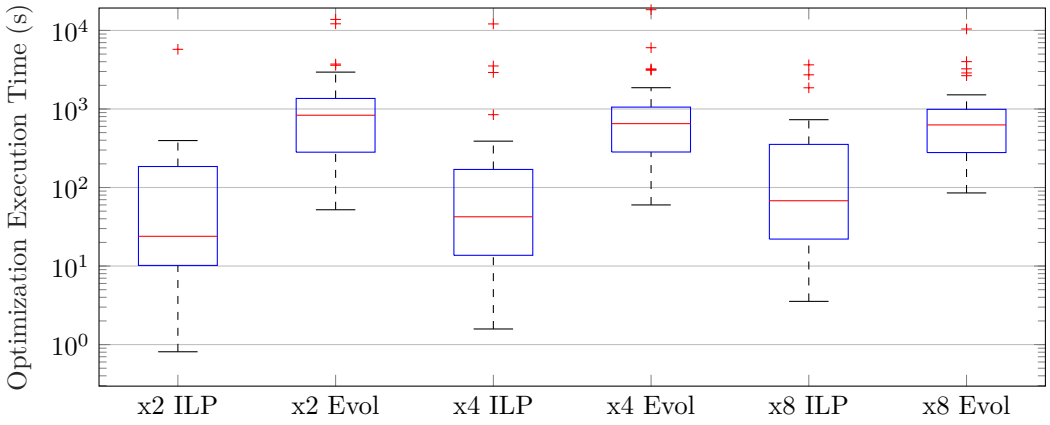


Figure 12 Overall Execution Time of the Optimizations for different Platforms.

In Figure 11, the results extracted from the experiments performed on an octacore ARM7TDMI platform are shown. In general, the results resemble a similar pattern as seen in the previous experiments. The bus-aware ILP-based optimization results in average a WCET which is 30% lower than the baseline WCET. The benchmark `crc` yields the lowest WCET in comparison to the bus-unaware ILP optimization for the octacore platform with a 90% lower WCET, while `cover` results in a 5% higher WCET.

The average WCET per benchmark using the evolutionary-based instruction SPM allocation is 12% higher in comparison to the baseline WCET. The greatest timing reduction using the evolutionary-based optimization in the octacore configuration is achieved for the `prime` benchmark with a 90% lower WCET in comparison to the bus-unaware ILP optimization, while `fft1` results in a 556% higher WCET.

Overall, it is observable that the advantage of the bus-aware ILP optimization in comparison to its unaware opponent rises with the number of cores in the system. This conclusion is expectable, since the impact of bus-related effects also increases with the number of cores, since the possible stalling times rise with a longer bus period. Meanwhile, the evolutionary-based optimization's quality degrades with an increasing number of cores. This is likely to be the case, since the penalty induced by only *one* badly allocated basic block heavily increases with an increasing number of cores, due to greater stalling times. Therefore, the evolutionary algorithm requires a larger number of generations to reach a proper allocation. Since we set a fixed upper bound of the maximum generations, it will get more likely that the optimization will be canceled before an adequate allocation is reached with an increasing number of cores.

Figure 12 shows the overall execution times of the bus-aware ILP-based instruction SPM allocation and the evolutionary-based approach, separated according to the number of cores used in the platform.

The central mark of each box denotes the median, while the edges depict the 25th and 75th percentiles. The maximum whisker length is defined as 1.5 times the difference between the 75th and 25th percentile. Execution times outside the region between the whiskers are depicted with a „+“-symbol. It is noticeable that independent from the number of cores used inside the system, the execution time of the ILP-based optimization is distinctively lower in terms of the median in comparison the evolutionary-based optimization. This is likely to be caused by the plenty of analysis runs required by the evolutionary algorithm, while the ILP-based approach only relies on two runs. The time required by the evolutionary algorithm could be decreased by decreasing the number of maximum generations or the

number of individuals per generation. However this would likely lead to a decreased quality of optimization. Furthermore, the evolutionary approach's timing can be still improved by enabling parallelism, which is yet to be implemented.

8 Conclusion and Future Work

We showed a precise bus-aware ILP-based instruction scratchpad memory allocation to reduce the worst-case execution time of a program. This approach includes the dynamic prediction of bus offsets and their resulting timing effects inside the ILP model. We showed, that using this optimization, it is possible to reduce the WCET of a program in comparison to a bus-unaware ILP-based optimization by on average 26%, with an average runtime significantly lower than the genetic approach. On the downside, the approach heavily increases the complexity of the underlying model, especially for data intensive programs.

Besides, we showed a first approach based on evolutionary algorithms for instruction SPM allocation in multicore platforms which considers the timing effects of the bus architecture. Using this approach, the experiments returned a WCET reduction up to 90% in comparison the a bus-unaware ILP-based optimization.

As a part of future work, we plan to consider caches in our bus-aware ILP model. Integrating the results of cache analyses could greatly improve the WCET of a program furthermore, since less bus accesses would be required.

Further we plan to relax the bus slot length restriction discussed in this paper and expand the model to data scratchpad memory allocation.

Besides, we plan to further fine-tune our approach based on evolutionary algorithms to speed up the convergence to near-optimal results. Therefore, we intend to introduce parallelism during the fitness calculation of the individuals.

References

1 AbsInt Angewandte Informatik, GmbH. aiT Worst-Case Execution Time Analyzers, 2017.

2 Sudipta Chattopadhyay and Abhik Roychoudhury. Static Bus Schedule Aware Scratchpad Allocation in Multiprocessors. In *Proceedings of the 2011 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES'11, pages 11–20, New York, NY, USA, 2011. ACM. doi:10.1145/1967677.1967680.

3 Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. Modeling Shared Cache and Bus in Multi-cores for Timing Analysis. In *Proceedings of the 13th International Workshop on Software and Compilers for Embedded Systems*, SCOPES'10, pages 6:1–6:10, New York, NY, USA, 2010. ACM. doi:10.1145/1811212.1811220.

4 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 *Proceedings of the 16th International Workshop on Worst-Case Execution Time Analysis (WCET)*, OpenAccess Series in Informatics (OASIcs), pages 2:1–2:10, Toulouse, France, 2016. doi:10.4230/OASIcs.WCET.2016.2.

5 Heiko Falk and Jan C. Kleinsorge. Optimal Static WCET-aware Scratchpad Allocation of Program Code. In *Proceedings of the 46th Annual Design Automation Conference*, DAC, pages 732–737, San Francisco, CA, USA, 2009. doi:10.1145/1629911.1630101.

6 Heiko Falk and Paul Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *Real-Time Systems*, 46(2):251–300, 2010. doi:10.1007/s11241-010-9101-x.

7 David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, Inc., Boston, MA, USA, 1989.

8 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)*, OASIcs, pages 136–146, Dagstuhl, Germany, 2010. doi:10.4230/OASIcs.WCET.2010.136.

9 Morteza Mohajjel Kafshdooz and Alireza Ejlali. Dynamic Shared SPM Reuse for Real-Time Multicore Embedded Systems. *ACM Transactions on Architecture and Code Optimization*, 12(2):12:1–12:25, May 2015. doi:10.1145/2738051.

10 T. Kelter, H. Borghorst, and P. Marwedel. WCET-aware scheduling optimizations for multi-core real-time systems. In *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pages 67–74, Samos, Greece, July 2014. doi:10.1109/SAMOS.2014.6893196.

11 Timon Kelter. *WCET Analysis and Optimization for Multi-Core Real-Time Systems*. PhD thesis, TU Dortmund, Department of Computer Science, Dortmund, Germany, March 2015.

12 Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. Static analysis of multi-core TDMA resource arbitration delays. *Real-Time Systems*, 50(2):185–229, 2014. doi:10.1007/s11241-013-9189-x.

13 Y. Kim, D. Broman, J. Cai, and A. Shrivastaval. WCET-aware dynamic code management on scratchpads for Software-Managed Multicores. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 179–188, Berlin, Germany, April 2014. doi:10.1109/RTAS.2014.6926001.

14 Donald Ervin Knuth. *Fundamental algorithms*. The art of computer programming. Addison-Wesley, Reading, MA, USA, 3. ed edition, 1997.

15 Yu Liu and Wei Zhang. Scratchpad Memory Architectures and Allocation Algorithms for Hard Real-Time Multicore Processors. *Journal of Computing Science and Engineering*, 9(2):51–72, 2015. doi:10.5626/JCSE.2015.9.2.51.

16 Dominic Oehlert, Arno Luppold, and Heiko Falk. Practical Challenges of ILP-based SPM Allocation Optimizations. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*, SCOPES'16, pages 86–89, New York, NY, USA, 2016. ACM. doi:10.1145/2906363.2906371.

17 Vivy Suhendra and Tulika Mitra. Exploring Locking & Partitioning for Predictable Shared Caches on Multi-cores. In *Proceedings of the 45th Annual Design Automation Conference*, DAC'08, pages 300–303, New York, NY, USA, 2008. ACM. doi:10.1145/1391469.1391545.

18 Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. WCET Centric Data Allocation to Scratchpad Memory. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, RTSS'05, pages 223–232, Washington, DC, USA, 2005. IEEE Computer Society. URL: 10.1109/RTSS.2005.45.

Contention-Aware Dynamic Memory Bandwidth Isolation With Predictability in COTS Multicores: An Avionics Case Study*

Ankit Agrawal[1], Gerhard Fohler[2], Johannes Freitag[3], Jan Nowotsch[4], Sascha Uhrig[5], and Michael Paulitsch[6]

1 **Chair of Real-Time Systems, Technische Universität Kaiserslautern, Kaiserslautern, Germany**
agrawal@eit.uni-kl.de
2 **Chair of Real-Time Systems, Technische Universität Kaiserslautern, Kaiserslautern, Germany**
fohler@eit.uni-kl.de
3 **Airbus Innovations, Munich, Germany**
johannes.freitag@airbus.com
4 **Airbus Innovations, Munich, Germany**
jan.nowotsch@airbus.com
5 **Airbus Innovations, Munich, Germany**
sascha.uhrig@airbus.com
6 **Base Systems, Thales Austria GmbH, Vienna, Austria†**
michael.paulitsch@thalesgroup.com

Abstract

Airbus is investigating COTS multicore platforms for safety-critical avionics applications, pursuing helicopter-style autonomous and electric aircraft. These aircraft need to be ultra-lightweight for future mobility in the urban city landscape. As a step towards certification, Airbus identified the need for new methods that preserve the ARINC 653 single core schedule of a Helicopter Terrain Awareness and Warning System (HTAWS) application while scheduling additional safety-critical partitions on the other cores.

As some partitions in the HTAWS application are memory-intensive, static memory bandwidth throttling may lead to slow down of such partitions or provide only little remaining bandwidth to the other cores. Thus, there is a need for dynamic memory bandwidth isolation. This poses new challenges for scheduling, as execution times and scheduling become interdependent: scheduling requires execution times as input, which depends on memory latencies and contention from memory accesses of other cores – which are determined by scheduling. Furthermore, execution times depend on memory access patterns.

In this paper, we propose a method to solve this problem for slot-based time-triggered systems without requiring application source-code modifications using a number of dynamic memory bandwidth levels. It is NoC and DRAM controller contention-aware and based on the existing interference-sensitive WCET computation and the memory bandwidth throttling mechanism. It constructs schedule tables by assigning partitions and dynamic memory bandwidth to each slot on each core, considering worst case memory access patterns. Then at runtime, two servers – for processing time and memory bandwidth – run on each core, jointly controlling the contention between the cores and the amount of memory accesses per slot.

* The research leading to these results was funded within the EMC^2 project by the EU ARTEMIS Joint Undertaking under grant agreement no. 621429.
† The work presented here was carried out while the author was at Airbus Innovations.

29th Euromicro Conference on Real-Time Systems (ECRTS 2017).
Editor: Marko Bertogna; Article No. 2; pp. 2:1–2:22

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

As a proof-of-concept, we use a constraint solver to construct tables. Experiments on the P4080 COTS multicore platform, using a research OS from Airbus and EEMBC benchmarks, demonstrate that our proposed method enables preserving existing schedules on a core while scheduling additional safety-critical partitions on other cores, and meets dynamic memory bandwidth isolation requirements.

1998 ACM Subject Classification D.4.7 Organization and Design

Keywords and phrases dynamic memory bandwidth isolation, safety-critical avionics, COTS multicores

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.2

1 Introduction

For future mobility, Airbus is pursuing autonomous aircraft targeting urban landscape to ease traffic, for instance, Uber-like CityAirbus [12], and the Vahana aircraft [17]. These helicopter-style aircraft will be electrically powered, requiring ultra-light weight to boost their power-to-weight ratio. They will need most avionics applications used in current aircraft, along with a DAL-A (the highest design assurance level) sense-and-avoid application for autonomous flying, unavailable today. Further, the electronic systems used in current aircraft need to be redesigned to reduce size, weight, and power consumption (SWaP), by integrating more avionics applications on the same number of processors, which is not feasible with single-core processors. The power consumption of current electronic systems is marginal compared the envisaged electric propulsion system. However, limiting it will eliminate the need for active cooling, further reducing SWaP. It will also eliminate the risk of a failure of the cooling system. Airbus is investigating COTS multicores to meet these future demands.

Safety-critical avionics hardware and software demand certification from certification authorities, which requires that the processes used in the design of digital hardware must relate to the DAL of the intended use [9]. However, COTS multicores are designed primarily for mass market and average-case performance and do not customarily follow DAL-based design processes. The CAST-32a position paper [27] describes the issues in the certification of COTS multicores, but the concrete implementation details are still open. Airbus is aiming at an incremental transition step towards the use of full COTS multicore performance: In the first step existing safety-critical single-core avionics application will be ported to a COTS multicore by preserving the original ARINC 653 schedule as well as the source code while executing it on only one core. Additional applications must be assigned to another core of the COTS multicore. This step reduces certification cost since documentation and verification of the software is already available. In the second (future) step, an application can be distributed over all the available cores. This paper focuses on the first step.

The Helicopter Terrain Awareness and Warning System (HTAWS), selected as reference application, is a pilot supporting system rated as DAL-C. It shows the helicopter pilot the surrounding topographical layout (including large buildings, power lines) with “flyable” areas together with warnings when the helicopter approaches rough terrain, e.g., when vision is degraded. Such a system also needs to be integrated into future autonomous aircraft to allow the aircraft to perform autonomous path planning and in-flight re-planning. HTAWS application is currently implemented on a dedicated avionics computer which is not feasible for ultra-light autonomous aircraft due to their SWaP constraints.

One of the major obstacles in certifying COTS multicores for use in safety-critical avionics systems is the contention between cores. The contention between cores arises due to the

implicit sharing of hardware resources like the network-on-chip, memory controller and main memory. When left unmitigated, it can slow down the partitions, resulting in deadline misses or even system failure. For example, the authors in [20] showed in an experimental setup, using the P4080 8-core COTS multicore platform, that the latency of a single store request is increased by a factor of 25.82 when the number of active cores is increased from 1 to 8. Approaches based on static memory bandwidth throttling, such as MemGuard [31], have been shown to be useful and gained adoption. In addition to controlling the amount of memory bandwidth available to cores, they allow the use of existing scheduling algorithms with minor modifications, as the effect of throttling can be seen as a slower processor. Measurements from the HTAWS application on a COTS multicore with only one active core show that some partitions are memory-intensive. Using static memory bandwidth isolation may lead to slow down of such partitions or provide only little remaining bandwidth to the other cores.

Static memory bandwidth isolation mechanisms assign a constant amount of memory bandwidth Q_{sm_n} to each core N_n before runtime. This limits the worst-case number of contentions any partition on a core can experience at any time, which allows computation of execution time separately from scheduling. However, under dynamic memory bandwidth isolation, as scheduling impacts the contention from the other cores and the dynamic memory bandwidth, execution time computation and scheduling cannot be performed independently. Specifically, the execution time of a partition depends on (a) the time taken for memory accesses which depends on the contention from the other cores, and (b) its worst-case memory access pattern and the dynamic memory bandwidth assigned in each slot during its execution.

Contention-aware dynamic memory bandwidth throttling can solve these issues but is not straightforward as it introduces an interdependency between scheduling, execution time, memory bandwidth, and contention: Scheduling requires execution time of a partition as input, which depends on the memory bandwidth and the worst-case memory access pattern. Memory bandwidth depends on the contention between cores and the number of memory accesses from each core, which depend on scheduling. Thus, determination of dynamic memory bandwidth and scheduling of partitions cannot be done in separate steps.

In this paper, we propose a method which solves the dynamic memory bandwidth isolation problem for time-triggered systems using a number of dynamic memory bandwidth levels. It is NoC and memory controller contention-aware and is based on the existing interference-sensitive WCET computation [22] and the memory bandwidth throttling mechanism [31], which it extends by including delay due to contention in the on-chip network and the DRAM controller as well. It constructs the schedule tables offline and assigns partitions and memory bandwidth to each slot on each core. Then, at runtime, two servers – processing time server and memory bandwidth server – run on each core, jointly controlling the contention between cores in each slot.

As a proof-of-concept, we generate schedule tables performing executing time computation and scheduling in the same step using a constraint solver. Experiments on a COTS multicore P4080, using a research OS from Airbus and EEMBC benchmarks, further demonstrate the feasibility of our proposed method.

Contributions

- We introduce a new scheduling problem for COTS multicores derived from a real avionics application – HTAWS, in which some partitions are memory-intensive.
- We present a method that solves the problem for time-triggered systems using a fixed number of dynamic memory bandwidth levels, is NoC and memory contention-aware, and based on the existing interference-sensitive WCET computation and memory bandwidth throttling mechanism.

- We show the execution time computation for a partition under dynamic memory bandwidth using worst-case memory access pattern.

Paper structure. The remainder of the paper is organised as follows: Section 2 presents the HTAWS application used in helicopters from Airbus. Section 3 presents the system model and the notation used in this work. Section 4 describes how the server-based runtime mechanism provides dynamic memory bandwidth isolation. Section 5 describes the scheduling and execution time computation for a partition under dynamic memory bandwidth using the worst-case memory access pattern, in a single step. Section 6 presents the implementation and evaluation of the proposed method. Section 7 presents the related work. Section 8 concludes the paper and also presents the future work.

2 Helicopter Terrain Awareness and Warning System Application

For future mobility, Airbus is pursuing autonomous electric aircraft targeting urban landscape to ease traffic. In addition to the avionics applications used in current aircraft, DAL-A sense-and-avoid applications for autonomous flying are needed, which still need to be developed. Nevertheless, as a starting point, a non-autonomous version of a sense-and-avoid application known as Helicopter Terrain Awareness and Warning System (HTAWS) is available today. This application is an optional feature of Airbus' helicopters.

HTAWS enables safer flying by assisting the pilot especially in degraded visual environments like flying at night, poor visibility conditions due to fog, rough terrain, and low-altitudes, useful for search and rescue mission by air ambulances and coastguards. Furthermore, it contains a map of power transmission lines and other obstacles which are hard to detect even in good weather conditions. This application is a DAL-C certified safety-critical avionics application executing on a single-core processor running VxWorks 653 RTOS from Wind River [25].

2.1 Single-core Data

HTAWS includes eight partitions with a major time frame H (MAF) of 66ms. Figure 1 shows its ARINC 653 partition-level schedule. At runtime, the inter-partition scheduler schedules the partition based on a static schedule. Table 1 shows the partition-level timing constraints. I/O operations are not part of the study here.

2.2 Measured Data on COTS Multicore

Cassidian, part of the Airbus group, performed measurements on the real HTAWS application in a test setup using a dual-core COTS P5020 platform running VxWorks with only 1 active core. A core is considered active if, in the time interval under consideration, it is allowed to issue memory accesses. Table 2 shows the maximum execution time and the maximum number of memory accesses for each partition observed in 1000 runs. Partitions π_4, π_5, and π_7, are memory-intensive.

Airbus Innovations, using its proprietary OS for research, provided the maximum observed memory latencies for a different number of active cores. Table 3 lists these values for two COTS multicore platforms: the dual-core P5020 platform and the eight-core P4080 platform.

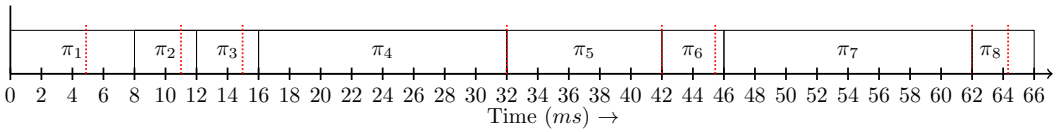


Figure 1 DAL-C certified ARINC 653 single-core schedule of the HTAWS application from Airbus. Dotted red lines indicate the maximum observed execution time of each of the partition, without contention, as shown in column 3 of Table 2.

Table 1 HTAWS application: Partition-level timing data.

Partition π_i	abs. release time r_i (ms)	abs. deadline d_i (ms)	Duration (ms)
π_1	0	8	8
π_2	8	12	4
π_3	12	16	4
π_4	16	32	16
π_5	32	42	10
π_6	42	46	4
π_7	46	62	16
π_8	62	66	4

Table 2 HTAWS application: Measurements on the dual-core COTS P5020 platform with only 1 active core.

Partition	Max. obs. num. memory accesses	Max. obs. ET (ms)
π_1	6618	4.88
π_2	2764	3.12
π_3	7381	2.97
π_4	477886	16.00
π_5	262962	10.00
π_6	4275	3.44
π_7	477886	16.00
π_8	7020	2.32

Table 3 Maximum observed memory latencies δ_j (in ns) for different number of active cores j on COTS multicores P5020 and P4080.

COTS Multicore	Mem. Lat. δ_1 (ns)	Mem. Lat. δ_2 (ns)
P5020	24.17	49.17
P4080	34.17	136.67

3 Models and Notation

This section presents the models and notation considered in this work for the COTS multicores, slots, servers, and the partitions.

3.1 COTS Multicore

We consider a COTS multicore comprising two types of hardware resources: homogeneous processing cores and a shared hardware resource consisting of a shared on-chip network and a shared DRAM sub-system including the DRAM controllers and the DRAM device.

Set $\mathcal{N}$ represents the homogeneous processing cores $\{N_1, ..., N_{|\mathcal{N}|}\}$. We consider for the shared hardware resource a set of contention-aware shared hardware resource latencies Δ, where each element δ_j denotes the maximum latency of a load/store request issued from a core under maximum contention from j active cores. A core is considered *active* if, in the time interval under consideration, it is allowed to issue memory accesses. Further, we assume that the latencies are non-decreasing with an increasing number of active cores i.e. $\frac{\delta_j}{j} \leq \frac{\delta_{j+1}}{j+1}$. This assumption allows to safely bound the time taken by a memory access accounted with e.g. δ_j latency while at runtime it may experience a lower latency δ_1 due to no contention from the $j-1$ active cores.

We assume that the COTS multicore provides a platform-level shared hardware timer and a hardware performance counter for each core to count accesses to the memory. Examples of compatible COTS multicore include Qualcomm P4080 with 8 cores [8], Qualcomm P5020 with 2 cores [23].

Figure 2 shows the hardware architecture of the 8-core COTS platform P4080 [8] from Qualcomm (initially Freescale and then NXP). On the top-left corner are the eight processing cores. The shared resource latency δ_j with j active cores includes the time taken to read/write to the DRAM device[1] for a fixed cache line size (64 bytes for P4080 and P5020 platforms) as well as the additional maximum contention delay (including arbitration time) due to j active cores in (a) the NoC – CoreNet Coherency Fabric and (b) the DRAM controller. For simplicity, we refer to the considered *shared hardware resource* as *memory* and *shared hardware resource latency* as *memory latency* in the rest of the paper.

The contention-aware memory latencies can be obtained from the hardware architecture model using static-analysis-based approach. However, Qualcomm did not provide the hardware architecture model for any of the two platforms – P4080 and P5020. An alternate approach to obtain these latencies is using measurements, described in detail in [19], previous work by Airbus Innovations (previously EADS). Table 3 and 4 show the maximum observed latencies provided by Airbus Innovations. The latencies listed in these tables for the P4080 platform have also been used in these existing works [20, 22].

3.2 Slots and Servers

We divide the timeline into fixed length time slices called slots, where slot duration of each slot S_x is $duration(S)$. It is the system designer who determines a suitable slot duration and it's time unit. For the rest of the paper, we consider slot duration in the order of ms. $\mathcal{S}$ represents the set of slots.

Our proposed method uses a server-based runtime mechanism on each core. Server τ_{sp_n} represents a processing time server on core a N_n that relates to processing resource, where

[1] Not shown in Figure 2.

Table 4 Eight-core P4080 platform: Maximum observed memory latencies δ_j (ns) for different number of active cores j.

No. act. cores j	Mem. Lat. δ_j (in ns)
1	34.17
2	136.67
3	204.17
4	385.83
5	430.83
6	614.17
7	653.33
8	839.17

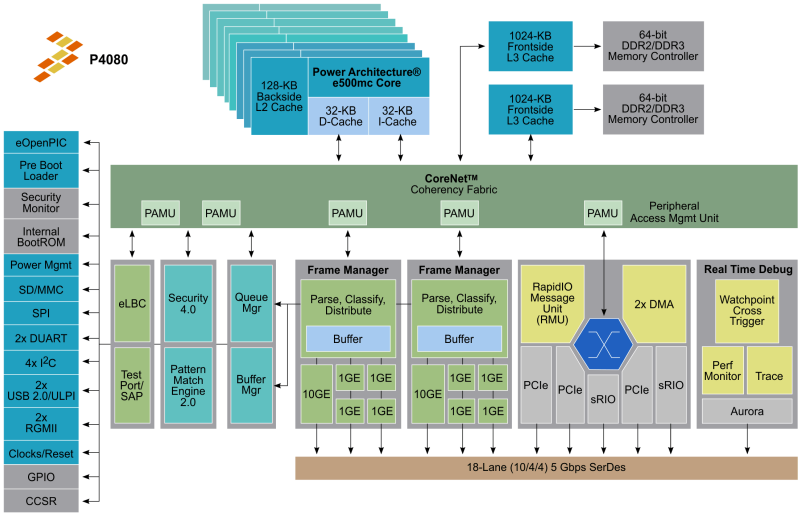


Figure 2 Architecture of the 8-core COTS platform Qualcomm P4080 [7].

sp denotes *server processing*. Server τ_{sm_n} represents the memory resource on a core N_n, where *sm* denotes *server memory*. T_{sp_n} and T_{sm_n} denote the period of each of the respective servers on a core N_n. In this work, we assume the period of all servers in the system equal to the slot duration $duration(S)$. We also assume that all servers are released synchronously on all cores.

$Q_{sp_n,x}$ (in ms) denotes the processing time server budget (simply called *processing budget*) in slot S_x on a core N_n. $Q_{sm_n,x}$ (in memory accesses) denotes the memory access server budget value (simply called *memory budget*) in slot S_x on a core N_n. Similarly, $q_{sp_n,x}$ and $q_{sm_n,x}$ denote the remaining budget during runtime for each of the corresponding servers.

3.3 Partition

The set Γ represents a set of partitions which are safety-critical i.e. their deadlines must be met. Each partition π_i is characterized by the tuple $\langle r_i, d_i, C_i^s, MA_i, C_i^m \rangle$, where

- r_i is the absolute release time,
- d_i is the absolute deadline,
- C_i^s is the core-local execution time excluding the time taken for memory accesses,

- MA_i is the maximum number of memory accesses (read/write requests) to the memory, and
- C_i^m is the multicore execution time (in slots) computed offline that includes the execution time needed for (a) C_i^s and (b) MA_i memory accesses considering possible runtime contentions from other cores, under dynamic memory bandwidth. For the rest of the paper, we use the terms – *execution time* and *multicore execution time* – interchangeably.

C_i^s and MA_i can, for instance, be acquired using a combination of static timing analysis tools like aiT and measurements as shown in [22]. We assume r_i and d_i to be integer multiples of $duration(S)$. Our proposed method is independent of the runtime memory access pattern of each partition. Thus, our partition model does not need to contain information on when and how many memory accesses a partition issues at runtime.

4 Server-Based Runtime Mechanism

Our proposed runtime mechanism provides dynamic memory bandwidth isolation, when integrated with the inter-partition scheduler. It is NoC and DRAM subsystem contention-aware and is based on MemGuard [31], an existing memory throttling mechanism .

It uses two servers per core – processing time server τ_{sp} and memory access server τ_{sm} – with a synchronised server period equal to the system-wide slot duration $duration(S)$.

4.1 Processing time server

On each core N_n a processing time server τ_{sp_n} regulates the execution time in each server instance. A processing time server budget decreases with the progression of time in a slot for each active core. During runtime, a partition on core N_n, executing in slot S_x consumes the server budget $Q_{sp_n,x}$ for the core-local execution time on core N_n and the time taken for memory accesses under contention from other cores. Due to the dynamic memory bandwidth, the processing budget may differ in distinct slots.

4.2 Memory Access Server

On each core N_n, a memory access server τ_{sm_n} regulates the total number of memory accesses in each server instance. Due to the dynamic memory bandwidth, the memory budget may differ in distinct slots. At runtime, an executing partition π_i on core N_n in slot S_x uses the memory budget $Q_{sm_n,x}$ only for memory accesses. Each access results in a decrease of remaining server budget by 1.

4.3 Runtime Behaviour

During runtime, each inter-partition-level scheduler, at the start of each slot S_x, sets the corresponding processing and memory budgets for each server based on the schedule table found in the offline phase. The server budgets need not be same in each slot. The processing time server budget decreases with the progression of time in a slot for each active core. The memory access server budget decreases by 1 on each memory access issued by an executing partition in the corresponding slot. A partition continues to execute in a slot S_x on a core N_n while both the servers have budgets greater than 0, i.e. $q_{sm_n,x} > 0 \wedge q_{sp_n,x} > 0$. If in a slot S_x on a core N_n any of the two servers exhausts its budget, the partition is stalled until the next server instance and the core is idle. If any of the two servers have a remaining server budget, it is discarded.

Jointly, the two servers on each core guarantee that the servers budgets provided for each slot in the offline schedule table, hold at runtime. This enables contention-aware dynamic memory bandwidth isolation between cores.

5 Scheduling for Dynamic Memory Bandwidth

Static memory bandwidth isolation mechanisms assign a constant amount of memory bandwidth to each core before runtime. This limits the worst-case number of contentions any partition on a core can experience at any time, which allows computation of execution time separately from scheduling. However, under dynamic memory bandwidth isolation, the execution time of a partition depends on (a) the time taken for memory accesses which depends on the contention from the other cores, and (b) its worst-case memory access pattern, which depends on the dynamic memory bandwidth assigned in each slot during its execution. As scheduling decides the amount of contention between cores which can vary between slots due to dynamic memory bandwidth, we cannot perform execution time computation and scheduling separately.

In the next sections, we show how to resolve each of the two dependencies of execution time – contention and memory access pattern. Later, we show how to perform scheduling and execution time computation together for a general case under dynamic memory bandwidth. Finally, we show an example with dynamic memory bandwidth for 2-cores.

5.1 Resolving Dependency – Contention

Under dynamic memory bandwidth, resolving contention dependency for execution time computation of a partition on a core requires knowledge of the number of active cores in each slot and the maximum contentions that can be introduced by each active core in each slot. In the next section, we describe a way to resolve this dependency considering $|\mathcal{N}| + 1$ dynamic bandwidth levels. Later, we show how to consider an arbitrary number of dynamic bandwidth levels.

5.1.1 $|\mathcal{N}| + 1$ Dynamic Bandwidth Levels

Instead of a constant memory budget for each core, we consider $|\mathcal{N}| + 1$ dynamic memory bandwidth levels. For simplicity, in each level, we divide the memory bandwidth equally between the active cores j. Each level corresponds to a memory budget Q^j_{sm}, which associates with a memory latency δ_j, providing contention-awareness to the memory bandwidth. It allows estimating the contention between cores in a slot just by knowing the number of active cores in that slot, without requiring knowledge of which partitions are scheduled on the other cores, and their exact number of memory accesses. Thus, for each memory access server instance $\tau_{sm_n,x}$ on a core N_n, there are $|\mathcal{N}| + 1$ possible budgets. For each processing time server instance $\tau_{sp_n,x}$ we consider only two fixed budgets: $Q^0_{sp} = 0$ and $Q^1_{sp} = X$. X is a fixed value determined by a system designer such that $0 < X \leq \mathit{duration}(S)$.

Equation (1) computes the different memory budgets based on the existing interference-sensitive WCET computation [22, 20] with equal memory budget distribution between the active cores in a slot. It also shows the relationship between the two servers and the memory latencies.

$$\forall \delta_j \in \Delta \ , \ Q^j_{sm} = \left\lfloor \frac{Q^1_{sp}}{\delta_j} \right\rfloor \tag{1}$$

Table 5 Per core memory access server budgets Q^j_{sm} with equal distribution of accesses corresponding to memory latencies from Table 3 for processing time server budget Q^1_{sp} of $duration(S) = 1ms$.

COTS Multicore	Mem. bud. Q^0_{sm}	Mem. bud. Q^1_{sm}	Mem. bud. Q^2_{sm}
P5020	0	41379	20338
P4080	0	29268	7317

E.g., consider a $duration(S)$ of 1ms and a processing budget Q^1_{sp} equals to $duration(S)$. Then, Table 5 shows the memory budgets Q^j_{sm} considering the memory latencies from Table 3. When a core is not active in a slot, we assume its memory budget and processing budget equal to 0. For computation purposes, we assume the memory latency δ_0 equals ∞. Thus, the use of dynamic memory bandwidth levels limits the maximum contentions each core can experience in each slot from the other cores.

5.1.2 Arbitrary Number of Dynamic Bandwidth Levels

Our proposed method also works for an arbitrary number of dynamic memory bandwidth levels and unequal distribution between active cores. A set $\mathcal{Q}_{sm} = \bigcup_{j=1}^{|\mathcal{N}|} \hat{Q}^j_{sm}$, where j is the number of active cores, represents memory budget distributions. A set $\hat{Q}^j_{sm}$ with j active cores, represents distinct valid memory budget distributions between the j active cores. For all the remaining cores, the assigned memory budget is 0. A memory budget distribution, sorted in increasing order of memory budgets, $\{y_1, ..., y_j\}$ for j active cores, is valid if the condition in Equation 2 holds.

$$\sum_{k=1}^{j} (y_k - y_{k-1}) * \delta_{j-k+1} \leq Q^1_{sp} \ , \text{ where } y_0 = 0 \, . \tag{2}$$

This is based on the interference-sensitive WCET computation [20, 22] and means that, in a slot, only a maximum of y_1 memory accesses from each active core will experience memory latency δ_j, $y_2 - y_1$ will experience latency δ_{j-1} and so on.

In slot S_x on an active core N_n with $j-1$ active cores, an offline scheduler can assign any memory budget from a valid budget distribution $\{y_1, ..., y_j\}$ to a memory access server instance $\tau_{sm_n,x}$ that has not been assigned to memory access servers of $j-1$ active cores. For example, a set $\hat{Q}^2_{sm}$ can contain $\{7000, 34137\}$ as a valid memory budget distribution for 2 active cores using P5020 memory latencies (row 2 of Table 3). If, in a slot S_2 on core N_1 an offline scheduler assigns a memory access server a budget of 7000, then the memory access server of the second other active core in slot S_2 will be assigned a budget of 34137. Another valid memory budget distribution for 2 active cores using P5020 memory latencies is $\{20338, 20338\}$ as listed in Table 5 (row 2 column 4). The set $\mathcal{Q}_{sm}$ is either provided as input to the offline scheduler by the system designer or generated by an offline scheduler based on the properties of an application.

For the rest of Section 5, to simplify the description, we limit to $|\mathcal{N}| + 1$ dynamic bandwidth levels with equal distribution of memory budgets between the active cores as described in Section 5.1.1. However, note that the ensuing description is still applicable to arbitrary bandwidth levels.

Table 6 Example slots to depict the interaction between memory access patterns and dynamic memory access server budgets.

Slot S_x	S_1	S_2	S_3	$S_{1'}$	$S_{2'}$	$S_{3'}$
Mem. ser. bud. $Q_{sm_1,x}$	45	100	15	100	45	15
Proc. ser. bud. $Q_{sp_1,x}$	1	1	1	1	1	1

5.2 Resolving Dependency – Memory Access Pattern and Dynamic Bandwidth

Static memory bandwidth isolations mechanisms assign same memory budget in each server instance on a core. Such mechanisms are agnostic of the number of active cores in each slot and need to consider the worst-case memory latency corresponding to a maximum number of active cores. Therefore, the use of a static memory budget per core considering worst-case memory latency ensures that if a partition performs a memory access in any of its assigned slots, a partition's execution time does not increase. However, under dynamic memory bandwidth, a partition may receive different memory budgets in each slot. Further, since the latencies of memory requests may differ between different memory budgets, the interaction between memory budgets and a partition's memory access pattern impacts a partition's execution time. The example in the next section highlights this interaction.

5.2.1 Example

Consider a partition π_1: $\langle r_1 = 0, d_1 = 3, C_1^s = 1, MA_1 = 60\rangle$.

An offline scheduler identifies three slots for π_1 on core 1. Let us consider two illustrative runtime memory access patterns of π_1 considering memory budgets and processing budgets as shown in Table 6 (columns 2,3 and 4):

1. If this partition uses slot S_1 for its core-local execution time C_1^s, and slot S_2 for its $MA_1 = 60$ memory accesses, then slots $\{S_1, S_2\}$ are sufficient to meet its C_1^s and MA_1 requirements.
2. On the other hand, if π_1 uses slot S_1 for 45 memory accesses, slot S_2 for its C_1^s, and slot S_3 for the remaining 15 memory accesses, then π_1 requires slots $\{S_1, S_2, S_3\}$ to meet its requirements.

If the memory budgets of the three slots were different as shown in columns 5,6 and 7 in Table 6, then using the previously considered memory access pattern 2, π_1 may use slot S_1 for 45 memory accesses and the remaining 0.55 time units in the slot for a part of its C_1^s, slot S_2 for its remaining 0.45 time units of C_1^s with the remaining slot for its 15 memory accesses. Thus, it just requires 2 slots. Similarly, using previously considered memory access pattern 1, the partition will need three slots.

Our proposed method is independent of the runtime memory access patterns of a partition as we consider the worst-case memory access pattern of a partition for assigned memory budgets in each slot.

5.2.2 Worst-case Memory Access Pattern

The example in the previous section illustrated the need to construct a worst-case memory access pattern for each partition to ensure the slots with dynamic memory budgets under consideration will meet the partition's requirements. Static WCET analysis tools in combination with measurements can help to find a worst-case memory access pattern. However,

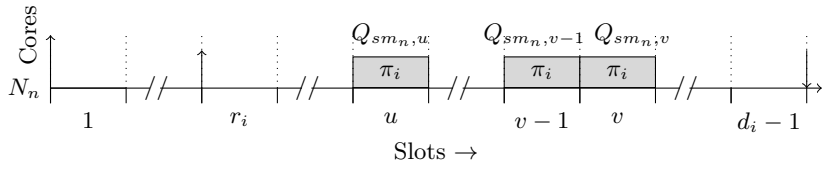


Figure 3 General case when an offline scheduler considers slots $\forall S_x \in [u, v]$ on some core N_n for partition π_i with possibly different memory budgets.

pragmatically such an approach is infeasible due to the computational complexity involved. Further such methods need to explore all valid inputs. To overcome these issues, our method uses a worst-case memory access pattern that only requires information about the partition model and the memory budget assigned to it in each slot.

The worst-case memory access pattern for a partition π_i assigned to execute on N_n in slots S_x $\forall x \in [u, v]$ manifests when a partition π_i uses $\frac{C_i^s}{Q_{sp}^1}$ slots $\in [u, v]$ with the largest assigned memory budgets for its core-local execution time C_i^s requirement, and the remaining slots for its MA_i memory access requirement. Figure 3 shows this general case with exemplary dynamic memory access server budgets for slots S_u, S_{v-1} and S_v.

Note, that the worst-case memory access pattern may not manifest at runtime. Nevertheless, considering it allows an offline scheduler to provide execution time guarantees, irrespective of the runtime memory access patterns. In the next section, we present a combined scheduling and execution time computation step.

5.3 Combined Scheduling and Execution Time

This section describes how scheduling and execution time computation are performed together.

When scheduling offline, consider a general case as shown in Figure 3 with a partition π_i: $\langle r_i, d_i, C_i^s, MA_i, C_i^m \rangle$, assigned to a core N_n. An offline scheduler considers the slots $S_x \in [u, v]$ to assign to π_i and needs to check if these slots will meet C_i^s and MA_i requirements of a partition π_i. For each slot S_x, each memory access server's budget $Q_{sm_n,x}$ relates to a valid memory budget distribution described in Section 5.1.

The offline scheduler must check if the slots S_x $\forall x \in [u, v]$ on core N_n are sufficient to meet the C_i^s and MA_i requirements of a π_i partition. Using the worst-case memory access pattern described in Section 5.2.2, the offline scheduler sorts and renumbers the slots S_x $\forall x \in [u, v]$ in descending order according to their memory budget $Q_{sm_n,x}$. $[u'v']$ represents the sorted and renumbered order of slots. Then the offline scheduler determines the number of slots needed to meet the core-local execution time requirement of π_i given by the Equation (3).

$$\mu = \frac{C_i^s}{Q_{sp}^1} \tag{3}$$

It may happen that μ is not an integer, so the offline scheduler computes the amount of memory accesses possible in the remaining part of slot $S_{u'+\lceil\mu\rceil}$ i.e.

$$\rho = \left\lceil (\lceil\mu\rceil - \mu) * Q_{sm_n,u'+\lceil\mu\rceil} \right\rceil \ .$$

Then the offline scheduler sums the memory budgets of the remaining slots from $[u' + \lceil\mu\rceil, v']$ i.e.

$$\forall x' \in [u' + \lceil\mu\rceil, v'] \ , \ \psi = \sum Q_{sm_n,x'} \ .$$

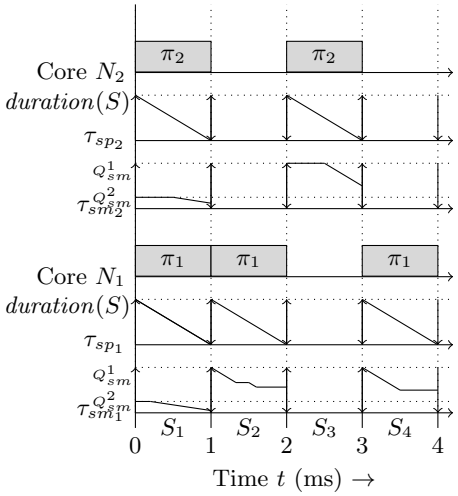


Figure 4 Illustrative example of our proposed method with three dynamic bandwidth levels.

If $MA_i \leq \rho + \psi$, then the slots $\forall S_x \in [u, v]$ will meet the partition's requirements. Thus, using the valid memory budget distributions and the worst-case memory access pattern allows scheduling and execution time computation in a single step for dynamic memory bandwidth.

5.4 Example

Figure 4 shows an illustrative example of our proposed method for 2-cores using memory latencies for P4080 platform (row 3 in Table 3). Each core N_n has two servers: processing time server τ_{sp_n} and memory access server τ_{sm_n}, with the period of each server is equal to the slot duration of 1ms. We consider 3 different dynamic bandwidth levels as described in Section 5.1.1, with valid budget distributions shown in row 3 of Table 5. The two levels relate to valid budget distributions $\{\{29268\}, \{7317, 7317\}\}$. The third level simply assigns each inactive core a 0 memory budget. For each server, the dotted horizontal lines depict the possible server budgets. During runtime, at the start of each slot, each inter-partition scheduler sets the corresponding server budgets for each server on its respective core, based on the offline schedule table (shown via budgets and partition assignment in Figure 4).

In slot S_1, i.e. at $t = 0$, both the cores are active and each inter-partition scheduler sets the corresponding memory budget $Q^2_{sm} = 7317$ and processing budget to 1ms. At the start of slot S_2 (at time $t = 1$ms), only partition π_1 is active and is assigned a memory budget of $Q_{sm_{1,2}} = Q^1_{sm} = 29268$ and processing budget of 1ms. In the time interval $[1, 1.33)$ms, the partition π_1 issues memory accesses as indicated by the corresponding decrease in remaining memory budget. In the time interval $[1.33, 1.5)$ms, π_1 does not perform any memory accesses as indicated by the memory budget being constant. Later, π_1 again briefly issues memory accesses for the next 100μs. In the time interval $[1.6, 2)$ms, since only the processing budget decreases and not the memory budget, the partition π_1 only performs computations.

At the end of slot S_3, the partition π_2 completes execution and the inter-partition scheduler of core N_2 discards the unused memory budget of the server instance $\tau_{sm_{1,3}}$. In slot S_4 (at time $t = 3$ms), as only core N_1 is active, the memory budget $Q_{sm_{1,4}}$ equals Q^1_{sm}. The partition π_1 issues memory accesses in the first half of the slot as indicated by the decrease in the remaining memory budget. Then, for the next 200μs, the partition π_1 does not issue any

memory accesses as the memory budget does not decrease and at time $t = 4$ms, π_1 completes execution.

6 Experimental Evaluation

Section 6.1 describes a proof-of-concept schedule table generation using a constraint solver. Section 6.2 presents the integration of our proposed runtime mechanism in a proprietary research OS for Qualcomm P4080 COTS multicore platform and the evaluation of our proposed method using EEMBC AutoBench benchmarks for a number of dynamic memory bandwidth levels.

6.1 Schedule Tables Generation

As a proof-of-concept, we modelled the partition allocation and scheduling problem for the Gecode [10] constraint programming (CP) solver, to find a valid schedule table for each of the two cores. The specified constraints schedule the HTAWS application on 1 core and additional partitions on the other core under dynamic memory bandwidth, while preserving the single-core HTAWS schedule.

6.1.1 Preparation of the HTAWS Application Data

The numbers shown in Table 2 Section 2 are the sole input to the study presented here as actual application internals are confidential. They give maximum observed execution times and memory accesses for each partition but do not mention the distribution of read/write requests as well as how many requests end in L3 cache. Our partition model needs the core-local execution times, which are not given. Thus, the observed data had to be "reverse engineered" for this study, making safe assumptions about the number of memory accesses and core-local execution time of each partition. We use the memory latencies shown in Table 3 to obtain core-local execution times shown in column 2 of Table 7. Partition π_5 requires on average, 73% of memory bandwidth using δ_1 memory latency. Column 3 shows the minimum constant memory bandwidth that needs to be reserved using existing interference-sensitive WCET computation [20, 22], to meet each partition's C_i^s and MA_i requirements with only 1 active core. The memory latencies are the largest observed latencies for the different number of active cores under different combinations of read and write memory requests. The execution time computation further assumes that the core stalls on each memory request. As a consequence, the processed data is pessimistic, calling for more detailed information available from the original application. The processed data is, however, safe for the experiments in this study.

6.1.2 Application and Schedule Table

As no real avionics application with more than one active core at the same time exists today, we construct a scenario by replicating some partitions such that multiple cores are active at the same time. We obtained schedule tables using the Gecode constraint solver with a model specified using our proposed method. Figure 5 shows one such schedule found by the Gecode solver in which partitions π_1, π_2, and π_8 are replicated on the second core. Appendix A lists the formulation of the key constraint that checks if the slots under consideration meet a partition's π_i requirements of C_i^s and MA_i as described in the Section 5.3 using a number of bandwidth levels with equal memory budget distribution. It demonstrates the feasibility of

Table 7 "Reverse-engineered" core-local execution time and minimum constant memory bandwidth that needs to be using reserved using existing interference-sensitive WCET computation [20, 22] for the HTAWS application.

Partition	Computed core-local ET C_i^s (ms)	Min. mem. b/w (in %) reqd. throughout partition considering δ_1 latency
π_1	4.72	4.88
π_2	3.05	7.03
π_3	2.79	14.74
π_4	4.45	100.00
π_5	3.64	100.00
π_6	3.34	15.66
π_7	4.45	100.00
π_8	2.15	9.17

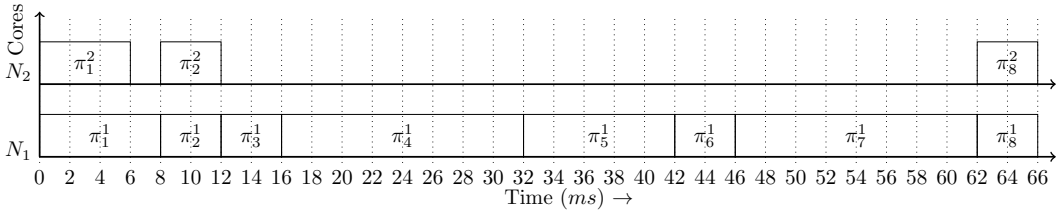


Figure 5 Partition schedule generated through Gecode constraint solver using our proposed method for the HTAWS application with some replicated partitions, such that existing HTAWS schedule is preserved.

scheduling and computing execution time offline, in the same step, under dynamic memory bandwidth, overcoming the inter-dependency challenge. At runtime, our proposed server-based runtime mechanism, described in Section 4, will execute the offline-computed scheduling decisions including assigning of the server budgets on each core in each slot.

Complexity. Slot duration plays a major role, for a larger slot duration reduces the length of the major time frame H (in slots), thereby restricting the search space. Further, the search space also depends on the number of cores, the number of partitions, and the different server budgets allowed. Due to the computational complexity of the problem, in the future, we plan to integrate our proposed method in our in-house heuristic-based offline scheduler.

6.2 Execution Time

We use the Qualcomm P4080 platform with eight e500mc cores. It is widely used by various groups conducting academic and industrial research.

6.2.1 Integration of Runtime Mechanism

We integrated our proposed runtime mechanism in a proprietary OS for research by Airbus that runs on the P4080 platform. For each core's processing time server, we use the multicore programmable interrupt controller (MPIC) timer, a hardware timer with auto-reload feature. It ensures slot-level synchronisation amongst all cores and provides a unique interrupt

Table 8 Number of memory accesses MA_i for EEMBC benchmarks obtained using configuration 1. Core-local execution time C_i^s is obtained by a "reverse-engineering" step.

Benchmark	MA_i	C_i^s (ms)
cacheb	1725	5.39
a2time	659	13.35
rspeed	591	0.621

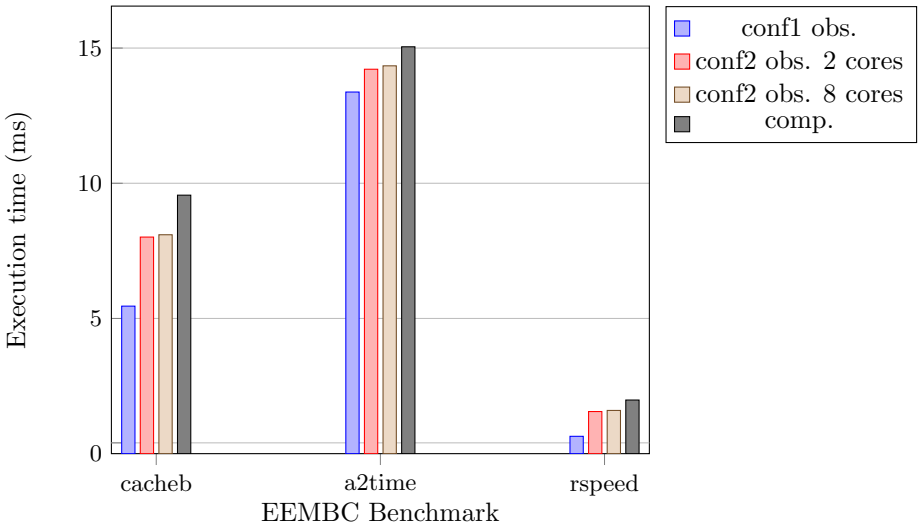


Figure 6 Execution time for EEMBC benchmarks: maximum observed values in configuration 1, maximum observed values in configuration 2 using our proposed method for 2 active cores and 8 active cores, and the computed execution time using our method based on the budgets assigned per slot in configuration 2.

instance [7] to each core on each interrupt. For the memory access server, we use the core-level hardware performance counter that counts the requests to the on-chip network [7] from each core.

6.2.2 Testbed Setup

In our experiments, we configure the platform clock frequency to 600 MHz and each core's clock frequency to 1200 MHz. The hardware timers used: MPIC timer, and Timebase timer, are configured to 37.5 MHz. The Timebase timer is used to generate timestamps for the benchmark under test. L1 and L2 caches are enabled, while L3 caches are disabled. We use both the available memory controllers.

6.2.3 Experiments

We ran benchmarks from the EEMBC AutoBench benchmark suite [28] in two different configurations for 10 runs each. Configuration 1 runs the benchmark in isolation with no memory throttling to obtain largest observed execution times and number of memory accesses. This configuration is required to obtain core-local execution time for each benchmark. Configuration 2 integrates our proposed server-based mechanism with $duration(S) = Q_{sp}^1 = 1ms$ and runs each of the benchmark on one core and a contending benchmark (matrix) replicated on the rest of the active cores. The memory budgets for the core executing the

benchmark under test are generated randomly for each of the 10 slots which are repeated in a schedule table. Thus, the schedule table contains 10 dynamic memory bandwidth levels. The memory budgets for the other active cores in each slot relate to the first core's memory budgets computed using Equation (2) with equal budget assignment. Table 8 shows the maximum number of memory accesses observed for each benchmark using configuration 1. This data is used to obtain core-local execution times by "reverse-engineering" using memory latencies from Table 3 (row 3). Figure 6 shows the largest observed execution time in configuration 1, in configuration 2 with 2 active cores and 8 active cores, and the computed execution time using our method based on the budgets assigned per slot in configuration 2. It shows that the observed execution times are less than or equal to the computed execution times using our method. We also observed per stall overhead due to the processing time server and memory access server of 10μs which is 1% of the considered slot duration of 1ms [21]. All measured execution times shown in Figure 6 exclude these stall overheads.

7 Related Work

Over the last years, several different scheduling approaches for COTS multicores have been presented. Schranzhofer et al. [26], Pellizzoni et al. [24], Boniol et al. [4] proposed deterministic execution models to control the access to shared resources. The basic concept is to divide program execution into multiple phases and restrict their capabilities. Schranzhofer et al. [26] proposed to divide each task into superblocks where each superblock consists of three phases: acquisition phase, execution phase and replication phase. A task is allowed to access the shared hardware resource only in acquisition and replication phases. Further, no two co-executing tasks on different cores can have overlapping acquisition and replication phases. Pellizzoni et al. [24] proposed the PRedictable Execution Model (PREM) for single-core COTS processors, introducing co-scheduling for shared resources. It splits each task into a sequence of non-preemptable intervals: predictable intervals, and compatible intervals. Predictable intervals are used to pre-load all data and instructions into local caches, while system calls and interrupt preemptions are prohibited. System calls and interrupt preemptions are, however, allowed in compatible intervals. Each predictable interval is further divided into two phases: execution phase and memory phase. Traffic from peripheral devices is only permitted during the execution phase of a predictable interval, resulting in an architecture with very few contentions for accesses to the shared resources. Boniol et al. [4] presented a sliced execution model. It splits each task into sub-tasks and each sub-task further into execution slices and communication slices. The access to a shared resource is only allowed in a communication slice. Such approaches are known to poorly utilise the respective resource [15]. In addition, to support transition from single-core processors to COTS multicore processors, such approaches require modification of the source code of legacy applications [11], which is a non-trivial step.

Another popular approach is joint analysis. To address sharing of resources those approaches analyse the program flows on all cores using the considered shared resource. Therefore, detailed knowledge of the state of execution is required. Yan and Wang [29] applied WCET analysis to multicore processors with shared L2 caches by analysing inter-thread dependencies. The analysis is based on the program control flow and accounts for all possible conflicts on the shared cache. Li et al. [16] extended the analysis by identifying possibly overlapping threads. Hardy et al. [13] further extended it by reducing the number of possible conflicts between overlapping threads. Chattopadhyay et al. [6, 5] and Kelter et al. [14] proposed combined shared cache and shared bus analysis and applied a TDMA bus

arbitration. Chattopadhyay et al. [5] combine cache and bus analysis with other architectural features such as pipelines and branch prediction. Overall, the joint analysis approaches have to explore huge state spaces due to the various possible interactions between different tasks, resulting in huge computational complexity.

With respect to monitoring, Bellosa [3, 2] introduced the idea to leverage built-in processor counters to acquire additional task runtime information. Yun et al. [32] proposed controlling memory accesses from all but one cores to limit contentions experienced by hard real-time tasks executing on one core to ensure that they meet their deadlines. Yun et al. extended the work in [31] by introducing a memory throttling mechanism – MemGuard, that regulates memory accesses using a memory server on each core. It assumes that memory bandwidth is statically partitioned between cores before runtime for safety-critical systems. Behnam et al. [1] propose a method to isolate the behaviour of different cores. They apply a hierarchy of servers to all cores using a server-based approach which assigns a certain limit on the amount of cache misses. Yao et al. [30] present a method to bound variability in execution time of each task on all cores considering round-robin arbitration between cores for memory accesses. Mancuso et. al. [18] present a method to compute the WCET under static partitioning of memory bandwidth between cores, which takes into account the amount of locked data in caches. The main difference of our proposed method against these works is our focus on dynamic memory bandwidth isolation and guarantees for safety-critical systems. Also, the goal of preserving static schedule on a core differentiates this paper from other server-based approaches. Moreover, most of these works do not consider additional arbitration and contention delay introduced by the shared on-chip network and the DRAM memory controller in the analysis.

The authors in [20, 22] consider a process frame that consists of co-executing tasks and introduced the interference-sensitive WCET (is-WCET) computation that computes, in offline phase, the execution time of each process considering contentions from the worst-case number of memory accesses from each of co-executing processes in a process-frame. Our proposed method is based on the same is-WCET computation, but allows a finer granularity of resource control (slots) instead of partition level. This enables our method to further relax the strong start time constraints imposed by [20, 22], that is, a new process can only be scheduled after the completion of all processes in a process frame.

8 Conclusion and Future Work

In this paper, we presented a method for a step in the investigation of Airbus using COTS multicores for safety-critical avionics applications. It addresses the need of preserving the ARINC 653 single core schedule of a Helicopter Terrain Awareness and Warning System application while scheduling additional safety-critical partitions on the other cores. As some partitions are memory-intensive, dynamic memory bandwidth isolation is needed, which requires performing the computation of execution times and scheduling together.

Our method solves this problem for slot-based time-triggered systems using a number of dynamic memory bandwidth levels. The method is NoC and DRAM controller contention-aware and is based on the existing interference-sensitive WCET computation and memory bandwidth throttling mechanisms and does not require application source-code modifications. It constructs schedule tables assigning partitions and dynamic memory bandwidth to each slot on each core. Then at runtime, two servers – for processing time and memory bandwidth – run on each core, jointly controlling the contention between the cores and the amount of memory accesses per slot. Thus, the number of active cores can vary over time.

As a proof of concept, we generated schedule tables performing executing time computation and scheduling in the same step using a constraint solver. The basic concepts can be included in a variety of offline algorithms for schedule table construction. We considered a generic case for execution time computation with scheduling that can be used, e.g., in search based algorithms.

We implemented our proposed runtime mechanism part in a proprietary research operating system from Airbus and EEMBC benchmarks, executed on a Qualcomm P4080 multicore platform, demonstrating its practicality. We evaluated our runtime mechanism and execution time computation under dynamic memory bandwidth levels and showed that the observed execution times are less than or equal to the computed execution times.

For future work, we will turn the focus away from the specific application and hardware settings in this paper to generalise our method. Instead of the proof-of-concept constraint solver used here, we will extend our in-house heuristic-based offline scheduling framework to integrate the proposed offline phase. This will allow us to focus on the complexity performance of the offline part of the method and its integration in existing tool chains.

Acknowledgements. We are grateful to Bernd Koppenhöfer and Max Gapp from Cassidian for the case study. We also thank the anonymous reviewers for their valuable feedback. We would also like to express our gratitude to Björn Brandenburg and the members of the RTS group at TUK for their helpful suggestions. We also thank Claire Pagetti, Daniel Gracia Pérez, and Rob Davis for the enriching discussions.

References

1 Moris Behnam, Rafia Inam, Thomas Nolte, and Mikael Sjödin. Multi-core composability in the face of memory-bus contention. *SIGBED Rev.*, 10(3):35–42, October 2013. doi:10.1145/2544350.2544354.

2 Frank Bellosa. Memory access – the third dimension of scheduling. Technical report, University of Erlangen, 1997.

3 Frank Bellosa. Process cruise control: Throttling memory access in a soft real-time environment. Technical report, University of Erlangen, 1997.

4 Frédéric Boniol, Hugues Cassé, Eric Noulard, and Claire Pagetti. Deterministic Execution Model on COTS Hardware. In Andreas Herkersdorf, Kay Römer, and Uwe Brinkschulte, editors, *Architecture of Computing Systems – ARCS 2012: 25th International Conference, Munich, Germany, February 28 – March 2, 2012. Proceedings*, pages 98–110, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-28293-5_9.

5 S. Chattopadhyay, C. L. Kee, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk. A Unified WCET Analysis Framework for Multi-core Platforms. In *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, pages 99–108, April 2012. doi:10.1109/RTAS.2012.26.

6 Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. Modeling Shared Cache and Bus in Multi-cores for Timing Analysis. In *Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems*, SCOPES'10, pages 6:1–6:10, New York, NY, USA, 2010. ACM. doi:10.1145/1811212.1811220.

7 Freescale Semiconductor. *e500mc Core Reference Manual Rev. 3*, 2013.

8 Inc. Freescale Semiconductor. P4080: QorIQ P4080/P4040/P4081 Communications Processors with Data Path, 2013. URL: http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=P4080.

9 R. Fuchsen. How to address certification for multi-core based IMA platforms: Current status and potential solutions. In *29th Digital Avionics Systems Conference*, pages 5.E.3–1–5.E.3–11, Oct 2010. doi:10.1109/DASC.2010.5655461.

10 Generic constraint development environment. URL: http://www.gecode.org/.

11 S. Girbal, X. Jean, J. Le Rhun, D. G. Perez, and M. Gatti. Deterministic platform software for hard real-time systems using multi-core COTS. In *Digital Avionics Systems Conference (DASC), 2015 IEEE/AIAA 34th*, pages 8D4–1–8D4–15, Sept 2015. doi:10.1109/DASC.2015.7311481.

12 Airbus Group. Future of urban mobility: My kind of flyover. URL: http://www.airbusgroup.com/int/en/news-media/corporate-magazine/Forum-88/My-Kind-Of-Flyover.html.

13 D. Hardy, T. Piquet, and I. Puaut. Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches. In *2009 30th IEEE Real-Time Systems Symposium*, pages 68–77, Dec 2009. doi:10.1109/RTSS.2009.34.

14 T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. Bus-Aware Multicore WCET Analysis through TDMA Offset Bounds. In *2011 23rd Euromicro Conference on Real-Time Systems*, pages 3–12, July 2011. doi:10.1109/ECRTS.2011.9.

15 Timon Kelter, Tim Harde, Peter Marwedel, and Heiko Falk. Evaluation of resource arbitration methods for multi-core real-time systems. In Claire Maiza, editor, *13th International Workshop on Worst-Case Execution Time Analysis*, volume 30 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–10, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2013/4117, doi:10.4230/OASIcs.WCET.2013.1.

16 Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *2009 30th IEEE Real-Time Systems Symposium*, pages 57–67, Dec 2009. doi:10.1109/RTSS.2009.32.

17 Rodin Lyasoff. Welcome to Vahana, Sept. 2016. URL: https://vahana.aero/welcome-to-vahana-edfa689f2b75#.fz78tkigh.

18 R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun. WCET(m) Estimation in Multi-core Systems Using Single Core Equivalenc. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 174–183, July 2015. doi:10.1109/ECRTS.2015.23.

19 J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *Dependable Computing Conference (EDCC), 2012 Ninth European*, pages 132–143, May 2012. doi:10.1109/EDCC.2012.27.

20 J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement. In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, pages 109–118, July 2014. doi:10.1109/ECRTS.2014.20.

21 Jan Nowotsch. *Interference-sensitive Worst-case Execution Time Analysis for Multi-core Processors*. PhD thesis, University of Augsburg, 2014.

22 Jan Nowotsch and Michael Paulitsch. Quality of service capabilities for hard real-time applications on multi-core processors. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, RTNS'13, pages 151–160, New York, NY, USA, 2013. ACM. doi:10.1145/2516821.2516826.

23 NXP. P5020: QorIQ® P5020 and P5010 64-bit Dual- and Single-Core Communications Processors, Sept. 2016. URL: http://www.nxp.com/products/microcontrollers-and-processors/power-architecture-processors/qoriq-platforms/p-series.

24 R. Pellizzoni, E. Betti, S. Bak, Gang Yao, J. Criswell, M. Caccamo, and R. Kegley. A Predictable Execution Model for COTS-Based Embedded Systems. In *Real-Time and*

Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE, pages 269–279, April 2011. doi:10.1109/RTAS.2011.33.

25 Wind River. Wind River VxWorks 653 Platform, 2015. URL: http://www.windriver.com/products/product-overviews/vxworks-653-product-overview/vxworks-653-product-overview.pdf.

26 A. Schranzhofer, R. Pellizzoni, Jian-Jia Chen, L. Thiele, and M. Caccamo. Timing analysis for resource access interference on adaptive resource arbiters. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 213–222, April 2011. doi:10.1109/RTAS.2011.28.

27 Certification Authorities Software Team. *Position Paper CAST-32A Multi-core Processors*. US Federal Aviation Administration, Nov. 2016.

28 Inc. The Embedded Microprocessor Benchmark Consortium. EEMBC AutoBench 1.1 benchmark software, 2013. URL: http://www.eembc.org/benchmark/automotivesl.php.

29 J. Yan and W. Zhang. WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches. In *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 80–89, April 2008. doi:10.1109/RTAS.2008.6.

30 Gang Yao, Heechul Yun, Zheng Pei Wu, R. Pellizzoni, M. Caccamo, and Lui Sha. Schedulability analysis for memory bandwidth regulated multicore real-time systems. *Computers, IEEE Transactions on*, 65(2):601–614, Feb 2016. doi:10.1109/TC.2015.2425874.

31 H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Transactions on Computers*, 65(2):562–576, Feb 2016. doi:10.1109/TC.2015.2425889.

32 Heechul Yun, Gang Yao, R. Pellizzoni, M. Caccamo, and Lui Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 299–308, July 2012. doi:10.1109/ECRTS.2012.32.

A Core-local Execution Time and Memory Accesses Constraints

Constraint (4), specified in our constraint-solver model, is a key constraint that ensures that the number of slots allocated to a partition meet its requirements of core-local execution time and the number of memory accesses, under fixed number of dynamic memory bandwidth levels with equal budget distribution between the active cores in each slot.

$slot1(i)$ represents the number of slots with memory budget $Q^1_{sm} = 41379$ considering 1 active core assigned to a partition π_i. $slot2(i)$ represents the number of slots with memory budget $Q^2_{sm} = 20338$ considering 2 active core assigned to a partition π_i.

$$
\begin{aligned}
&\forall i \in \Gamma\ , \\
&\left(slot1(i) == 0\ \wedge\ slot2(i) == \left\lceil \frac{C^s_i}{Q^1_{sp}} + \frac{MA_i}{Q^2_{sm}} \right\rceil\right) \\
\vee &\left(slot1(i) == \left\lceil \frac{C^s_i}{Q^1_{sp}} + \frac{MA_i}{Q^1_{sm}} \right\rceil\ \wedge\ slot2(i) == 0\right) \\
\\
&\quad (slot1(i) > 0\ \wedge\ slot2(i) > 0) \\
\vee &\quad \wedge \left(slot1(i) \geq C^s_i\ \wedge\ part(i) == slot1(i) * Q^1_{sm} - C^s_i * Q^1_{sm}\right) \\
&\quad \wedge \left(slot2(i) * Q^2_{sm} + part(i) - MA_i < Q^2_{sm}\right) \\
&\quad \wedge \left(slot2(i) * Q^2_{sm} + part(i) - MA_i \geq 0\right) \\
\\
&\quad slot1(i) < C^s_k \\
&\quad \wedge \left(part(i) == -slot1(i) * Q^1_{sm} + C^s_i * Q^1_{sm}\right) \\
\vee &\quad \wedge \left(slot2(i) + \frac{part(i)}{Q^1_{sm}}\right) * Q^2_{sm} - MA_i < Q^2_{sm} \\
&\quad \wedge \left(slot2(i) + \frac{part(i)}{Q^1_{sm}}\right) * Q^2_{sm} - MA_i \geq 0
\end{aligned}
\tag{4}
$$

WCET Derivation Under Single Core Equivalence With Explicit Memory Budget Assignment*

Renato Mancuso[1], Rodolfo Pellizzoni[2], Neriman Tokcan[3], and Marco Caccamo[4]

1 **University of Illinois at Urbana-Champaign, Urbana, IL, USA**
rmancus2@illinois.edu
2 **University of Waterloo, Waterloo, Canada**
rpellizz@uwaterloo.ca
3 **University of Illinois at Urbana-Champaign, Urbana, IL, USA**
tokcan2@illinois.edu
4 **University of Illinois at Urbana-Champaign, Urbana, IL, USA**
mcaccamo@illinois.edu

Abstract

In the last decade there has been a steady uptrend in the popularity of embedded multi-core platforms. This represents a turning point in the theory and implementation of real-time systems. From a real-time standpoint, however, the extensive sharing of hardware resources (e.g. caches, DRAM subsystem, I/O channels) represents a major source of unpredictability. Budget-based memory regulation (throttling) has been extensively studied to enforce a strict partitioning of the DRAM subsystem's bandwidth. The common approach to analyze a task under memory bandwidth regulation is to consider the budget of the core where the task is executing, and assume the worst-case about the remaining cores' budgets.

In this work, we propose a novel analysis strategy to derive the WCET of a task under memory bandwidth regulation that takes into account the exact distribution of memory budgets to cores. In this sense, the proposed analysis represents a generalization of approaches that consider (i) even budget distribution across cores; and (ii) uneven but unknown (except for the core under analysis) budget assignment. By exploiting the additional piece of information, we show that it is possible to derive a more accurate WCET estimation. Our evaluations highlight that the proposed technique can reduce overestimation by 30% in average, and up to 60%, compared to the state of the art.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

Keywords and phrases real-time multicore, WCET, single-core equivalence, DRAM management, certification

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.3

1 Introduction

Multi-core system-on-chip (SoC) are mainstream products. In multi-core platforms, applications can concurrently access shared hardware resources, such as: DRAM, memory bus(es), shared cache(s), and I/O channels. From a real-time perspective, the extensive sharing of

* The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant numbers CNS-1302563 and CNS-1646383, NSERC DG 402369-2011 and CMC Microsystems. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF and other sponsors.

29th Euromicro Conference on Real-Time Systems (ECRTS 2017).
Editor: Marko Bertogna; Article No. 3; pp. 3:1–3:23

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

hardware resources shakes a fundamental assumption of traditional real-time theory, i.e. that the composition of individually analyzed real-time tasks could be used to reason about the system as a whole. Many factors affect the composability of multi-core systems. There is a large consensus, however, that unregulated contention at the level of memory resources (caches, buses, DRAM banks) represents a major source of unpredictability. In fact, when one or more of these resources creates a bottleneck, it causes severe inter-core interference. When a given shared resource becomes a bottleneck with respect to overall system's activity, the worst-case execution time (WCET) of a task on a core can be significantly impacted. Clearly, without imposing a deterministic regulation strategy to cope with high memory resource utilization, interference-induced delays can be hard to model, forcing WCET and schedulability analysis to be more pessimistic.

In recent years, there has been a downtrend in the production and availability of high-performance single-core chips. On the other hand, avionic and automotive industries possess a large base of certified software developed for single-core chips. As such, an *en-masse* migration of industrial players in the safety-critical systems domain are due to a migration to multi-core SoCs. Unfortunately, due to inter-core interference, schedulability analysis results derived for single-core systems cannot be reused when migrating to multi-core platforms. As a part of our previous work, we proposed Single Core Equivalence technology (SCE) [21]. Under SCE, access to shared memory resources are strictly partitioned using a set of OS-level techniques. SCE exploits budget-based memory bandwidth regulation to cope with concurrent activity of tasks on the DRAM subsystem. In our previous work [21, 28], we considered even memory budget assignment across cores, and proposed an analytic model of SCE to estimate the WCET(m) of a task under analysis given the knowledge of its behavior in isolation. In this work, we derive the WCET analysis by making the following extensions: (i) we relax the assumption that the main memory controller bandwidth is evenly distributed among the different cores; and (ii) we consider the exact distribution of memory budgets to cores, and derive a more accurate WCET estimation. Hence, in this paper, we discuss how to calculate the WCET based on the specific budget assignment $\mathcal{Q}$ known at system design time. Only the m active cores have a non-zero budget assignment in $\mathcal{Q}$. It follows that the new WCET($\mathcal{Q}$) implicitly depends on m, but also takes into account the specific setup of a per-core memory bandwidth regulation mechanism. We detail the derivation of WCET($\mathcal{Q}$) for a real-time task given a knowledge of its behavior in isolation. This is the first work to derive a WCET analysis with *uneven* and *explicitly known* memory budget distribution for tasks that run on top of a performance isolation framework for COTS multi-core systems. Thereby, this work makes the following contributions:

- Analysis of WCET for tasks under budget-based memory bandwidth regulation with explicit per-core budget assignments;
- Extension of response-time analysis to incorporate the new WCET derivation for tasks that are scheduled synchronously (bound) or asynchronously (unbound) with respect to the memory regulation period;
- Simulation-based evaluation to compare the derived analysis with exact (brute-force) analysis and state-of-the-art analysis for uneven memory budgets, as proposed in [37].

The rest of the paper is organized as follows. Section 2 provides an overview of the related work. We first briefly review the key components of the SCE framework in Section 3. Next, we discuss the system model and assumptions in Section 4. Section 5 details the proposed WCET($\mathcal{Q}$) analysis, while scheduling considerations are discussed in Section 6. Simulation-based evaluation is presented in Section 7. The paper concludes in Section 8.

2 Related Work

The problem of inter-core interference in multi-core architectures is well known and has been largely studied in literature. As a result, several different ways of approaching the new challenges introduced by multi-core platforms have been proposed. Static analysis has been widely adopted to model the behavior of shared caches in [35, 17, 12] and shared memory buses [26, 14]. A system-wise static analysis for an abstracted multi-core platform that takes into account CPU pipelines, a shared cache and a main memory bus has been proposed in [6], on top of which a unified WCET framework has been formulated [5]. Static analysis certainly represents a promising approach to the problem of determining the WCET of tasks in a multi-core system. Nonetheless, at the current level of refinement, strong assumptions are needed to carry out static analysis, e.g. a LRU cache policy, TDMA-based buses, or the presence of a fixed workload in the system.

A different approach consists in designing multi- and many-core architectures to implement deterministic resource sharing schemes to provide better guarantees on the WCET of real-time tasks. The precision timed (PRET) architecture [9, 4] embeds runtime control and deadline enforcement at the level of processor instruction set while proposing a set of hardware modifications to achieve good performance without sacrificing predictability. In the context of PRET, Reineke et al. proposed a PRET DRAM controller [25] that prevents contention at the level of memory controller by partitioning the physical address space. Specific hardware support in the memory subsystem was also proposed in [22] to lower the pessimism in WCET estimation. Predator, a predictable DRAM controller that internally implements bandwidth regulators was proposed in [2]. Similarly, predictability is enhanced in the MERASA project [32] by reducing inter-core interference at the hardware level.

This work targets commercially available systems, and effectively improves the analysis of a software-based memory bandwidth regulation technique, namely MemGuard [40, 41]. The most related work in this area concerns the design and analysis of protocols for access to shared (memory) resources. Time division multiple access (TDMA) was proposed in [26] as an arbitration protocol for shares buses. A timing analysis for TDMA arbitration was presented in [27]. By using code re-factoring, the PRedictable Execution Model (PREM) [23, 20] allows high-level co-scheduling of clusters of memory requests and CPU execution. PREM has been extended to multi-core systems in [36], and a similar model was adopted in [30, 33, 31] in the context of scratchpad-based platforms. The memory regulation technique (MemGuard) that we analyze belongs to a class of budget-based memory regulation techniques [39]. The key insight is that to each task is assigned a server, so that a fraction of the shared resource's available bandwidth is reserved. CPU resource sharing under periodic resource reservation was considered to derive hierarchical scheduling analysis [29, 8].

Our work has a number of similarities with [3], as both propose a way to estimate a task's WCET on multi-core platforms with shared DRAM subsystem. Three main differences however set the works apart. First, in [3] it is assumed that the exact behavior of a task is known in the form of a trace of executed instructions and memory references. Conversely we hereby assume that only the number of DRAM memory accesses is known, and then derive the arrangement of memory accesses and execution that leads to the worst-case. Second, [3] considers a non-arbitrated DRAM subsystem, while we explicitly account for the effect of memory bandwidth regulation. Third, unlike [3], we only focus on the problem of extending the WCET experimentally calculated in isolation (single-core case) to the multi-core case.

The work in [24] introduced a memory server for multicore systems that can provide better predictability by controlling at a fine granularity internal parameters of the DRAM

subsystem. WCET analysis for the considered bandwidth regulation technique was originally proposed in the context of SCE [21, 28], under the assumption that equal budget is assigned to each core. Yao et al. in [37] proposed an analysis for MemGuard considering *uneven* bandwidth assignment. In [37, 24], however the analysis is carried out assuming that only the budget of the core under analysis is known, while the budgets assigned to other cores are not. Conversely, we relax this assumption and show that major improvements in terms of WCET calculation can be obtained.

3 Background about SCE Components

In this section, we briefly introduce few background concepts about each of the integrated resource management techniques comprising SCE: Colored Lockdown for shared cache management; MemGuard for DRAM bandwidth reservation; PALLOC for DRAM bank partitioning.

3.1 Colored Lockdown – Cache Assignment

SCE leverages Colored Lockdown [19] to mitigate inter-core interference at the cache level by allocating (locking) application memory areas in last-level cache. Colored Lockdown involves two main stages: an offline profiling stage; and an online cache allocation stage. During the offline stage, the task is analyzed to build a *profile*. The generated profile is effectively a list of memory pages ranked by access frequency. During the online stage, the most frequently accessed (hot) pages in the profile, up to a cutoff threshold, are locked in cache. By varying the cutoff threshold from 0 to the number of entries of the profile[1], it is possible to derive a *progressive lockdown curve* (PLC) [21]. The PLC plots the WCET of a task as function of the number of hot memory pages allocated in cache. In other words, if x is the number of profile pages to allocate for an application, the output of PLC(x) contains two pieces of information: (1) a corresponding value of WCET C for the task running in isolation (single-core scenario); and (2) a residual maximum number of cache misses μ, corresponding to accesses to all those profile pages not allocated in cache. As we show in Section 4, the C and the μ parameters obtained at this step represent the starting point to derive the value of WCET under memory bandwidth regulation with m active cores. Finally, note that the PLC could also be derived using static analysis tools.

3.2 MemGuard – Memory Bandwidth Partitioning

Similarly to shared caches, DRAM memory is one of the main sources of inter-core interference. To improve isolation, SCE uses MemGuard [40]. The goal of MemGuard is to provide bandwidth reservation on a per-core basis. MemGuard uses a series of per-core regulators that are responsible for monitoring and enforcing the memory bandwidth allocation. Each regulator monitors the amount of DRAM transactions performed by each core (or alternatively, the number of last-level cache misses) via hardware-specific performance monitoring capabilities. By considering the worst-case latency L_{max} for a single memory request to be serviced, it is possible to derive a worst-case (guaranteed) bandwidth at which the memory subsystem can operate. MemGuard operates as follows: it is configured to enforce the bandwidth assignment at a given period P. Based on L_{max}, it is possible to compute a **total budget**

[1] Or up to the maximum number of pages that can be locked in cache.

in terms of number of memory transactions that can be globally performed during P. This parameter, namely Q, can be computed as $Q = \frac{P}{L_{max}}$. From the global budget, **per-core budgets** Q_i can be assigned arbitrarily, as long as their total does not exceed Q_i. At the beginning of each period, MemGuard configures the hardware performance counters to trigger an event when any of the cores exceeds its Q_i threshold of completed DRAM memory requests. To enforce the strict bandwidth assignment, upon reception of a budget-exhausted event, MemGuard idles the associated core. Any idled core resumes its activity at the beginning of the next replenishment period P. The length of the regulation period P is a system-wide parameter and should be much smaller than the minimal application task period. In our current implementation, $P = 1$ ms, matching also the OS scheduler tick interval. MemGuard also offers different schemes to share reserved yet unused memory bandwidth across cores to achieve significant average-case performance improvements. In this paper, we are only concerned with strict bandwidth assignment, while additional details on bandwidth reclaiming and sharing mechanisms can be found in [40].

3.3 PALLOC – DRAM Bank Partitioning

The DRAM structure is organized into ranks, banks, rows and columns [13]. Whenever a given row is accessed in a bank, subsequent accesses on the same row (row-hits) can be serviced with a small latency. Conversely, if a subsequent access requires data in a different row (row-miss), a significant increase in the latency is introduced. Different banks of the same DRAM chip can satisfy requests in parallel [38, 15]. In a multi-core scenario, several cores can potentially access the same DRAM bank. In this case, (i) the row-miss ratio of a task can increase as multiple cores access the same bank; and (ii) requests originated by the core under analysis can be re-ordered after other cores' requests, introducing additional delay [15, 24]. To mitigate inter-core interference at the level of DRAM banks, *private banking* can be used. Under private banking, non overlapping sets of DRAM banks are assigned to different cores. SCE uses a DRAM bank-aware OS-level memory allocator, namely PALLOC [38], which allows system designers to assign specific DRAM banks to cores (or applications) and to enforce private banking. This way, tasks running in parallel do not collide on DRAM banks and do not suffer inter-core conflicts at this level, as long as there is a sufficient number of banks to accommodate them. A detailed discussion on how PALLOC works can be found in [38].

4 System Model and Assumptions

In this section, we discuss the considered system model as well as the assumptions under which the analysis is performed. Table 1 summarizes the list of parameters that will be used throughout the paper to calculate the WCET($\mathcal{Q}$) of a task in a system where m represents the number of active cores and $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$ represent the budget assignment to the cores. The budgets in $\mathcal{Q}$ are **sorted** in ascending order. The parameter P represents the budget replenishment period of MemGuard such that the memory access budget for each core i will be restored to Q_i every P time units. Q_i represents the number of memory accesses that a given core i is allowed to perform within each MemGuard period of length P.

The parameter C captures the WCET of the considered task running in isolation once the last-level cache assignment for the task has been determined using Colored Lockdown. Under this scenario, the maximum value of residual cache misses μ can be obtained. The values of C and μ can be derived by using either static analysis or an experimental approach. Static analysis can be used whenever a micro-architectural model for the considered platform

Table 1 Summary of parameters for SCE response-time analysis.

Param.	Interpretation
m	Number of active cores in the system
P	MemGuard budget replenishment period
C	WCET for the considered task in isolation
μ	Number of residual misses after last-level cache assignment
E	Execution-only time in slots of length L_{max}
L_{min}	Minimum amount of time for a single memory request
L_{max}	Maximum amount of time for a single memory request
Q_i	Maximum number of memory requests for core i over P
Q	Maximum number of memory requests that can be globally performed in P

is available [7, 1]. On single-core systems, it is common industrial practice to experimentally derive the value of WCET. Hence, C could be derived by reusing the same practices on a multi-core platform by idling all but one cores. Existing tools that adopt this approach are part of the industry practice toward WCET determination on single-core platforms [34, 16].

Prefetchers, branch predictors, and speculative execution units are assumed to be disabled. Thanks to private banking, the DRAM requests from a core under analysis are never re-ordered after a group of requests from a different core. Each core is allowed to have more than one outstanding memory request, and we assume that the DRAM controller globally implements a round-robin[2] scheduling policy. In other words, read (write) memory requests reach the DRAM controller. Requests are then dispatched to DRAM banks, and their responses from the banks are forwarded on the bus in a round-robin scheme (single memory server). We assume that read and write requests are treated equally. We also assume that the maximum time to complete a memory transaction L_{max} is also the maximum delay introduced by other cores' individual requests. Note that this is not true in general, as the latter value can be significantly smaller than L_{max}. Albeit significant improvements can be obtained on the final WCET estimation, assuming a delay value different from L_{max} complicates the mathematical formulation without fundamentally impacting the general approach. For this reason, we leave this discussion as part of our future work. The best-case memory access latency is captured by the parameter L_{min}. It follows that the time to perform a single transaction is bounded between L_{min} and L_{max}.

It is important to distinguish between time spent in memory and time spent for pure execution on the CPU C^e. For in-order processors, where each memory request is blocking, there is no overlapping in time between memory and computation. Hence, a safe upper-bound on C^e can be computed as: $C^e = C - \mu \cdot L_{min}$. For out-of-order processors, the overlapping could range from 0 to $\mu \cdot L_{max}$. Hence $C^e = C$ is always a safe upper-bound, but it could be significantly improved using static analysis to reduce the pessimism on the amount of computation/memory overlapping. Since the granularity at which we conduct our analysis is L_{max}, we will often consider pure computation time (no memory) in slots of length L_{max}. The resulting slotted computation time E can be derived as $E = \lceil \frac{C^e}{L_{max}} \rceil$.

We do not address the problem of contention at the level of shared cache bus and controller. The cache bus is normally on-chip and features a much higher bandwidth

[2] Many modern COTS multi-core chips (e.g. Freescale MPC56xx and MPC57xx chip families) designed for safety-critical operations typically implement a round-robin policy on the main memory controller arbiter.

compared to the main memory bus. Contention at this level has been found to affect some modern platforms [10]. However, due to its high bandwidth capabilities, cache buses are normally able to sustain, without hitting the saturation point, the simultaneous activity of several CPUs. According to our benchmarks, on the Freescale P4080 platform used for our evaluations, contention at this level produces no visible effects up to 4 active cores and yields negligible slowdowns with 8 active cores.

Additionally, COTS architectures are often not time-composable between CPU pipeline and cache hierarchy [18]. This means that under certain circumstances, a local reduction of execution time (e.g. a cache hit) can produce a global increment in the execution time and vice versa, determining what is known as timing anomaly. If an experimental approach is used, the effect of timing anomalies is largely embedded in the experimentally derived WCET. This is because, thanks to Colored Lockdown, there is no variation in the sets of accesses that hit/miss in cache. Nonetheless, recent studies [11] suggest that the timing effect arising from timing anomalies can be statically analyzed and accounted at design time without significantly pessimistic overestimation.

In this work, we assume that per-core budgets Q_i are assigned at design-time to each core and are thus explicitly known. Moreover, we assume that the total bandwidth assignment does not exceed the guaranteed bandwidth. In other words, it must hold that:

$$\sum_{i=1}^{m} Q_i = \frac{P}{L_{max}} . \tag{1}$$

The analysis performed in the next sections will be done on each core i individually. We will indicate parameters that are core-specific with the i subscript (e.g. Q_i). To avoid overloading the notation, we will omit any index on per-task parameters (e.g. C, μ).

5 Worst-Case Derivation

In this section, we discuss how WCET derivation can be performed using the assumptions made in Section 4.

5.1 Memory/CPU Configurations

A distinctive aspect of our analysis is that we explicitly consider the effect that bandwidth regulation has on the behavior of the cores. For instance, suppose that our core under analysis (Core 2) has budget $Q_2 = 2$. Now consider a task running on core 2 that wants to perform 2 memory accesses back-to-back. Now, assume that there is only one more core in the system (Core 1), with budget $Q_1 = 1$. Thanks to the regulation mechanisms, Core 1 can interfere with only one of the two memory requests performed by Core 2, because Core 1 will be regulated after interfering once.

In order to study the worst-case execution time of tasks under analysis, it is fundamental to understand all the possible worst-case memory access patterns within a single MemGuard period P. Since we consider round-robin arbitration, the possible memory access patterns within a single regulation period can be derived combining (i) round-robin arbitration of resources from different cores; and (ii) the effect of regulation. In order to explain how this can be achieved, let us consider an example setup.

In our example, we consider $m = 4$ cores and a period of length $P = 2\ ms$. Assuming a value of $L_{max} = 0.2\ ms$, the number of memory transactions that can be performed at the guaranteed bandwidth within a regulation period is: $Q = \frac{P}{L_{max}} = 10$. Let us assume that

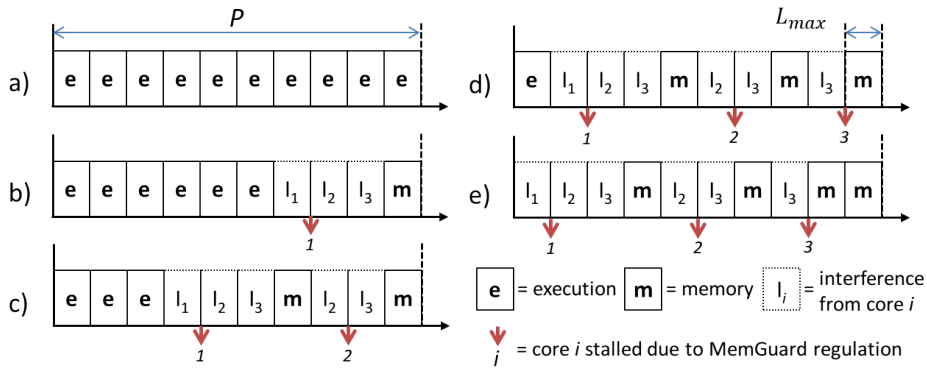


Figure 1 Possible memory access patterns within a regulation period P for Core 4 with budget assignment $\mathcal{Q} = \{1, 2, 3, 4\}$.

the bandwidth assignment to cores is the following: $Q_1 = 1, Q_2 = 2, Q_3 = 3$ and $Q_4 = 4$. In order to visualize the possible patterns for Core $i = 4$, consider Figure 1.

In Figure 1a, a generic task running on core 4 only performs execution, hence, it suffers no interference from other cores. For this reason, the task will execute for an amount of time that corresponds to the length of 10 memory transactions of length L_{max}. In order to carry out our analysis, we consider time spent for execution (no memory) at a granularity that is useful for our calculations. For this reason, we consider execution as if it progresses in *slots* of length L_{max}. In this sense, the pattern in Figure 1a can be interpreted as if the task under analysis performs 10 *execution slots* of size L_{max}.

In Figure 1b, the task under analysis performs one memory transaction. Since memory transactions are satisfied following a round-robin policy, in the worst case the task will suffer interference on that single transaction from all the other cores. The corresponding pattern consists of 6 execution slots and a single memory transaction. From the point of view of Core 4 (core under analysis), The rest of the time in P is wasted due to the activity of other cores (contention).

In Figure 1c, the task under analysis performs 2 memory transactions. However, since $Q_1 = 1$, Core 1 will only interfere with either the first or the second transaction before being regulated by MemGuard. Hence, 3 cores will interfere on one memory transaction, but only 2 cores (Core 2 and 3) will interfere on the other memory transaction. Similarly, in Figure 1d, the same reasoning applies. Moreover, after the second memory transaction, Core 2 is regulated as well, leaving only Core 3 to interfere on the third transaction of the core under analysis. Finally, in Figure 1e, it can be seen that after three memory transactions, all the cores except the core under analysis are regulated, allowing an interference-free memory transaction (two consecutive m-slots in the figure). Each of the patterns in Figure 1 captures a different worst-case given that a task wants to perform a certain number of memory requests, say $M \in \{0, \ldots, Q_i\}$ within P. Three observations can be made to clarify why each pattern captures the worst-case, given that the number M of memory requests to be performed in a given regulation period is known:

1. each request takes L_{max};
2. unless the other cores are regulated, they always interfere with the memory transactions of the core under analysis;
3. the amount of execution performed by the core is the minimum under any scenario that allows M memory requests to be performed within P. Here, the minimum is considered because the final objective is to maximize the number of periods across which computation is performed.

Given a known budget assignment $\mathcal{Q}$ that meets the constraint in Equation 1, it is possible to derive all the possible (worst-case) patterns of memory and computation as in Figure 1. We refer to such set of possible patterns as memory/computation configurations, or simply as **M/C configurations**. A single M/C configuration can be expressed as a pair of the form $\langle M, C\rangle$, where M represents the number of performed memory transactions while C represents the number of executions slots performed within the considered regulation period.

The set Z_i of all the possible M/C configurations for a task running on a given core i can be constructed as follows. First, we define the *cumulative interference* operator $|Q|_h$. This operator considers a given bandwidth-to-core assignment of the form $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$ and returns the number of cores whose bandwidth assignment is greater or equal than the threshold h. For instance, consider the example in Figure 1. In this case, $|Q|_1 = 4, |Q|_2 = 3, |Q|_3 = 2$ and $|Q|_4 = 1$. Next, we construct Z_i as follows:

$$Z_i = \bigcup_{h=0}^{Q_i - 1} \{\langle M_h, C_h\rangle\} \cup \{\langle Q_i, 0\rangle\}, \tag{2}$$

where $M_h = h$, $C_0 = Q$ and $C_h = Q - \sum_{j=1}^{h} |Q|_j$. Hence, the set Z_4 derived for the system presented in Figure 1 will be: $Z_4 = \{\langle 0, 10\rangle, \langle 1, 6\rangle, \langle 2, 3\rangle, \langle 3, 1\rangle, \langle 4, 0\rangle\}$. Note that the quantity $Q - C_h - M_h = \sum_{j=1}^{h} |Q|_j - M_h$ captures the maximum interference suffered by a task on a given pattern with index h in the ordered set Z_i. Clearly, the trend of M_h is linear as one more memory access is considered as the index h increases. Conversely, the trend of C_h depends on the amount of interference suffered by the core and the regulation, as we discuss below.

5.2 Objective Formulation

Once the M/C configurations Z_i for a given core i are known, they apply to all the tasks scheduled on the considered core. Next, we reason on a single task's parameters to formulate the objective for the worst-case execution time calculation, i.e. WCET($\mathcal{Q}$). The work in [37] analyzed the WCET of a task under memory bandwidth regulation by calculating a *stall* term to be added to the task WCET obtained in isolation. In this paper, we adopt a different strategy. Instead of computing a *stall* term, we derive the maximum number of regulation periods required to complete a given task. The problem of deriving the worst-case execution time of a task WCET($\mathcal{Q}$) can be thought as the problem of finding the longest memory access pattern, given the parameters $\mathcal{Q}$, E, μ and the index of the considered core i.

From the way they are constructed, M/C configurations are in a discrete domain. **However, the problem of finding the maximum number of regulation periods in the discrete domain becomes a combinatorial problem. Instead, we reason in a continuous domain and use Theorem 4 to show that what obtained in the continuous domain represents an upper-bound on the value of WCET($\mathcal{Q}$)**. In other words, by only reasoning on **continuous** M/C curves, we show how to calculate an upper bound on the maximum number of regulation periods through which μ memory requests and E slots of computation can span.

▶ **Definition 1.** We define a sequence of regulation periods as *memory access pattern*. A memory access pattern, spanning through L regulation periods, is structured in the following way: the first $L - 1$ periods consume memory and computation according to some $\langle M_h, C_h\rangle \in Z_i$. Each of the $L - 1$ periods can use a different $\langle M_h, C_h\rangle$ element. During the L-th period, any leftover computation/memory is performed. In fact, for the last period we

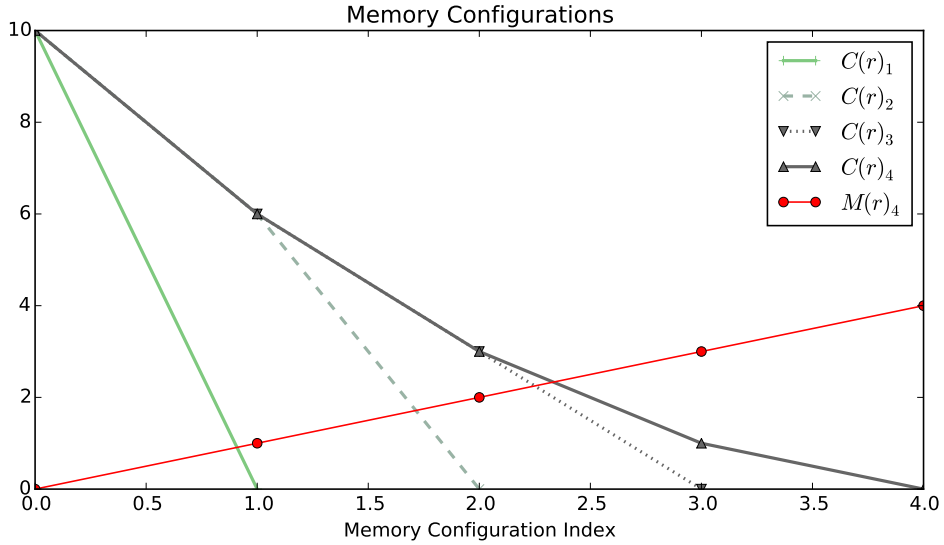


Figure 2 Trend of $M(r) = r$ and $C(r)$. Note the convex trend of the connecting line between the C_h components for $C(r)_4$.

can consider any element in Z_i with some C_h and M_h components that are both larger than the computation and memory leftover, respectively.

▶ **Definition 2.** We define the length of a memory access pattern as the number L of regulation periods that compose the pattern.

The following transformation is used to convert the problem from the discrete domain into the continuous domain. Specifically, we consider each component of each element of Z_i as a point along a *computation consumption curve* $C(r)$ and a *memory consumption curve* $M(r)$ for C_h and M_h components, respectively, with $r \in \mathbb{R}$. For both the curves, the domain depends on the MemGuard budget assignment for the core under analysis. Specifically, $r \in [0, Q_i]$. Since the memory consumption trend is always linear, $M(r)$ can be simply defined as the connecting line among M_h components, or simply $M(r) = r$. Conversely, $C(r)$ is defined as follows:

$$C(r) = \begin{cases} C_r & \text{if } \langle M_r, C_r \rangle \in Z_i \\ C_h + (C_{h+1} - C_h)(r - h) & \text{if } C_h < r < C_{h+1} : \langle M_h, C_h \rangle, \langle M_{h+1}, C_{h+1} \rangle \in Z_i \end{cases} \tag{3}$$

In other words, $C(r)$ is defined as the connecting line between the components C_h in Z_i, as depicted in Figure 2. In the figure, we consider $m = 4$ cores and a fixed budget assignment $\mathcal{Q} = \{1, 2, 3, 4\}$, and show the resulting $C(r)$ curves for each core, labeled as $C(r)_1, \ldots, C(r)_4$. In the figure, only $M(r)_4$ is depicted, since it is similar on all the cores and only varies in the length of its domain.

Consider $C(r)_4$ and $M(r)_4$ depicted in Figure 2. In this example, the two functions are piece-wise continuous and **convex curves**. Note that $M(r)_i$ always exhibits a linear trend, hence it is always convex. $C(r)_i$ has a convex trend when we consider the core(s) i with the highest budget assignment in $\mathcal{Q}$. Highest-budget cores will never suffer regulation. This sufficient condition can be simply expressed as $Q_i = \max\{\mathcal{Q}\}$. If this condition is not satisfied, $C(r)$ could be still convex. It is the case of $C(r)_3$ in Figure 2. Nonetheless, in general, this property cannot be assumed if $Q_i \neq \max\{\mathcal{Q}\}$. $C(r)_2$ in Figure 2 falls in the

latter case. If both functions are convex, it means that in a single period, a linear increase in execution slots results in a more-than-linear decrease in memory requests, and vice-versa. We first discuss how an upper-bound on the worst-case execution time can be derived in case of convexity. Next, we discuss how non-convexity can be handled in Section 5.4.

5.3 Max Length for Memory Access Pattern (convex case)

We now focus on identifying the worst-case access pattern under the hypothesis of convex M/C configurations. Let us now introduce the quantity $P(r)$ as the upper-bound on the length L of the memory access pattern when performing $C(r)$ slots of computation and $M(r)$ memory requests per period. $P(r, E, \mu)$ is defined according to Lemma 3. Note that $P(r, E, \mu)$ depends on three parameters. Since E and μ are constant for a given task, we abuse the notation and only denote this function (and successive derivations) as $P(r)$.

▶ **Lemma 3.** *Consider a point $r \in [0, Q_i]$ in the domain of continuous M/C curves. When $C(r)$ slots of computation and $M(r)$ memory transactions are performed during each regulation period, the resulting length of the memory access pattern is $P(r)$:*

$$P(r) = \begin{cases} P^e(r) & \text{if } \frac{E}{C(r)} < \frac{\mu}{M(r)} \\ P^m(r) & \text{if } \frac{E}{C(r)} > \frac{\mu}{M(r)} \\ \max\{P^e(r), P^m(r)\} & \text{if } \frac{E}{C(r)} = \frac{\mu}{M(r)} \end{cases} \tag{4}$$

where $P^e(r)$ and $P^m(r)$ are defined as follows:

$$P^e(r) = \left\lceil \frac{E}{C(r)} \right\rceil + \left\lceil \frac{\mu - M(r)\left\lceil \frac{E}{C(r)} \right\rceil}{Q_i} \right\rceil^+ \tag{5a}$$

$$P^m(r) = \left\lceil \frac{\mu}{M(r)} \right\rceil + \left\lceil \frac{E - C(r)\left\lceil \frac{\mu}{M(r)} \right\rceil}{Q} \right\rceil^+ \tag{5b}$$

and the operator $\lceil f(x) \rceil^+ = \max\{\lceil f(x) \rceil, 0\}$.

Proof. Let us first focus on the condition between the first two cases. $E/C(r)$ represents the (decimal) number of regulation periods to complete the amount of computation slots E. Similarly, $\mu/M(r)$ expresses the number of regulation periods to perform all μ memory transactions. The first case in Equation 4 represents the case in which E computation slots are completed before μ memory transactions are performed. The second case captures the opposite case.

Case $E/C(r) < \mu/M(r)$: In this case, execution is completed within the first $P_{init} = \lceil E/C(r) \rceil$ regulation periods. After that point, only memory requests are left to be performed. However, *some* memory requests have been performed in the initial P_{init} periods. This amount is at most $M(r) \cdot P_{init}$. If the quantity $\mu - M(r) \cdot P_{init}$ is positive, then this represents the amount of memory transactions that are left to be performed. Since execution slots have been completed, the memory transactions after the first P_{init} periods will be performed back-to-back. Since core i under analysis is subject to regulation with budget Q_i, it will take at least $P_{left} = \left\lceil \frac{\mu - M(r) \cdot P_{init}}{Q_i} \right\rceil$ to complete the leftover memory requests (if any) with budget Q_i. Equation 5a is obtained by summing $P_{init} + P_{left}$.

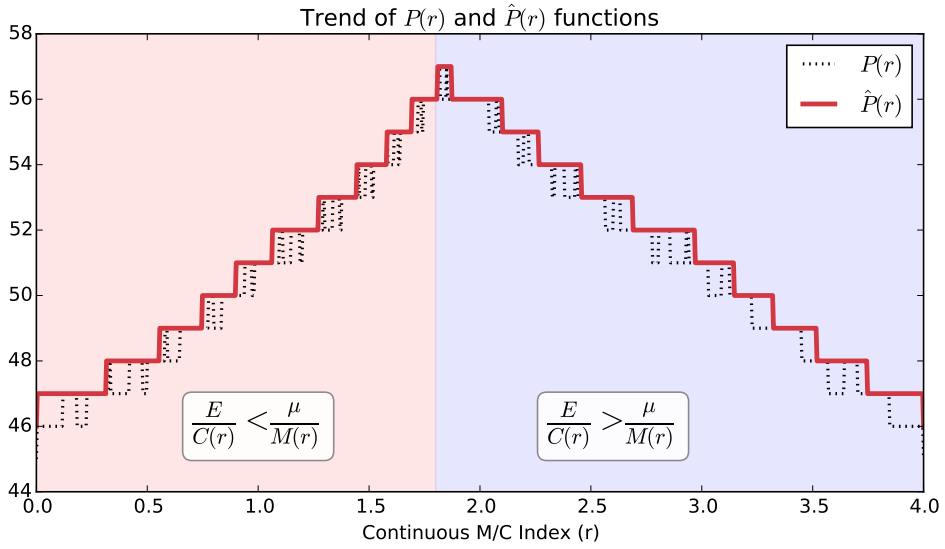


Figure 3 Trend of $P(r)$, $\hat{P}(r)$ for a system with $m = 4$, $\mathcal{Q} = \{1, 2, 3, 4\}$, and a task on core $i = 4$ with parameters $E = 200, \mu = 100$.

Case $E/C(r) > \mu/M(r)$: This case is analogous to the first case, by swapping the role of memory and computation. In fact, in this case $P_{init} = \lceil \mu/M(r) \rceil$ is the number of periods to entirely perform μ memory requests at a per-period rate of $M(r)$. The leftover computation is $E - C(r) \cdot P_{init}$, if this quantity is positive. Once memory has been entirely performed, the remaining regulation periods are entirely filled with computation since no regulation is suffered. Hence, Q execution slots per period will be performed after P_{init}. The leftover (if any) computation is completed in $P_{left} = \left\lceil \frac{E - C(r) \cdot P_{init}}{Q} \right\rceil$. The sum of $P_{init} + P_{left}$ provides the length of the memory access pattern in this case and is equivalent to Equation 5b.

Case $E/C(r) = \mu/M(r)$: Finally, in this point, the value of $P(r)$ is simply the maximum between Equation 5a and 5b. ◀

To find an upper-bound on WCET($\mathcal{Q}$), we first find the value of r that maximizes $P(r)$. Since $C(r)$ is defined as a piece-wise linear curve, an easy way to reason independently on each segment and find the value of r on each segment that maximizes the function. In order to find the maximum value of $P(r)$, we reason on each of the individual segments of the $C(r)$ curve individually. In practice, this can be significantly optimized by considering that the changes of slope in $C(r)$ is less than or equal to $i - 1$. Due to space constraints, we do not discuss possible optimization. Consider an arbitrary (integer) value of $h \in \{0, \ldots, Q_i - 1\}$. Clearly $M(r) = r$. We can also write $C(r)$ in the domain $r \in [h, h + 1]$ as:

$$C(r) = C_h + (C_{h+1} - C_h) \cdot (r - h) = \beta r + \gamma, \tag{6}$$

where $\gamma = \alpha - h\beta$, $\alpha = C_h$, and $\beta = (C_{h+1} - C_h)$. Note that α, β and γ are all constant in the considered segment of $C(r)$. Furthermore, note that $\gamma \geq 0$ and $\beta \leq 0$.

Let us focus on the case where the second term of $P^e(r)$ and $P^m(r)$ is non-zero. By expanding Equation 4 in the considered segment, we can rewrite the two terms of $P(r)$,

i.e. $P^e(r)$ and $P^m(r)$, locally as[3]:

$$P^e(r) = \left\lceil \left\lceil \frac{E}{\beta r + \gamma} \right\rceil \left(1 - \frac{r}{Q_i}\right) + \frac{\mu}{Q_i} \right\rceil \tag{7a}$$

$$P^m(r) = \left\lceil \left\lceil \frac{\mu}{r} \right\rceil \left(1 - \frac{\beta r + \gamma}{Q}\right) + \frac{E}{Q} \right\rceil \tag{7b}$$

Furthermore, since it always holds that $\lceil f(x) \rceil \leq f(x) + 1$, it is possible to upper-bound each term in $P(r)$ and remove the inner ceiling by considering $\hat{P^e}(r)$ and $\hat{P^m}(r)$ defined as follows:

$$\hat{P^e}(r) = \begin{cases} \left\lceil \frac{E}{\beta r + \gamma} \right\rceil & \text{if } \left\lceil \frac{E}{C(r)} \right\rceil > \frac{\mu}{M(r)} \\ \left\lceil \left(\frac{E}{\beta r + \gamma} + 1 \right) \left(1 - \frac{r}{Q_i}\right) + \frac{\mu}{Q_i} \right\rceil & \text{otherwise} \end{cases} \tag{8a}$$

$$\hat{P^m}(r) = \begin{cases} \left\lceil \frac{\mu}{r} \right\rceil & \text{if } \left\lceil \frac{\mu}{M(r)} \right\rceil > \frac{E}{C(r)} \\ \left\lceil \left(\frac{\mu}{r} + 1 \right) \left(1 - \frac{\beta r + \gamma}{Q}\right) + \frac{E}{Q} \right\rceil & \text{otherwise} \end{cases} \tag{8b}$$

Note that Equation 8a (resp., Equation 8b) has two cases. The first case is used when the second term of Equation 5a (resp., Equation 5b) is zero; otherwise, the second case is used with the simplification performed in Equation 7a (resp., Equation 7b) and following upper-bounding. By using $\hat{P^e}(r)$ and $\hat{P^m}(r)$, we define $\hat{P}(r) \geq P(r)$ as:

$$\hat{P}(r) = \begin{cases} \hat{P^e}(r) & \text{if } \frac{E}{\beta r + \gamma} < \frac{\mu}{r} \\ \hat{P^m}(r) & \text{if } \frac{E}{\beta r + \gamma} > \frac{\mu}{r} \\ \max\{\hat{P^e}(r), \hat{P^m}(r)\} & \text{if } \frac{E}{\beta r + \gamma} = \frac{\mu}{r} \end{cases} \tag{9}$$

Note that $\hat{P}(r)$ is a function with one variable (r). Moreover, the value r^* that maximizes $\hat{P}(r)$ can be found by reasoning without the outer ceiling in $\hat{P^e}(r)$ and $\hat{P^m}(r)$. Within each segment, it is possible to find its critical points using the first-derivative test. The first two terms of Equation 9 are used according to a condition on r that can be rewritten as:

$$\frac{E}{\beta r + \gamma} < \frac{\mu}{r} \implies r < \frac{\mu\gamma}{E - \mu\beta}. \tag{10}$$

The value r_{sw} where the condition in Equation 10 changes, represents a first critical point:

$$r_{sw} = \frac{\mu\gamma}{E - \mu\beta}. \tag{11}$$

Whenever Equation 10 is satisfied, the first term of Equation 9 is considered; otherwise the second term of the equation is used. Let us reason on these two terms of $\hat{P}(r)$ separately. For the first term, the values of r that constitute critical points are:

$$r_{1,1} = -\frac{\sqrt{-E\beta Q_i - E\gamma} + \gamma}{\beta}; \quad r_{1,2} = \frac{\sqrt{-E\beta Q_i - E\gamma} - \gamma}{\beta}. \tag{12}$$

[3] We rely on the property that $\lceil \lceil x \rceil + y \rceil = \lceil x \rceil + \lceil y \rceil$, with $x, y \geq 0$.

Note that $r_{1,1}$ and $r_{1,2}$ are real numbers only when $E\gamma \leq -E\beta Q_i$. The functions $\hat{P^e}(r)$ and $\hat{P^m}(r)$ have a point where the derivative may not exist in r_2 and r_3, respectively:

$$r_2 = -\frac{\gamma}{\beta}; \quad r_3 = \sqrt{\frac{\mu(Q-\gamma)}{\beta}}, \tag{13}$$

under the condition that $\gamma \geq Q$. Finally, the function has a point where the derivative may not exist in $r_4 = 0$.

Recall that $\hat{P}(r)$ is defined over the closed interval $[h, h+1]$, and it has a different expression as the selected segment $h \in \{0, \ldots, Q_i - 1\}$ changes. The boundaries of the interval constitute additional test-points for the maximum. It follows that, for a given segment h, the maximum L_h^* of $P(r)$ can be found as:

$$L_h^* = \max_{r \in \{h, h+1, r_{sw}, r_{1,1}, r_{1,2}, r_2, r_3, r_4\}} \{\hat{P}(r)\}. \tag{14}$$

Clearly, the points $r_{1,1}$, $r_{1,2}$ and r_2 do not need to be evaluated if they do not satisfy Equation 10, or they lie outside the interval $(h, h+1)$; similarly, r_3 and r_4 do not need to be tested if they satisfy Equation 10, or they lie outside the range $(h, h+1)$. Moreover, $r_{1,1}$, $r_{1,2}$ and/or r_3 need to be removed from the set of test-points if they are not real numbers. Conversely, r_{sw} is always evaluated. Finally, an upper-bound on the global maximum L^* of $P(r)$ can be found as follows:

$$L^* = \max_{h \in \{0, \ldots, Q_i - 1\}} \{L_h^*\}. \tag{15}$$

Equation 15 not only provides an upper-bound on the length of the worst-case memory access pattern in the continuous case; but also the rate r^* such that the worst-case memory access pattern can be constructed by a sequence of identical L^* regulation periods, during which $C(r^*)$ (resp., $M(r^*)$) units of computation (resp., memory requests) are performed. Theorem 4 allows us to use the obtained result in case of memory access patterns where each regulation period consumes resources according to discrete M/C configurations.

▶ **Theorem 4.** *Consider a task τ that performs E units of computation and μ memory transactions. Consider the maximum length L_d^* of any memory access pattern of τ constructed using discrete and convex M/C configurations. An upper bound on L_d^* is given by the maximum length L_c^* of a pattern computed using $P(r)$ (see Equation (4)) for a task τ' that performs $E + Q$ units of computation and $\mu + Q_i$ memory transactions, with continuous and convex M/C configuration.*

Proof. The theorem follows from the fact that $C(r)$ and $M(r)$ are convex curves. Let us assume that the longest memory access pattern for τ constructed using a sequence of discrete M/C configurations has length L_d^*. Consider the structure of a task that performs E units of computation and μ memory transactions. It is a sequence of L_d^* regulation periods. Let us use a_h (non-negative integers) to count how many times the element $\langle C_h, M_h \rangle \in Z_i$ appears in the sequence. We have this relation:

$$E \leq E_{tot} = a_0 C_0 + \ldots + a_{Q_i} C_{Q_i} \leq E + Q \tag{16a}$$

$$\mu \leq M_{tot} = a_0 M_0 + \ldots + a_{Q_i} M_{Q_i} \leq \mu + Q_i \tag{16b}$$

where $a_0 \ldots a_{Q_i}$ are integer coefficients that represent how many times a certain (discrete) M/C configuration appears in the pattern. Recall that, by Definition 1, any M/C configuration

can be considered in the last regulation period, as long as the leftovers in computation and memory are both equal or smaller than the considered last-period configuration. This choice does not violate the conditions $E_{tot} \leq E + Q$ and $M_{tot} \leq \mu + Q_i$. It follows that:

$$\sum_{i=0}^{Q_i} a_i = L_d^*. \tag{17}$$

Consider a transformation that translates the non-repeating pattern composed by a sequence of discrete M/C configurations into a repeating pattern with continuous M/C configurations. In the transformed pattern, in each period we perform $C(r_c)$ and $M(r_c)$ units of computation and memory respectively, where r_c is selected as follows:

$$r_c = \frac{\sum_{i=0}^{Q_i} a_i \cdot i}{L_d^*}. \tag{18}$$

Consider the amount of computation performed in each period:

$$C\left(\frac{\sum_{i=0}^{Q_i} a_i \cdot i}{L_d^*}\right) = C\left(\sum_{i=0}^{Q_i} \frac{a_i \cdot i}{\sum_{i=0}^{Q_i} a_i}\right). \tag{19}$$

Recall that the definition of a convex function $f(x)$ imposes that:

$$\lambda_i \in \mathbb{N} \text{ s.t. } \sum_i \lambda_i = 1 \implies f\left(\sum_i \lambda_i x_i\right) \leq \sum_i \lambda_i f(x_i). \tag{20}$$

Further, note that:

$$\sum_{i=0}^{Q_i} \frac{a_i}{\sum_{i=0}^{Q_i} a_i} = \frac{a_0}{a_0 + \ldots + a_{Q_i}} + \ldots + \frac{a_{Q_i}}{a_0 + \ldots + a_{Q_i}} = 1. \tag{21}$$

Since $C(r)$ is convex, it holds that:

$$C\left(\sum_{i=0}^{Q_i} \frac{a_i \cdot i}{\sum_{i=0}^{Q_i} a_i}\right) \leq \sum_{i=0}^{Q_i} \frac{a_i \cdot C(i)}{\sum_{i=0}^{Q_i} a_i} \implies C(r_c) \leq \frac{E_{tot}}{L_d^*}. \tag{22}$$

From this result, it follows that:

$$L_d^* \leq \frac{E_{tot}}{C(r_c)} \leq \frac{E + Q}{C(r_c)}, \tag{23}$$

and, by repeating the convexity considerations, that:

$$L_d^* \leq \frac{M_{tot}}{M(r_c)} \leq \frac{\mu + Q_i}{M(r_c)}. \tag{24}$$

Recall that $\hat{P}(r)$ is actually defined with three parameters: $\hat{P}(r, E, \mu)$, and that $\hat{P}(r, E, \mu)$ can only increase when both the values of E and μ are increased. Take r^* as the value that maximizes $\hat{P}(r^*, E + Q, \mu + Q_i)$, it must hold that:

$$L_c^* = \hat{P}(r^*, E + Q, \mu + Q_i) \geq \hat{P}(r_c, E + Q, \mu + Q_i) \tag{25}$$

Moreover, it follows directly from from Equations 5a and 5b that:

$$\hat{P}(r_c, E + Q, \mu + Q_i) \geq \frac{E + Q}{C(r_c)} \text{ if } \frac{E + Q}{C(r_c)} < \frac{\mu + Q_i}{M(r_c)} \tag{26a}$$

$$\hat{P}(r_c, E + Q, \mu + Q_i) \geq \frac{\mu + Q_i}{M(r_c)} \text{ if } \frac{E + Q}{C(r_c)} > \frac{\mu + Q_i}{M(r_c)} \tag{26b}$$

In case of equality $(E+Q)/C(r_c) = (\mu+Q_i)/M(r_c)$, Equations 26a and 26 are both satisfied. Due to what expressed in Equations 23-26, it follows that $L_c^* \geq L_d^*$, which proves the theorem. ◀

The following corollary can be used to formalize the calculation of WCET($\mathcal{Q}$) based on Theorem 4.

▶ **Corollary 5.** *Consider a task τ that performs E slots of computations and μ memory transactions on core i. Then, an upper-bound on WCET($\mathcal{Q}$) can be computed as:*

$$WCET(\mathcal{Q}) = P \cdot \hat{P}(r^*, E+Q, \mu+Q_i), \tag{27}$$

i.e., by finding the value r^* that maximizes $\hat{P}(r, E+Q, \mu+Q_i)$.

Proof. The proof simply follows from Theorem 4. Note that WCET($\mathcal{Q}$) is expressed in time by multiplying the maximum number of regulation periods $\hat{P}(r^*, E+Q, \mu+Q_i)$ by the length of each period P. ◀

5.4 Max Length for Memory Access Pattern (non-convex case)

Whenever the core under analysis is not one of the cores with the highest memory budget assignment, i.e. when $Q_i \neq \max\{\mathcal{Q}\}$, the curve $C(r)$ may not be convex, but it will be convex until the (Q_i-1)-th component in Z_i.

In order to upper-bound the WCET($\mathcal{Q}$) in this case, we use the following observation. Suppose that the length of the worst-case (longest) memory access pattern is L^*. In this pattern, L^*-k are periods without regulation, where, in each period, computation and memory is consumed according to an element in $Z_i \setminus \langle 0, Q_i \rangle$. The remaining k periods are regulation-only periods, where 0 computation slots are performed and Q_i memory transactions are performed, with k varying between 0 and $\lfloor \frac{\mu}{Q_i} \rfloor$.

Following this observation, an upper-bound on WCET($\mathcal{Q}$) can be calculated as:

$$\max_{k \in [0, \lfloor \frac{\mu}{Q_i} \rfloor]} \{\hat{P}(r_k^*, E+Q, \mu+Q_i-k\cdot Q_i)+k\} = \max_{k \in [0, \lfloor \frac{\mu}{Q_i} \rfloor]} \{\hat{P}(r_k^*, E+Q, \mu+Q_i(1-k))+k\}, \tag{28}$$

where r_k^* is the value of the variable r that maximizes $\hat{P}(r)$ when the amount of computation slots and memory requests to perform is set to $E+Q$ and $\mu+Q_i(1-k)$, respectively.

6 Schedulability Analysis

Once an upper-bound on WCET($\mathcal{Q}$) is derived according to what described in Section 5, it is possible to compute the schedulability of a task-set under fixed-priority scheduling and SCE resource partitioning. First, let us make explicit the parameters with which WCET($\mathcal{Q}$) is invoked, and use the notation WCET($\mathcal{Q}$, C, μ) to indicate an upper bound on the time to perform C execution time units and μ memory transactions under the budget assignment $\mathcal{Q}$. For the purpose of the analysis, we consider a set of implicit-deadline tasks $\tau_1, \ldots, \tau_n$, where each task τ_k is characterized by a period T_k, a WCET in isolation C_k, and a maximum number of residual last-level cache misses after lockdown μ_k. We assume that the scheduling policy is based on tasks' fixed priority assignment (e.g. Rate Monotonic) upon a partitioned multi-core system; this is a common practice used in industrial applications. Next, we consider a level-k busy interval.

We consider two cases: (i) all the tasks' releases and deadlines are aligned with system's tick (*inbound case*); and (ii) tasks' releases and deadlines may not be aligned with system's tick

(*outbound case*). In general, since MemGuard regulation period is synchronized with system's tick, it is a good design practice to have inbound tasks. This has two main benefits: first, a tasks is never released when the MemGuard budget has been exhausted by a previously running task; second, it is possible to derive the response time of tasks by individually considering their WCET = WCET($\mathcal{Q}$, C_k, μ_k). This is because tasks are never preempted in the middle of a regulation period.

6.1 Analysis of Inbound Tasks

In order to check the schedulability of inbound tasks, it is enough to reuse typical response-time analysis. Specifically, for a task under analysis τ_k, it is possible to derive its response-time by finding the first h such that $R_k^{(h+1)} = R_k^{(h)}$ or such that $R_k^{(h)} > T_k$, given that:

$$R_k^{(0)} = \text{WCET}(\mathcal{Q}, C_k, \mu_k) \tag{29a}$$

$$R_k^{(h+1)} = \text{WCET}(\mathcal{Q}, C_k, \mu_k) + \sum_{\tau_j \in hp(\tau_k)} \left\lceil \frac{R_k^{(h)}}{T_j} \right\rceil \cdot \text{WCET}(\mathcal{Q}, C_k, \mu_k), \tag{29b}$$

where $hp(\tau_k)$ represents the set of tasks with priority higher than the task under analysis τ_k. Intuitively, if there exist a h such that $R_k^{(h+1)} = R_k^{(h)}$ and $R_k^{(h)} \leq T_k$, the considered task is schedulable and it is non-schedulable otherwise.

6.2 Analysis of Outbound Tasks

The analysis of outbound tasks is slightly more complicated because tasks can not only be activated in the middle of a regulation period, but also preempted within a regulation period. This, in turn, makes the behavior of a task dependent upon other tasks' memory access patterns. For this reason, instead of reasoning on tasks as separate entities, it is easier to consider the overall worst-case memory access pattern of a level-k busy interval as a whole. When we consider a set of outbound tasks, we can effectively merge the execution (and memory accesses) of an instance of the task under analysis with all the interfering instances of higher priority tasks. In this case, $R_k^{(0)}$ and $R_k^{(h+1)}$ can be calculated as follows:

$$R_k^{(0)} = \text{WCET}(\mathcal{Q}, C_k, \mu_k) + B_i. \tag{30a}$$

$$R_k^{(h+1)} = \text{WCET}(\mathcal{Q}, C_{hep(\tau_k)}, \mu_{hep(\tau_k)}) + B_i, \tag{30b}$$

where $hep(\tau_k)$ denotes the set of tasks with priority greater than or equal to task τ_k. The operators $C_{hep(\tau_k)}$ and $\mu_{hep(\tau_k)}$ are defined as follows:

$$C_{hep(\tau_k)} = \sum_{\tau_j \in hep(\tau_k)} \left\lceil \frac{R_k^{(h)}}{T_j} \right\rceil \cdot C_j, \tag{31a}$$

$$\mu_{hep(\tau_k)} = \sum_{\tau_j \in hep(\tau_k)} \left\lceil \frac{R_k^{(h)}}{T_j} \right\rceil \cdot \mu_j. \tag{31b}$$

Finally, the term B_i is a blocking term that represents the maximum amount of time that a task must wait if it is activated immediately after the MemGuard budget has been exhausted by a previously running task. Since core i is stalled once it has consumed its allocated budget Q_i. The term B_i can be calculated as: $B_i = P - Q_i \cdot L_{min}$.

7 Evaluation

In this section, we evaluate the pessimism of the derived WCET($\mathcal{Q}$) bound compared to the exact worst-case memory access pattern found in the discrete case. We also compare the quality of the derived bound with respect to the analysis in [37].

7.1 Budget generation

A key component of our evaluation is the distribution of budgets to cores. The proposed analysis, in fact, exploits the full knowledge about the assigned budgets to account for regulation of interfering cores. In fact, if even budgets are assigned to all the cores, no benefits are observed compared to the method proposed in [21] and [37]. Thus, we introduce a metric of variance of the budget assignment, namely δ. Consider a total budget Q. The distribution of Q to cores follows a linear trend where δ is the slope of the increase. The total budget assigned to all the cores, however, is always less than or equal to Q. For instance, consider a system with $m = 8$ cores and a value of $Q = 100$. When $\delta = 0$, the assignment $\mathcal{Q}_{\delta=0} = \{12, 12, 12, 12, 13, 13, 13, 13\}$. Conversely, for $\delta = 0.035$ we have $\mathcal{Q}_{\delta=0.035} = \{1, 4, 8, 11, 14, 17, 21, 24\}$.

7.2 Analysis of Pessimism

To compare the discussed bound with the theoretical optimum, we have implemented a back-tracking algorithm that uses a brute-force approach to explore all the possible memory access patterns and to find the maximum. Clearly, the algorithm has a large complexity and its runtime explodes quickly with realistic system parameters. For this reason, we compare the proposed analysis and the exact (brute-force) algorithm on system instances with small parameters. We consider a system with $m = 8$ cores, with $L_{max} = 0.1$ ms, $P = 10$ ms, such that $Q = 100$. We inspect the system with $\delta \in [0, 0.035]$, with randomly-distributed values of $E \in [1, 110]$ and $\mu \in [1, 110]$. Each sample consists of 100 different task parameters. Each of the 100 randomly generated tasks is evaluated on all the 8 cores.

Figure 4 depicts the relative overestimation compared to the exact case in the considered δ range. Given a core and a value of δ, the figure reports the average of the overestimation percentage between the exact derivation and the proposed analysis. To increase the readability of the plot, we omit cores 3,4 and 6 that exhibit intermediate trends. Three characteristics can be noted. First, for low values of δ, tasks behave similarly on all the cores, since budget is evenly distributed. Second, the overestimation is high, starting at 80% on the left of the figure. This, however, is an artificial effect because the randomly generated tasks span across few regulation periods. Unfortunately, performing an evaluation of the brute-force algorithm on larger task parameters becomes computationally unfeasible. Third, as we move toward the right of the figure, there is a sharp drop in overestimation for the low-budget cores. This is because, as less budget is assigned, the length of the task increases; hence the overestimation, which remains constant, has a proportionally smaller weight on the task length.

To validate the last claim, i.e. that the overestimation added by our analysis is roughly constant compared to the exact case, consider Figure 5. In this figure, we use the same tasks generated to plot Figure 4. For each core, the bar at 0 reports the number of tasks with the exact same length in both the brute-force algorithm and our analysis; the bar at 1 counts the number of tasks that are longer by 1 regulation period compared to the exact case; and so on. It can be noted that in all the cases, the overestimation does not exceed 5 regulation periods. Moreover, especially, when the cores with intermediate budgets are considered, tasks are mostly overestimated only by 3 regulation periods.

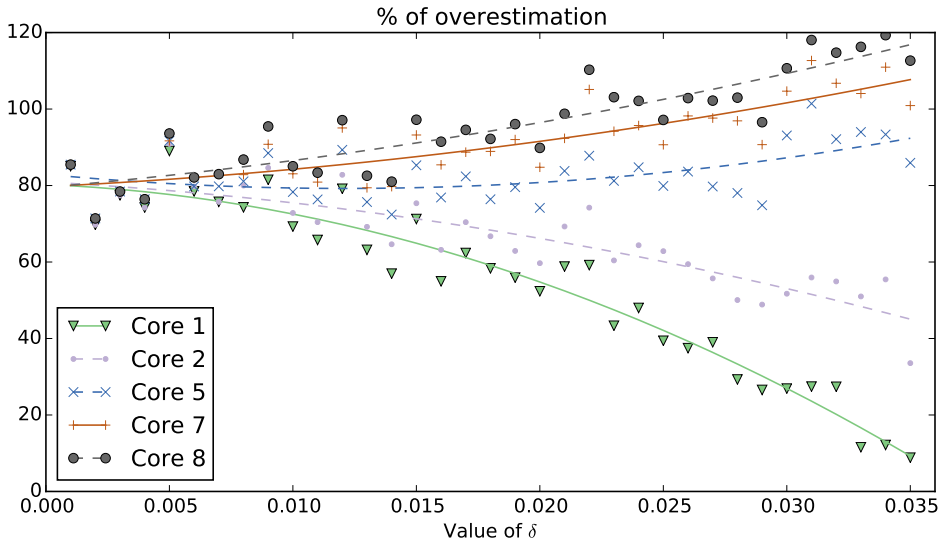


Figure 4 Relative (%) overestimation compared to exact (brute-force) derivation on cores 1, 2, 7 and 8 as a function of δ. Higher y-values correspond to larger overestimation.

7.3 Performance Comparison

In order to understand the benefit of explicitly considering the bandwidth regulation on all the cores, we compare the proposed approach against the analysis in [37]. Both ours and the analysis in [37] share many fundamental assumptions about the system model. The main difference is that the analysis proposed in this work explicitly considers the budget assignment $\mathcal{Q}$ to all the cores. Without this extra piece of information, the analysis in [37] is necessarily more pessimistic, as the worst-case corresponds to the case where the remaining budget is evenly distributed among the remaining cores.

The goal of this experiment is to quantify the gain in terms of WCET derivation. Since the complexity of our analysis is pseudo-polynomial with respect to budgets and task parameters, in this experiment we use system and task parameters that are reflective of a realistic use-case. These parameters are aligned with the evaluation conducted in [21]. We consider $m = 8$, $L_{max} = 4.96 \times 10^{-8}$ s, $P = 1$ ms, such that $Q = 20161$. We inspect the system with $\delta \in [1, 7]$, with randomly-distributed values of $E \in [1, 300000]$ and $\mu \in [1, 200000]$. Once again, each sample consists of 100 different task parameters. Each of the 100 randomly generated tasks is evaluated on all the 8 cores.

The results for this experiment are reported in Figure 6. Three main aspects are relevant to mention. First, for highly differentiated budget assignments (larger values of δ), the proposed algorithm outperforms the analysis in [37], with a reduction of about 30% (left – average), and up to 60% (right – max) in the overestimation of the task's WCET. Second, for lower values of δ, the two algorithms behave almost identically. Third, a slight performance degradation (around 1%) can be observed for high values of δ and low-budget cores. This arises from the fact that in order to upper-bound the WCET, an additional Q and Q_i units of computation and memory, respectively, are added to the task (see Corollary 5). We also believe that different budget assignment schemes, e.g. with exponential increase as opposed to linear, may significantly affect the analysis performances.

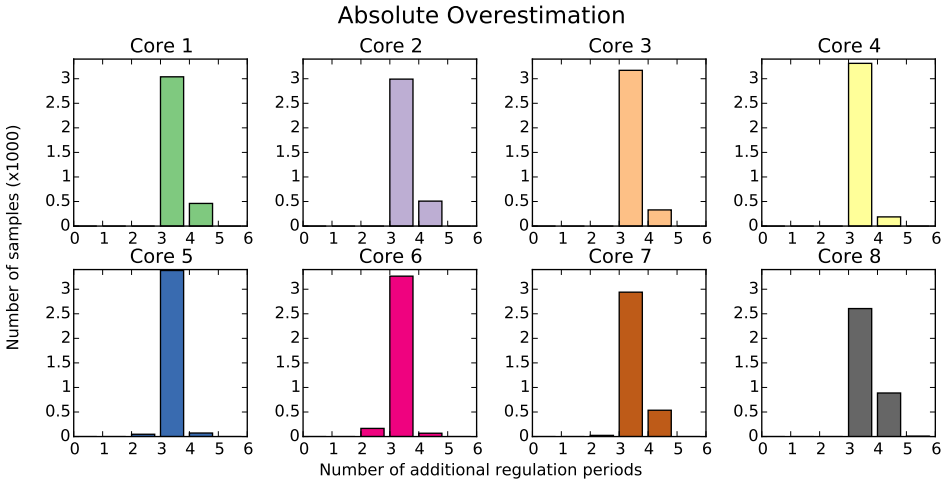


Figure 5 Absolute overestimation, in terms of additional regulation periods, compared to exact (brute-force) derivation on cores 1 to 8.

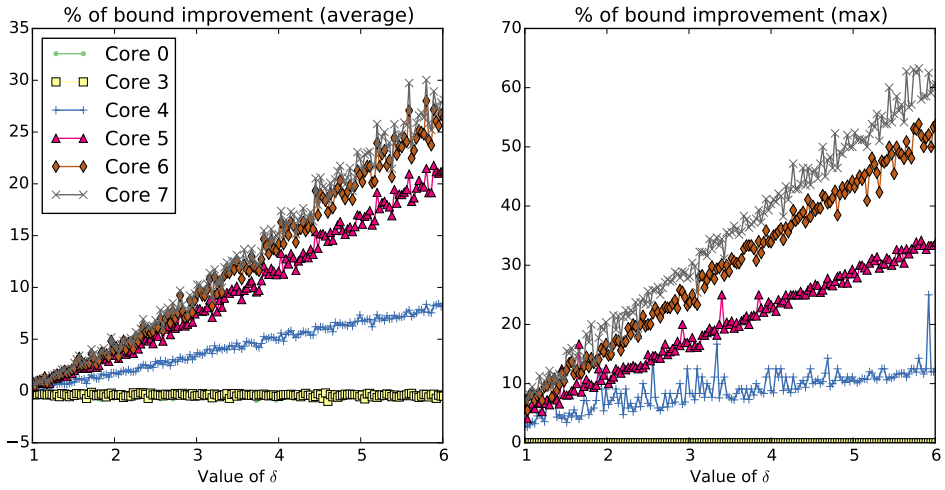


Figure 6 Percentage of WCET improvement over analysis proposed in [37] as a function of δ on cores 1 to 8. Higher y-values means better improvements.

8 Conclusion and Future Work

In this work, we discussed an improved analysis strategy to derive the WCET of a task under memory bandwidth by exploiting exact knowledge of budget-to-core assignments. In this way, we show that it is possible to derive a more accurate WCET estimation, with performance gains that go from 30% in average up to 60%, compared to the state of the art.

As a future work, we plan to validate our analysis on a commercial multi-core platform, and to study how different budget assignment strategies affect the performance of the WCET estimation. Additionally, we plan to extend this work with more in-depth considerations about algorithmic complexity as well as possible optimizations.

References

1 AbsInt. aiT worst-case execution time analyzers, 2014. URL: http://www.absint.com/ait/.

2 B. Akesson, K. Goossens, and M. Ringhofer. Predator: A predictable SDRAM memory controller. In *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS'07, pages 251–256, New York, NY, USA, 2007. ACM. doi:10.1145/1289816.1289877.

3 S. Altmeyer, R. I. Davis, L. Indrusiak, C. Maiza, V. Nelis, and J. Reineke. A generic and compositional framework for multicore response time analysis. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS'15, pages 129–138, New York, NY, USA, 2015. ACM. doi:10.1145/2834848.2834862.

4 D. Bui, E. A. Lee, I. Liu, H. Patel, and J. Reineke. Temporal isolation on multiprocessing architectures. In *Proceedings of the 48th Design Automation Conference*, DAC'11, pages 274–279, New York, NY, USA, 2011. ACM. doi:10.1145/2024724.2024787.

5 S. Chattopadhyay, C. L. Kee, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk. A unified wcet analysis framework for multi-core platforms. *ACM Trans. Embed. Comput. Syst.*, 13(4s):124:1–124:29, April 2014. doi:10.1145/2584654.

6 S. Chattopadhyay, A. Roychoudhury, and T. Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems*, SCOPES'10, pages 6:1–6:10, New York, NY, USA, 2010. ACM. doi:10.1145/1811212.1811220.

7 P. Cousot. Abstract interpretation based formal methods and future challenges. In *Informatics – 10 Years Back. 10 Years Ahead.*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156, London, UK, UK, 2001. Springer-Verlag. doi:10.1007/3-540-44577-3_10.

8 R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, RTSS'06, pages 257–270, Washington, DC, USA, 2006. IEEE Computer Society. doi:10.1109/RTSS.2006.42.

9 S. A. Edwards and E. A. Lee. The case for the precision timed (PRET) machine. In *44th Annual Design Automation Conference*, DAC'07, pages 264–265, New York, NY, USA, 2007. ACM. doi:10.1145/1278480.1278545.

10 G. Gracioli and A. Fröhlich. On the influence of shared memory contention in real-time multicore applications. In *Proceedings of the IV Brazilian Symposium on Computing Systems Engineering (SBESC)*, Washington, DC, USA, 2014. IEEE Computer Society. doi:10.1109/SBESC.2014.8.

11 S. Hahn, M. Jacobs, and J. Reineke. Enabling compositionality for multicore timing analysis. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, RTNS'16, pages 299–308, New York, NY, USA, 2016. ACM. doi:10.1145/2997465.2997471.

12 D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, RTSS'09, pages 68–77, Washington, DC, USA, 2009. IEEE Computer Society. doi:10.1109/RTSS.2009.34.

13 B. Jacob, S. Ng, and D. Wang. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2007.

14 T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. Bus-aware multicore wcet analysis through tdma offset bounds. In *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*, pages 3–12, July 2011. doi:10.1109/ECRTS.2011.9.

15 H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. R. Rajkumar. Bounding and reducing memory interference in COTS-based multi-core systems. *Real-Time Systems*, 52(3):356–395, 2016. doi:10.1007/s11241-016-9248-1.

16 L. Kosmidis, E. Quiñones, J. Abella, G. Farrall, F. Wartel, and F. J. Cazorla. Containing timing-related certification cost in automotive systems deploying complex hardware. In *Proceedings of the 51st Annual Design Automation Conference*, DAC'14, pages 22:1–22:6, New York, NY, USA, 2014. ACM. doi:10.1145/2593069.2593112.

17 Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. *Real-Time Systems*, 48(6):638–680, November 2012. doi:10.1007/s11241-012-9160-2.

18 T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 12–, Washington, DC, USA, 1999. IEEE Computer Society. doi:10.1109/REAL.1999.818824.

19 R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, RTAS'13, pages 45–54, Philadelphia, PA, USA, April 2013. IEEE Computer Society. doi:10.1109/RTAS.2013.6531078.

20 R. Mancuso, R. Dudko, and M. Caccamo. Light-prem: Automated software refactoring for predictable execution on cots embedded systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*, pages 1–10, Aug 2014. doi:10.1109/RTCSA.2014.6910515.

21 R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun. WCET(m) estimation in multi-core systems using single core equivalence. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 174–183, July 2015. doi:10.1109/ECRTS.2015.23.

22 M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware Support for WCET Analysis of Hard Real-time Multicore Systems. *SIGARCH Comput. Archit. News*, 37(3):57–68, June 2009. doi:10.1145/1555815.1555764.

23 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 *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, RTAS'11, pages 269–279, Washington, DC, USA, 2011. IEEE Computer Society. doi:10.1109/RTAS.2011.33.

24 R. Pellizzoni and H. Yun. Memory servers for multicore systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, April 2016.

25 J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. PRET DRAM controller: bank privatization for predictability and temporal isolation. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS'11, pages 99–108, New York, NY, USA, 2011. ACM. doi:10.1145/2039370.2039388.

26 J. Rosén, A. Andrei, P. Eles, and Zebo Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor Systems-on-Chip. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, RTSS'07, pages 49–60. IEEE Computer Society, 2007. doi:10.1109/RTSS.2007.13.

27 A. Schranzhofer, J. J. Chen, and L. 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, April 2010. doi:10.1109/RTAS.2010.24.

28 L. Sha, M. Caccamo, R. Mancuso, J. E. Kim, M. K. Yoon, R. Pellizzoni, H. Yun, R. B. Kegley, D. R. Perlman, G. Arundale, and R. Bradford. Real-time computing on multicore processors. *Computer*, 49(9):69–77, Sept 2016. doi:10.1109/MC.2016.271.

29 I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*, pages 2–13, Dec 2003. doi:10.1109/REAL.2003.1253249.

30 R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo. A real-time scratchpad-centric OS for multi-core embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016 IEEE 22th*, April 2016. doi:10.1109/RTAS.2016.7461321.

31 R. Tabish, R. Mancuso, S. Wasly, S. S. Phatak, R. Pellizzoni, and M. Caccamo. A reliable and predictable scratchpad-centric OS for multi-core embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE 23th*, April 2017.

32 T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quinones, M. Gerdes, M. Paolieri, J. Wolf, H. Casse, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff, and J. Mische. MERASA: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5):66–75, 2010. doi:10.1109/MM.2010.78.

33 S. Wasly and R. Pellizzoni. A dynamic scratchpad memory unit for predictable real-time embedded systems. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 183–192. IEEE, 2013. doi:10.1109/ECRTS.2013.28.

34 I. Wenzel, R. Kirner, B. Rieder, and P. Puschner. Measurement-based worst-case execution time analysis. In *Third IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'05)*, pages 7–10, May 2005. doi:10.1109/SEUS.2005.12.

35 J. Yan and W. Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS'08, pages 80–89, Washington, DC, USA, 2008. IEEE Computer Society. doi:10.1109/RTAS.2008.6.

36 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. doi:10.1007/s11241-012-9158-9.

37 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. doi:10.1109/TC.2015.2425874.

38 H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni. PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms. In *Proceedings of the IEEE Intl. Conference on Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Berlin, Germany, April 2014. doi:10.1109/RTAS.2014.6925999.

39 H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, ECRTS'12, pages 299–308, Washington, DC, USA, 2012. IEEE Computer Society. doi:10.1109/ECRTS.2012.32.

40 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 *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, RTAS'13, pages 55–64. IEEE Computer Society, 2013. doi:10.1109/RTAS.2013.6531079.

41 H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Transactions on Computers*, 65(2):562–576, February 2016. doi:10.1109/TC.2015.2425889.

LTZVisor: TrustZone is the Key*

Sandro Pinto[1], Jorge Pereira[2], Tiago Gomes[3], Adriano Tavares[4], and Jorge Cabral[5]

1 **Centro Algoritmi, Universidade do Minho, Guimarães, Portugal**
`sandro.pinto@dei.uminho.pt`
2 **Centro Algoritmi, Universidade do Minho, Guimarães, Portugal**
`jorge.m.pereira@algoritmi.uminho.pt`
3 **Centro Algoritmi, Universidade do Minho, Guimarães, Portugal**
`tgomes@dei.uminho.pt`
4 **Centro Algoritmi, Universidade do Minho, Guimarães, Portugal**
`atavares@dei.uminho.pt`
5 **Centro Algoritmi, Universidade do Minho, Guimarães, Portugal**
`jcabral@dei.uminho.pt`

Abstract

Virtualization technology starts becoming more and more widespread in the embedded systems arena, driven by the upward trend for integrating multiple environments into the same hardware platform. The penalties incurred by standard software-based virtualization, altogether with the strict timing requirements imposed by real-time virtualization are pushing research towards hardware-assisted solutions. Among existing commercial off-the-shelf (COTS) technologies, ARM TrustZone promises to be a game-changer for virtualization, despite of this technology still being seen with a lot of obscurity and scepticism. In this paper we present a Lightweight TrustZone-assisted Hypervisor (LTZVisor) as a tool to understand, evaluate and discuss the benefits and limitations of using TrustZone hardware to assist virtualization. We demonstrate how TrustZone can be adequately exploited for meeting the real-time needs, while presenting a low performance cost on running unmodified rich operating systems. While ARM continues to spread TrustZone technology from the applications processors to the smallest of microcontrollers, it is undeniable that this technology is gaining an increasing relevance. Our intent is to encourage research and drive the next generation of TrustZone-assisted virtualization solutions.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

Keywords and phrases hypervisor, virtualization, TrustZone, space and time partitioning, real-time, embedded systems

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.4

1 Introduction

Platform virtualization, which enables multiple operating systems (OSes) to run on top of the same hardware platform, is gaining momentum in the embedded systems arena, driven by the growing interest in consolidating and isolating multiple and heterogeneous environments [6]. While in industrial control or automotive systems virtualization has been used to integrate real-time control functionalities with high-level or infotainment environments [20, 9], in aeronautics and aerospace virtualization provides isolation for safety-critical components

* This work has been supported by COMPETE: POCI-01-0145-FEDER-007043 and FCT – Fundação para a Ciência e Tecnologia – (grant SFRH/BD/91530/2012 and UID/CEC/00319/2013).

[10, 26]. Despite the differences among several embedded industries, all share an upward trend for integration, due to the common interest in building systems with reduced size, weight, power and cost (SWaP-C) budget [6, 10].

Typically, solutions for embedded virtualization [10, 1, 7, 26] follow two different approaches: full-virtualization and paravirtualization. Between both approaches there is a trade-off between performance and flexibility: the traditional full-virtualization [7, 26] incurs on a higher performance cost, while the static paravirtualization approach [1, 10, 26] incurs on a higher design cost. Recently, due to penalties incurred by software-based virtualization approaches, as well as the strict timing requirements and constraints imposed by real-time virtualization [31], academia and industry have recently begun focusing their attention in providing hardware support to assist virtualization. Intel introduced Intel Virtualization Technology (VT) [24], ARM presented ARM Virtualization Extensions (VE) and ARM TrustZone [28, 4, 5, 17], and, recently, Imagination/MIPS released MIPS Virtualization and OmniShield technhology [31].

Among existent COTS technologies, ARM VE and ARM TrustZone [30] have attracted particular attention, due to the ubiquitous adoption of ARM-based processors in the embedded market. Although ARM VE is the specific technology from ARM for virtualization, ARM TrustZone is also seen as a hardware-based alternative for system virtualization [5]. This technology is gaining momentum due to the supremacy and lower cost of TrustZone-enabled processors in comparison with VE-enabled processors, and because it is seen as the only implementable hardware-based approach on ARM processors where VE are not available. Examples of such processors include the well-established ARM Cortex-A9, and the newest Cortex-A32. Furthermore, due to the recent ARM announcement of introducing TrustZone technology in the new generation of Cortex-M processors [27], this technology also promises to be a game-changer in the low-end sector, opening the possibility of breaking the barrier to the adoption of system virtualization in resource-constrained embedded devices.

TrustZone technology virtualizes a physical core as two virtual cores, providing two completely separate execution domains. The non-secure world acts as a virtual machine (VM) under the control of a hypervisor running in the secure world side. Some TrustZone-based solutions for virtualization have been proposed [30, 3, 5, 22, 13, 17]. While some of them just support a single guest execution, others present a dual-OS configuration for running an RTOS side-by-side with a GPOS. The problem is that they still lack in providing detailed information about their implementation and deployment on physical platforms, as well as in performing extensive experiments and presenting convincing results. We believe that ARM TrustZone, when adequately exploited, opens up a number of opportunities for (real-time) virtualization, despite some researchers still arguing that perceiving TrustZone as a virtualization mechanism is very limiting and ill-guided [28, 8].

To give answers to a plethora of doubts and questions we developed LTZVisor (Lightweight TrustZone-assisted Hypervisor) as a tool to clearly understand and evaluate how TrustZone hardware can be efficiently exploited to assist virtualization. We describe all the details behind the implementation, highlighting its benefits and discussing identified limitations and how they can be overcome. We conducted an extensive set of experiments which clearly demonstrate how TrustZone-assisted virtualization can effectively meet real-time needs. LTZVisor is the outcome of years of our experience in working and developing TrustZone-based solutions for a multitude of applications and domains [17, 16, 18, 19, 15]. The amount of open-source software for TrustZone systems is scarce. We plan to make LTZVisor available for the open-source community, encouraging research, whilst providing the foundation to drive the next generation of TrustZone-assisted virtualization solutions.

1.1 Contributions

In this paper, we present LTZVisor with the following contributions:

- an open-source tool to understand, evaluate, and encourage research towards TrustZone-assisted virtualization;
- an extensive evaluation over a physical hardware platform with popular benchmark suites, focusing on the penalties incurred on the real-time properties of the secure guest OS, as well as on performance of the non-secure guest OS;
- a complete discussion about the identified drawbacks and advantages of using TrustZone for (real-time) virtualization, and how we suggest to overcome those limitations based on the knowledge and expertise consolidated over the years.

2 ARM TrustZone

TrustZone technology [29] refers to the security extensions introduced with ARMv6K in all ARM Cortex-A processors. The TrustZone hardware architecture can be seen as a dual-virtual system, partitioning all system's physical resources into two isolated execution environments. Recently, ARM also decided to extend TrustZone for the Cortex-M processor family [27]. TrustZone for ARMv8-M has the same high-level features as TrustZone for applications processors, with the benefit that context switching between both worlds is done in hardware for faster transitions. In the remainder of this section, when describing TrustZone, we are focused on the specificities of this technology for Cortex-A processors. The distinctive aspects of TrustZone for ARMv8-M are out of the scope of this paper.

At the processor level, the most significant architectural change is its partition into two separate worlds: the secure and the non-secure worlds. A new 33^{rd} processor bit, the *Non-Secure* (*NS*) bit, accessible through the *Secure Configuration Register* (*SCR*) register, indicates in which world the processor is currently executing, and is propagated over the memory and peripherals buses. To preserve the processor state during the world switch, TrustZone adds an extra processor mode: the monitor mode. The monitor mode is completely different from other supported modes, because when the processor runs in this mode the state is always considered secure, independently of the NS bit state. Software stacks in the two worlds can be bridged via a new privileged instruction – *Secure Monitor Call* (*SMC*). The monitor mode can also be entered by configuring it to handle IRQ, FIQ, and Aborts exceptions in the secure world. To ensure a strong isolation between secure and normal states, some special registers are banked, such as a number of *System Control Coprocessor* (*CP15*) registers. Some secure critical processor core bits and *CP15* registers are either totally unavailable to non-secure world or access permissions are closely under supervision of the secure world. The TrustZone Address Space Controller (TZASC) and the TrustZone Memory Adapter (TZMA) extend TrustZone security to the memory infrastructure. TZASC can partition the DRAM into different memory regions: this hardware controller has a programming interface, accessible only from the secure side, that can be used to configure a specific memory region as secure or non-secure. By design, secure world applications can access normal world memory but the reverse is not possible. TZMA provides similar functionality but for off-chip ROM or SRAM. The TrustZone-aware Memory Management Unit (MMU) provides two distinct MMU interfaces, enabling each world to have a local set of virtual-to-physical memory address translation tables. The isolation is still available at the cache-level, because processor's caches have been extended with an additional tag which signals in which state the processor accesses the memory. System devices can be dynamically configured as secure or non-secure through the TrustZone Protection Controller (TZPC). To

support the robust management of secure and non-secure interrupts, the Generic Interrupt Controller (GIC) provides both secure and non-secure prioritized interrupt sources.

3 LTZVisor: Design

The main design idea behind LTZVisor is the use of TrustZone hardware to assist virtualization. The key towards TrustZone-assisted virtualization is to rely on hardware support as much as possible, while containing software implementation and components privileges, and promoting the secure environment with a higher privilege of execution. This leads to three fundamental principles:

- *The principle of minimal implementation:* Spaghetti code is the main source of vulnerabilities in software and provides an avenue of exploitation for hackers. Relying on the hardware support of TrustZone technology for virtualization as much as possible, as well as promoting the careful design and static configuration of each hypervisor component, will definitively help us minimize the trusted computing base of the system and, consequently, contain the attack surface.
- *The principle of least privilege:* Components must be given access only to those resources (e.g., I/O devices, system services, etc) that are absolutely required. TrustZone technology guarantees, by design, that the non-secure world is always less privileged than the secure one, despite the CPU execution mode. Furthermore, in the secure world, the monitor mode introduces a third level of privileges. Exploring these features to implement a well-layered virtualization approach will help promoting privileged execution and hardware-enforced isolation of the real-time environment from the non-real-time one.
- *The principle of asymmetric scheduling:* Virtualization of a real-time environment is very challenging, mainly due to strict timing requirements and hierarchical scheduling problems that those systems introduce. The adoption of an asymmetric scheduling policy, where the secure environment has a higher privilege of execution than the non-secure one, will ensure that timing requirements are met, even while executing real-time tasks over the RTOS running on top of a virtual CPU.

3.1 General Architecture

LTZVisor provides a virtualization solution based on the two virtual execution environments provided by the TrustZone hardware. The secure world is responsible for hosting the privileged software, while the non-secure world is responsible for hosting the non-privileged software. Figure 1 depicts the proposed virtualization architecture. In this figure, three main software components can be identified: the hypervisor, the secure VM, and the non-secure VM.

LTZVisor runs in the highest privileged processor mode, i.e., in monitor mode. When running in this mode, the processor state is considered always secure. The hypervisor has full control of all hardware and software resources, and is responsible for configuring memory, interrupts and devices assigned to each VM, as well as managing the Virtual Machine Control Block (VMCB) of each VM during a partition switch. When a virtual machine is about to be executed by the physical processor, the hypervisor transfers the VM state, saved on the respective VMCB, to the physical processor context. When the hypervisor assigns the physical processor to another virtual machine, the processor context of the active VM is saved back into the respective VMCB.

The secure VM runs in the supervisor mode of the secure world side. This VM needs to have a small footprint, because when the processor state is secure it has full view over the

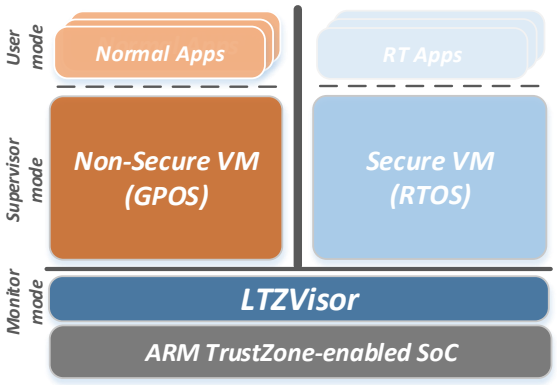


Figure 1 LTZVisor General Architecture.

non-secure world side. As such, the privileged guest code can interfere with the other virtual machine, by accessing or modifying its state or the state of its resources (memory or memory mapped devices). For this reason, the operating system hosted on the secure VM must be aware of the virtualization, and is considered part of the system's Trust Computing Base (TCB). The secure VM is ideal to run an RTOS, because the higher privilege of execution helps meeting the timing requirements of such environments. Furthermore, RTOSes typically have small memory footprints.

The non-secure VM runs in the supervisor mode of the non-secure world side. This VM is ideal to host a general purpose guest OS, useful for running human-machine interfaces as well as internet-based applications and services. The software running on the non-secure world side is completely isolated from the privileged software running on the secure world side. When the processor is operating in a privileged mode but not in the secure state, it cannot access nor modify any state information belonging to the secure world. Any attempt from the non-secure guest OS to access any resource of the secure world side immediately triggers an exception to the hypervisor.

4 LTZVisor: Implementation

LTZVisor exploits ARM TrustZone to provide time and space isolation between both partitions. The asymmetric design principle allows to preserve the real-time characteristics of the secure virtual machine (RTOS). This section provides all the details behind LTZVisor implementation, describing how CPU virtualization and memory isolation is ensured, presenting how MMU and caches are managed, describing how device partition is achieved, explaining how interrupts and time are managed for different guest OSes, and illustrating how inter-VM communication is implemented.

4.1 Virtual CPU

TrustZone technology virtualizes each physical CPU into two virtual CPUs: one for the secure world and another for the non-secure world. Between both worlds there is a list of banked registers, i.e., an individual copy of those registers exists for each world. Since each guest OS is running in a different world, in this particular case, a huge part of the virtual CPU support is guaranteed by the hardware itself, minimizing the number of registers to be saved and restored in each partition-switching operation. The VMCB of the non-secure

side is composed by 27 registers: 13 *General Purpose Registers* (*R0-R12*), the Stack Pointer (*SP*), the Linker Register (*LR*) and *Saved Program Status Register* (*SPSR*) for each of the following modes: Supervisor, System, Abort and Undef. The "high" *General Purpose Registers* (*R8-R12*), as well as the *SP*, *LR* and *SPSR* of the FIQ and IRQ modes are not included, as they are mutually exclusive for each world. Among the coprocessor registers, almost all of them are banked: only the *SCTLR* and the *ACTLR* need to be preserved. For optimization purposes, the VMCB of the secure side is composed of only 16 registers: 13 *General Purpose Registers* (*R0-R12*), the Stack Pointer (*SP*), the Linker Register (*LR*) and *SPSR* for the System mode. The Monitor mode is, by design, uniquely dedicated to the secure world side. These optimizations reduce the interrupt latency from the secure guest OS (RTOS) perspective, speeding up the transition from the non-secure to the secure world side, when a secure interrupt arises while the non-secure OS is executing.

Among the aforementioned unbanked registers, there are those which are only modifiable from the secure side: they can be read when the processor is in the non-secure state, but an attempt to modify them will be ignored. This is stated on TrustZone specification to guarantee a high degree of security in the system, which has a cost for the non-secure guest OS. For example, the *System Control Register* (*SCTLR*) and the *Auxiliary Control Register* (*ACTLR*) provide control and configuration over memory, cache, MMU, AXI accesses, etc. These registers are used to enable and disable MMU, and are only accessible in the secure state. During the non-secure guest OS boot process, an attempt to modify them will be ignored, leading the GPOS to get stuck. For that reason, the hypervisor must fill some registers of the non-secure VMCB with a specific initialization value. For example, the *SCTLR* register of the non-secure VMCB should be initialized appropriately, so that MMU and Level 1 cache of the non-secure world side are enabled before the GPOS starts booting.

4.2 Scheduler

An identified issue in virtualizing a real-time environment is the well-known hierarchical scheduling problem. Typically, a hypervisor schedules virtual CPUs while a guest RTOS running over the virtual CPU schedules its own tasks. Ensuring real-time execution of tasks over the RTOS executing on top of a virtual CPU involves a complex hierarchical scheduling analysis, requiring that both schedulers are accordingly modeled [31].

LTZVisor overcomes this problem by implementing an asymmetric or idle scheduler. This scheduling policy guarantees that the non-secure guest OS is only scheduled during the idle periods of the secure guest OS, and the secure guest OS can preempt the execution of the non-secure one. In fact, the secure virtual machine (RTOS) has a higher scheduling priority than the non-secure one, and LTZVisor is not the software component that directly schedules the virtual machines, but it is scheduled itself by the secure guest OS. Although this can seem contradictory, the concept of ring protection is never jeopardized, as the LTZvisor continues executing in a more privileged mode than the secure guest OS: the hypervisor is just configured to behave in a passive way.

4.3 Memory Partition

Traditional hardware-assisted memory virtualization relies on Memory Management Unit (MMU) support for 2-level address translation, mapping guest virtual to guest physical addresses and then guest physical to host physical addresses. This MMU feature is a key enabler to run unmodified partition OSes, and also to implement isolation between partitions.

TrustZone-enabled SoCs (which are not VE-enabled) only provide MMU support for single-level address translation. Therefore, the existence of a TZASC is a major requirement

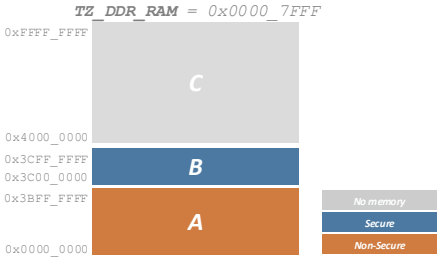


Figure 2 LTZVisor: Memory Configuration.

for the proposed solution, because this component allows partition of memory into different segments. This memory segmentation feature can be exploited to guarantee spatial isolation between the non-secure VM and the secure one, basically by adequately configuring the security state of the memory segments of respective partitions. The non-secure VM should have its own memory segment(s) configured as non-secure, and the remaining memory as secure. If the non-secure guest OS tries to access a secure memory region (either belonging to the secure partition or the hypervisor), an exception is automatically triggered and the execution control redirected to the hypervisor.

Memory segments can be configured with a specific granularity, which is implementation defined, depending on the vendor. In the hardware under which our system was deployed, Xilinx ZC702, memory regions can be configured with a granularity of 64MB. This configuration is provided via a system level control register named `TZ_DDR_RAM`. A 0 or 1 on a particular bit indicates a secure or non-secure memory region for that particular memory segment, respectively. Figure 2 depicts the memory setup and respective secure/non-secure mappings, for a virtualized system consisting of the hypervisor altogether with the secure virtual machine (B), and the non-secure virtual machine (A). In this specific configuration, the non-secure VM (GPOS) uses the first fifteen memory segments (0x00000000 – 0x3BFFFFFF), corresponding to a total of 960MB of non-secure memory. The hypervisor and the secure VM, due to their low memory footprint, use only the last available memory segment (0x3C000000 – 0x3FFFFFFF), corresponding to a 64MB of secure memory. The remainder of the 32-bit memory address space is not accessible (C), because Xilinx ZC702 is only endowed with a 1GB DDR3 memory.

4.4 MMU and Cache Management

The TrustZone-aware Memory Management Unit (MMU) provides two distinct MMU interfaces, enabling each world to have a local set of virtual-to-physical memory address translation tables. This means each world has its own copy of the TTBR register set, as well as an independent MMU configuration. This reduces the list of activities to perform on each guest-switching operation, because translation lookaside buffer (TLB) entries do not need to be invalidated.

The same kind of isolation is still available at cache-level. The processor caches have been extended with an additional tag (NS bit) which records the security state of the transaction that accesses the memory. This NS bit is set by hardware and it is not directly accessible by system software. Therefore, in this cache coherence design, when the system switches between the two worlds, none of the cache lines need to be flushed. This means that this design feature at cache-level significantly improves the performance of LTZVisor, because no cache management operation needs to be performed on each guest-switching operation: cache

isolation is enforced and guaranteed by the hardware itself. On Xilinx ZC702, there are a few notes regarding the TrustZone support in L2 cache. The L2 Control register (`reg1_control`) can only be written with an access tagged as secure, which means that an attempt to enable or disable the L2 cache from the non-secure world side will be ignored. Similarly to the support that the hypervisor needs to perform in the L1 cache initialization (aforementioned in Section 4.1), LTZVisor also needs to enable the L2 cache on the secure world side before the non-secure guest OS starts booting. Once the L2 cache is enabled, maintenance operations on the non-secure entries can be performed directly from the non-secure world side.

4.5 Device Partition

TrustZone technology allows devices to be (statically or dynamically) configured as secure or non-secure. This hardware feature allows the partition of devices between both worlds while enforcing isolation at the device level.

LTZVisor implements device virtualization adopting a pass-through policy, which means devices are managed directly by guest partitions. To ensure strong isolation between them, devices are not shared between guests and are assigned to the respective partitions at design time, and then configured during boot time. The devices assigned to the RTOS (secure VM) are configured as secure devices, while devices assigned to the GPOS (non-secure VM) are configured as non-secure devices. This guarantees the GPOS cannot compromise the state of any device belonging to the RTOS, and if the non-secure guest partition tries to access a secure device then an exception will be automatically triggered and handled by hypervisor. On Xilinx ZC702, the security state of devices can be configured through a set of secure registers accesible from the secure side. This registers include, for example, the SDIO slave security registers (`security2_sdio0` and `security3_sdio1`) and the APB slave security register (`security6_apb_slaves`).

4.6 Interrupt Management

In TrustZone-enabled SoCs, the GIC supports the coexistence of secure and non-secure interrupt sources. It also allows the configuration of secure interrupts with a higher priority than the non-secure interrupts, and has several configuration models that enable the assignment of IRQs and FIQs to secure or non-secure interrupt sources.

LTZVisor configures interrupts of secure devices (i.e., secure interrupts) as FIQs, and interrupts of non-secure devices (i.e., non-secure interrupts) as IRQs. A TrustZone-enabled GIC permits all implemented interrupts to be individually defined as Secure or Non-secure, through the *Interrupt Security Registers* set (*ICDISRn*). To program secure interrupts to use the FIQ interrupt mechanism of the processor, the *FIQen* bit in the *CPU Interface Control Register* (*ICPICR*) must be set. When the secure guest OS (i.e., RTOS) is under execution, secure interrupts (i.e., FIQs) are redirected to the RTOS without hypervisor interference, guaranteeing that no overhead is added to the interrupt latency of the secure guest OS. This can be done by disabling the *FIQ* bit into the *Secure Configuration Register* (*SCR*). If an IRQ (i.e., an interrupt for the GPOS partition) arises while the RTOS is executing, it doesn't affect the expected RTOS behavior. As soon as the non-secure guest becomes active, the interrupt will be then processed. The prioritization of secure interrupts prevents a denial-of-service attack against the secure side (from the GPOS partition). From a different perspective, when the non-secure guest OS (i.e., GPOS) is executing and an FIQ (i.e., an interrupt for the RTOS partition) arises, the execution flow is immediately redirected to the hypervisor, which will be responsible for handling the interrupt directly in monitor mode.

This design decision minimizes the interrupt latency from the RTOS perspective, ensuring the interrupt is attended as soon as possible. On the other hand, if an IRQ arises, it will be directly managed by the non-secure guest. Non-secure interrupts are always signaled (by design) using the IRQ mechanism of the processor.

4.7 Time Management

Temporal isolation in virtualized systems is typically achieved using two levels of timing: at hypervisor level and at partition level. For the partition level, hypervisors typically provide timing services which allow guests to have notion of virtual or real time. In the first case, each time a partition is inactive, the time is paused, and once the guest is rescheduled, the timekeeping is resumed. In the second case, when the partition is paused, the hypervisor is responsible for keeping track of the wall-clock time, and, once resumed, update the partition timing structures.

LTZVisor provides a distinctive time management implementation. Due to its dual-OS configuration, as well as the intrinsic design principle of asymmetric scheduling, the hypervisor dedicates one independent timing unit for each guest OS. The secure VM uses the Triple Timer Counter (TTC) 0, while the non-secure VM uses the TTC1. It is fundamental that the hypervisor configures TTC1 as a non-secure device, otherwise an exception will be triggered on the first attempt to access it. This specific time management implementation ensures that each VM has its timing structures updated at all times. The RTOS does not miss any system-tick interrupt, and the GPOS, as a tickless OS, is completely aware of the real passage of time.

4.8 Inter-VM Communication

Inter-VM communication provides a transparent virtual mechanism for implementing communication between different VMs. In contrast with other solutions, which follow an non-standard approach [24, 22, 26], LTZVisor uses the standardized VirtIO [21] as a transport abstraction layer. VirtIO has been used in several implementations targeting I/O virtualization [21, 4], and has recently started being adopted to implement inter-guest [14] and inter-processor communication on multicore platforms (e.g., Texas Instrument RPMsg and Mentor Graphics MEMF) [2].

LTZVisor implements an adaptation of the RPMsg API from the Texas Instrument and OpenAMP group to a supervised single-core architecture. The implementation from Texas provides the foundation for implementing communication on top of the GPOS, while the implementation from OpenAMP provides the foundation for a bare-metal approach. The main modifications encompass: (i) the complete elimination of the Remoteproc executable loader and processor life cycle management, since it is supported by the hypervisor; (ii) the VirtIO device Remoteproc configuration implemented through a static approach (at compile-time); and (iii) RPMsg slave mode support following the VirtIO standard. Figure 3 depicts the communication architecture. As it can be seen, the data path is completely isolated by the event path, a design decision that promotes asynchronous communication, essential to guarantee the timing requirements of the secure VM. The data path is defined by a shared block of memory, configured as non-secure. The event path is defined by software generated interrupts (SGIs) routed through the hypervisor. This mechanism is based in requests from guest OSes to the hypervisor, via the SMC instruction. All requests are stored in a circular buffer, following a first-in, first-out policy. During each partition switch, LTZVisor triggers SGIs to the respective guest OSes, enabling asynchronous notifications. In spite of the

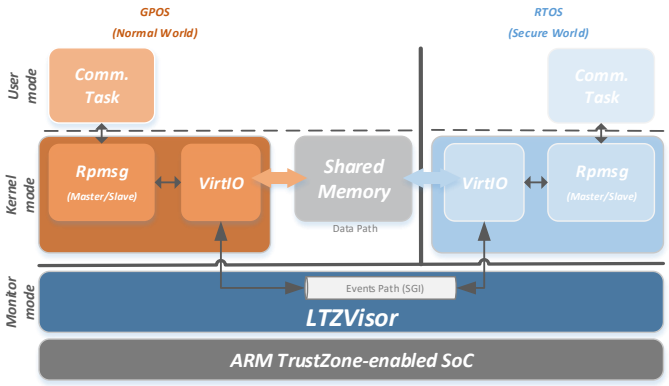


Figure 3 LTZVisor: Inter-VM Communication.

imposition of a slight increase to the partition-switching time, this trade-off guarantees the reliability of the communication as the hypervisor has control over every transaction.

5 Evaluation

LTZVisor was evaluated on a Xilinx ZC702 evaluation board targeting a dual ARM Cortex-A9 running at 667MHz. In spite of using a multicore hardware architecture, the evaluated implementation only supports a single-core configuration. Our evaluation focused on three metrics: (i) memory footprint, (ii) performance overhead and (iii) interrupt latency. LTZVisor and both OS partitions were compiled using the ARM GNU toolchain, with compilation optimizations disabled (-O0). The idea of presenting results with compilation optimizations disabled is because it represents the worst case scenario. Linaro Linux (v3.3.0) and FreeRTOS (v7.0.2) were used as non-secure and secure partitions, respectively. MMU, data and instruction cache and branch prediction were disabled on the secure world side.

5.1 Memory Footprint

In order to assess the memory footprint of each software component of the implemented architecture we used the size tool of the ARM GNU Toolchain. We evaluated LTZVisor, as well as the native, modified and virtualized version of FreeRTOS. Table 1 presents the collected measurements, where boot code, libraries and drivers were not taken into consideration. As it can be seen, the memory overhead introduced by the hypervisor is really small, i.e., 2880 bytes. The main reasons behind such a low memory footprint are related to the principle of minimal implementation followed during LTZVisor design which relies on (i) the hardware support of TrustZone technology for virtualization and (ii) the careful design and static configuration of each hypervisor component. The native version of FreeRTOS, supporting IRQ, requires 18882 bytes, the modified version, supporting FIQ, requires 18898, and the virtualized version requires 18918 bytes. From the native version to the modified one there is a slight increase of 0.8% in the memory footprint, while from the native version to the virtualized one there is an increase of 1.9%. This slight increase is completely acceptable and encompasses small modifications and adaptations for FIQ and context-switch handling (from native to modified), and in the FreeRTOS scheduler (from modified to virtualized).

Table 1 LTZVisor memory footprint (bytes).

Software	*Memory Footprint*			
	.text	*.data*	.bss	Total
LTZVisor	2368	0	512	2880
FreeRTOS IRQ (v7.0.2)	17942	20	920	18882
FreeRTOS FIQ (v7.0.2)	17954	20	924	18898
vFreeRTOS FIQ (v7.0.2)	17974	20	924	18918

5.2 Performance

The performance evaluation process was split into three different test case scenarios. First, LTZVisor was evaluated for specific micro-operations of the guest-switching operation. Then, we evaluated the virtualization overhead (using the Thread Metrics Suite) as well as the interrupt latency over the secure VM (RTOS). Finally, we assessed the virtualization overhead over the non-secure VM (GPOS) using the LMBench3 Suite.

5.2.1 Partition context switching

To evaluate the guest context switch time we used the Performance Monitor Unit (PMU) component. To measure the time consumed by each internal activity of a round-trip world switch, a PMU-specific instruction was added at the beginning and end of each code portion to be measured. Results were gathered in clock cycles and converted to microseconds accordingly to the processor's frequency (667MHz). The values represent the average and the standard deviation of 1000 collected samples.

The list of internal activities to perform a full switch between secure to non-secure and non-secure to secure worlds are:

1. *SMC handling.* The secure guest OS schedules the idle task. The idle task performs a secure call that is responsible for invoking the hypervisor (*SMC*). Time since the processor enters in the monitor's vector table until LTZVisor completes the SMC handling;
2. *Save secure guest OS context.* LTZvisor handles the SMC request and saves the context of the secure guest OS. Time to save the current state of the secure guest OS to its respective VMCB;
3. *Restore non-secure guest OS context.* LTZvisor saves the context of the secure guest OS and then restores the context of the non-secure guest OS. Time to restore the state of the non-secure guest OS from its respective VMCB;
4. *FIQ acknowledge.* The non-secure guest OS is running while a secure interrupt is triggered (e.g., RTOS timer tick). Time since the processor enters in the monitor's vector table until LTZVisor acknowledges the FIQ;
5. *Save non-secure guest OS context.* LTZvisor acknowledges the FIQ request and then saves the context of the non-secure guest OS. Time to save the current state of the non-secure guest OS to its respective VMCB;
6. *FIQ handling.* LTZvisor saves the context of the non-secure guest OS and then immediatly handles the FIQ request. Time since the hypervisor save the current state of the non-secure guest OS untill LTZVisor completes the FIQ handling;
7. *Restore secure guest OS context.* LTZvisor handles the FIQ and then restores the context of the secure guest OS. Time to restore the state of the secure guest OS from its respective VMCB.
8. *Scheduler.* LTZvisor restores the execution of the RTOS. The RTOS continues executing the idle task loop and verifies if there are real-time tasks to run. If not, the idle task performs a system call (*SMC*) that is responsible for invoking the hypervisor. Time since the processor restores the idle task execution until it enters in the monitor's vector table.

Table 2 LTZVisor performance statistics (clock cycles).

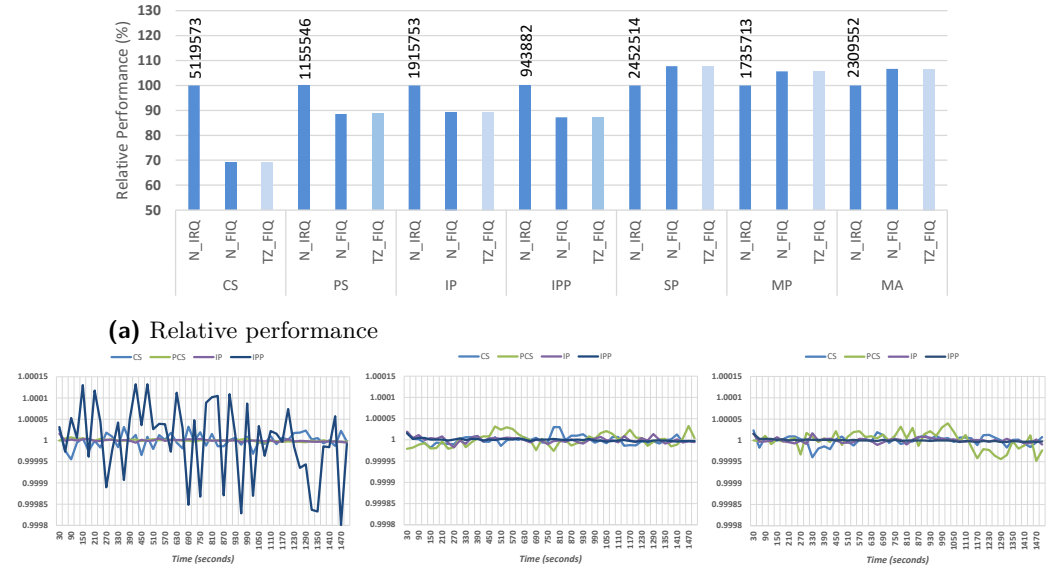
World Switch	*Operation*	*Performance*		*Time*
		$\overline{x}$	s	@667MHz
Switch to NS world	*(1) SMC handling*	571	0.943	$856ns$
	(2) Save S guest OS context	422	1.274	$633ns$
	(3) Restore NS guest OS context	949	2.324	$1423ns$
Switch to S world	*(4) FIQ acknowledge*	467	0.614	$700ns$
	(5) Save NS guest OS context	982	1.321	$1472ns$
	(6) FIQ Handling	1648	47.367	$2471ns$
	(7) Restore S guest OS context	243	0.524	$364ns$
Scheduler	*(8) Assymetric Policy*	7542	9.316	$11307ns$

Table 2 presents the collected results. As it can be seen, the complete partition-switch operation takes around 19.23 microseconds. This value assumes there are no real-time tasks ready to run once the RTOS is rescheduled. The process of verifying, from the RTOS execution, there are no real-time tasks to run and, hence, trigger the switch to the non-secure world takes around 11.31 microseconds. The process of switching from the RTOS to the GPOS takes just 2.91 microseconds, and is the most deterministic activity of the partition-switching operation. Our experiments demonstrated less than four clock cycles of deviation from the average value (for each individual activity). Once the GPOS is executing and a FIQ is triggered, the hypervisor ensures a 2.17 microseconds of interrupt latency, and then in a further 2.84 microseconds the RTOS is restored. The FIQ handling operation is the major source of non-determinism of the partition-switching operation. We strongly believe the reason is related with the required accesses to the peripheral bus when handling the interrupt request (in this specific case, the system tick timer).

5.2.2 Secure VM (RTOS)

The Thread-Metric Benchmark Suite consists of a set of benchmarks properly conceived to evaluate RTOSes performance. The suite comprises 7 benchmarks, evaluating the most common RTOS services and interrupt processing: cooperative scheduling (CS); preemptive scheduling (PS); interrupt processing (IP); interrupt preemption processing (IPP); synchronization processing (SP); message processing (MP); and memory allocation (MA). Each benchmark outputs a counter value, representing the RTOS impact on the running application: the higher the value, the smaller the impact.

Benchmarks were executed in the native version of FreeRTOS (N_IRQ), where interrupts are handled as IRQs, in a modified version of FreeRTOS, where interrupts are handled as FIQs (N_FIQ), and then compared against the virtualized version (TZ_FIQ). Figure 4a presents the achieved results, corresponding to the average relative performance (as well as the average absolute performance) of 1000 collected samples for each benchmark. Each sample reflects the benchmark score for a 30 seconds execution time, encompassing a total execution time of 500 minutes, per benchmark. In accordance with Figure 4a the execution of the modified version of FreeRTOS (N_FIQ) is very dependent from the benchmark. In some cases, the performance decreases, while in others the performance increase. The increase of performance on the modified version of FreeRTOS is completely understandable since FIQ interrupts present low hardware latency than IRQs, but the decrease is apparently strange. The reason behind this phenomenon is related with an adaption we did on the yield macro of FreeRTOS. The native version of FreeRTOS implements the yield through the use of the SVC exception. When a SVC is triggered a context-switch happens and the IRQ bit of the CPSR is set, so that there is no preemption during the execution of the critical routine



(a) Relative performance

(b) Variation: Native (IRQ) **(c)** Variation: Native (FIQ) **(d)** Variation: Virtualized (FIQ)

Figure 4 Thread-Metric benchmarks results.

(atomic execution). Thus, the modification of FreeRTOS for handling interrupts as FIQs should include the modification of the context-switch function to set the FIQ bit, instead of the IRQ bit. The problem is in the ARMv7-A specification, this bit is implementation defined. In the case of Xilinx Zynq, for security reasons, this bit is read-only, and only changes when triggered by hardware (e.g. when a FIQ happens). For this reason, we were forced to change the yield function to use an SGI (FIQ) instead of the SVC exception. The SGI has a higher latency than the SVC, which, on yield-intensive tests (i.e., the case of CS, PS, IP and IPP), this translates in a decrease of performance. It should be noted this is platform- and worlkload-specific problem that does not necessarily mean it can occur in other platforms and be noticeable in real application scenarios. In fact, the overhead introduced by LTZVisor is null, as demonstrated by the comparison of the N_FIQ and TZ_FIQ versions. This is perfectly understandable because, once FreeRTOS starts running real-time tasks, it will never be interrupted by the hypervisor. Regarding the variation, Figure 4b, Figure 4c and Figure 4d present the normalized variation of the collected results over time for the native, modified and virtualized versions of FreeRTOS, respectively. It is clear that the use of FIQ for handling interrupt sources slightly reduces the variation of results, and variation in the virtualized system is also in the same order of magnitude as the modified version, which means the virtualized system remains as deterministic as the (modified) native one. In sum, the asymmetric scheduling policy gives the RTOS a higher execution privilege, so it can preserve its real-time characteristics. Furthermore, the necessity of handling interrupts as FIQs promotes a deterministic execution, and most of the cases can either increase performance.

Interrupt latency is the measurement of system's response-time to an interrupt, which corresponds to the elapsed time between interrupt assertion and the instant when a response happens. Equation 1 expresses the system latency: τ_H is the hardware dependent time which depends on the interrupt controller, on the board, as well as the type of the interrupt; τ_{OS} is

the OS-specific induced overhead; and τ_{HYP} is the hypervisor-specific induced overhead.

$$\tau_{IL} = \tau_H + (\tau_{HYP}) + \tau_{OS} \quad (1)$$

Experiments showed that the latency in the native system (FreeRTOS) is 0.89 microseconds, which corresponds to the average interrupt handling overhead of our system, because when the secure guest OS is executing the FIQ requests are directly forward to the RTOS. The τ_{HYP} expression of Equation 1 represents the extra overhead induced by our approach, which only occurs when the RTOS has no real-time ready-to-run tasks, and consequently the hypervisor is invoked to perform a world switch. Since LTZVisor runs with all interrupt sources disabled, the worst case scenario happens when an FIQ request (e.g., RTOS tick) arrives while a context switch from the secure to the non-secure world is starting. In this case, the request is handled with a worst case interrupt latency of 5.08 microseconds. This is a very sporadic situation that can happen under rare conditions, because two asynchronous and independent events need to occur at the same time: (i) an asynchronous FIQ needs to be triggered while (ii) a world switch is happening. Nevertheless, since the overhead introduced on latency has a deterministic upper bound (5.08 microseconds), it can be taken into account when designing the real-time system.

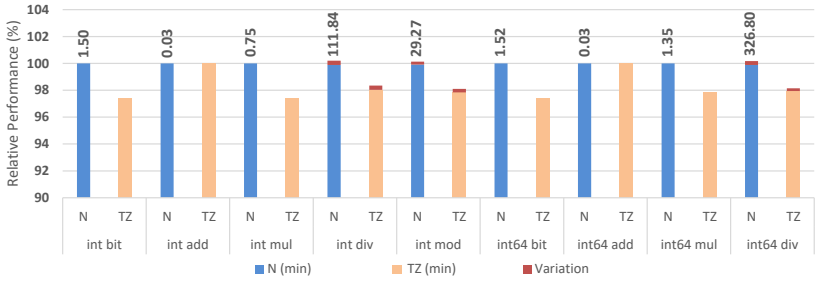
5.2.3 Non-Secure VM (GPOS)

LMBench [11] is a widely used suite of micro-benchmarks that measure a variety of important aspects of system performance, such as latency and bandwidth. The suite is written in portable ANSI-C using POSIX interfaces and targeting UNIX systems. The LMBench 3.0 suite includes more than fourthy micro-benchmarks within three different categories: *bandwith*, *latency*, and *other*. We focus our evaluation on two specific benchmarks, evaluating different architectural subsystems:

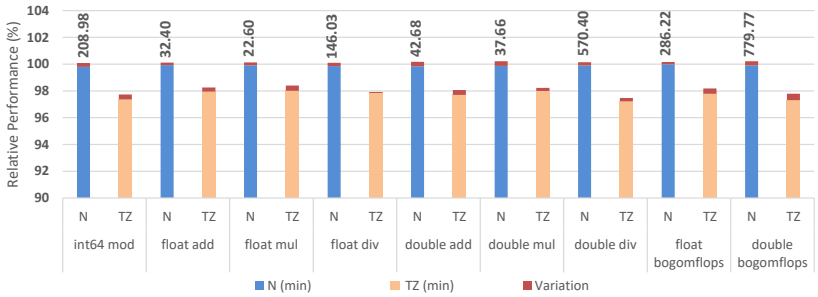
- *lat_ops:* Arithmetic operations latency, to evaluate general CPU performance (VFP and Neon are disabled);
- *bw_mem:* Memory operations bandwidth for different blocks size (2K, 128K, 4M), to evaluate the interference of the TZASC as well as Level 1 (4-way set-associative 32 KB) and Level 2 (8-way set-associative 512 KB) data caches.

For the first part of the experiment, FreeRTOS was configured with a 1 millisecond tick rate (i.e., guest-switching rate) and no real-time tasks were added to the system (i.e., the RTOS will be infinitely executing the idle task). We ran the micro-benchmarks in the native version of Linux (N) and compared them against the virtualized version (TZ). L1 and L2 caches and branch prediction were enabled for both test case scenarios. For each micro-benchmark, we performed 100 consecutive experiments. For each experiment the micro-benchmark was configured for 10 warm-ups and 1000 repetitions (–W 10 –N 1000). Presented results correspond to the average relative performance and variation (as well as the average absolute performance) of the 100 consecutive experiments, encompassing a total of 100000 samples (per bar).

Figure 5 presents the achieved results for the arithmetic operations latency benchmark. The values on top of the bars corresponds to the average latency, in nanoseconds. As it can be seen, the virtualized version of Linux only presents an average performance degradation of 2%, when compared with its native execution. This value is practically uniform among all micro-benchmarks (apart from the small variations due to the benchmark's lack of accuracy and the system's nonlinearities), except for the `int add` and `int64 add` cases. For these specific micro-benchmarks, the achieved results do not reflect the real performance penalty,



(a) *lat_ops* benchmark results (part 1)

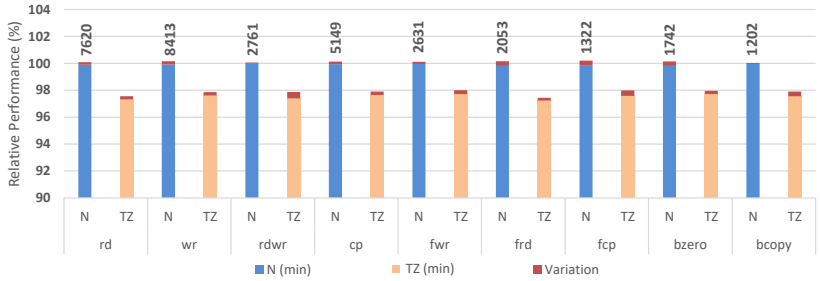


(b) *lat_ops* benchmark results (part 2)

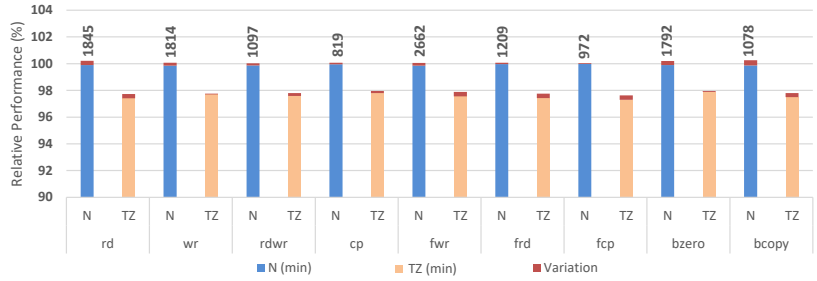
Figure 5 LMBench arithmetic operations latency (*lat_ops*) benchmark results.

due to the lack of precision. The assessed latency is 0.03 nanoseconds, and the minimal time unit is 0.01 nanoseconds. Regarding variation, it is clear the virtualized Linux presents a variation in the same order of magnitude as the native version. This means the virtualized system remains as deterministic as the native one.

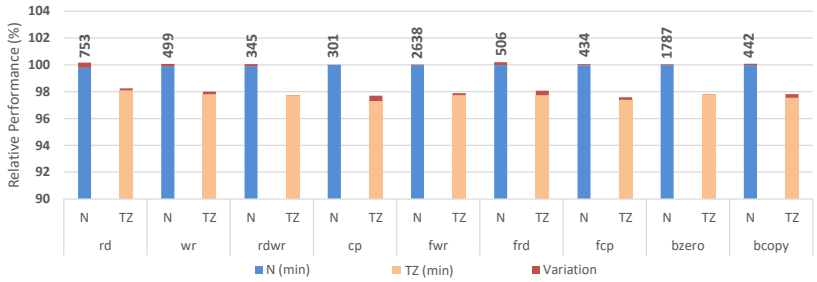
Figure 6 presents the achieved results for the memory bandwith benchmark. The values on top of the bars corresponds to the average memory bandwith, in megabytes per second (MB/s). Figure 6a, Figure 6b and Figure 6c depict the assessed results for a memory block size of 2KB, 128KB and 4MB, respectively. These memory block sizes were selected with the intention to fit and not within the L1 and L2 cache sizes. Looking at the three figures, it is clear the relative performance of the system is practically uniform among all micro-benchmarks, presenting an average performance degradation of 2% when comparing the virtualized Linux to the native one. Contrasting these values with the results presented in Figure 5, three main conclusions can be drawn: first, it is clearly noticed the effect of each cache on the accessed absolute memory bandwith results – the higher is the memory block size, the lower is the memory bandwith; second, cache isolation is in fact guaranteed by hardware, and does not introduce any extra overhead neither requires any cache maintenance operation on each guest-switch; and, finally, (memory) space isolation provided by means of the TZASC does not have associated any extra source of overhead. To corroborate the viability of our conclusions, experiments were performed without some of the hypervisor support (please refer to Sections 4.1 and 4.4) for caches and memory initialization. For example, one set of experiments were performed without the hypervisor having enabled L2 cache before booting the GPOS. The results were very straightforward: an abrupt decrease of performance, reaching almost 70% in some cases, happen for memory block sizes higher than 32KB and lower than 512KB. These experiments clearly demonstrates the effect of L2 cache in the overall system, as well as the coexistence of non-secure and secure cache entries without any cache maintenance support. Despite not being presented in Figure 6, due to



(a) *bw_mem* benchmark results (2KB)



(b) *bw_mem* benchmark results (128KB)



(c) *bw_mem* benchmark results (4MB)

Figure 6 LMBench memory bandwith (*bw_mem*) benchmark results.

shortage of space, we also performed a larger set of experiments encompassing memory block sizes of 16KB, 64KB and 1MB. The achieved results were identical to the ones presented in Figure 6, which reinforces the reliability of our conclusions.

For the second part of the experiment, instead of fixing the FreeRTOS tick with a 1 millisecond rate, the same experiments were repeated for eight different guest-switching rates within a time window between 100 microseconds to 10 milliseconds. Again, no real-time tasks were added to the system. We ran the arithmetic operations latency benchmark in the virtualized version of Linux. L1 and L2 caches and branch prediction were enabled for all test case scenarios. For each micro-benchmark we performed 100 consecutive experiments, and for each experiment the micro-benchmark was configured for 10 warm-ups and 1000 repetitions (–W 10 –N 1000). Presented results correspond to the average performance overhead of measured results for the 18 (arithmetic) micro-benchmarks, encompassing a total of 1800000 samples per test case scenario (per mark). Eight different test case scenarios were setup, corresponding to a tick rate of 100, 200, 300, 500, 1000, 2000, 5000 and 10000 microseconds, encompassing a cumulative number of 14400000 samples. Figure 7 presents the achieved results. The performance overhead ranges from 22.93% to 0.65% for a guest-switching rate of 100 and 10000 microseconds, respectively. For a system configured with a tick rate above 500 microseconds, the expected performance overhead is less than 5%.

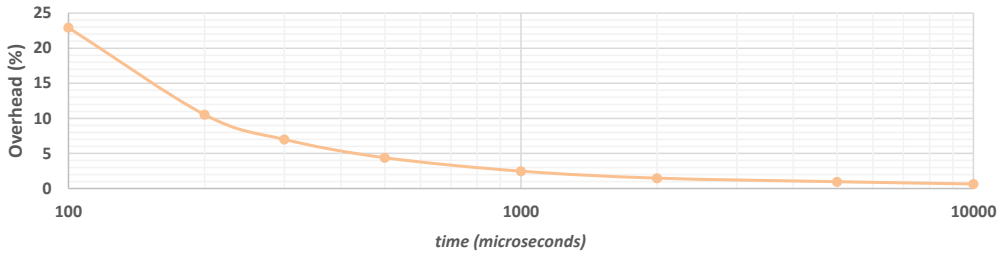


Figure 7 LTZVisor: guest-switching rate vs performance overhead.

6 Discussion

With LTZVisor we demonstrated how hardware enhancements introduced by TrustZone technology can be adequately exploited to assist virtualization, especially in the case of two virtual machines, because this number coincides exactly with the number of isolated states directly supported by the processor. We demonstrated and explained how several TrustZone features can be adequately exploited to run an RTOS side-by-side with a GPOS.

The asymmetric design principle, which dictates the secure VM has a greater scheduling priority than the non-secure one, ensures the timing requirements of the real-time environment remains nearly intact, at the cost of integrating the hypervisor with the RTOS on the secure world side. In doing so, the RTOS has full control over the system, and can access or modify the state of the non-secure VM. Recently, Ngabonziza et al. [12] presented some doubts about how our solution [17] could prevent the RTOS (secure world) from accessing the GPOS (non-secure world): in fact, it cannot; this is the price we pay to preserve the real-time demands of the system, while keeping performance acceptable for low-end and low-cost devices. Anyway, two possible solutions to guarantee a higher degree of isolation on high-end devices are: run all guest OSes in the non-secure world side, as demonstrated by our recent work [18]; or either paravirtualizing the RTOS, so that it can run in the user mode of the secure world side, and mediate each memory access through the hypervisor. Another point outlined by Ngabonziza et al. is related to guest OSes preemption and consequent starvation. They argue in our design "either OS cannot preempt the other OS". This is wrong; LTZVisor guarantees, by design, the secure guest OS (RTOS) preempts the non-secure guest OS (GPOS) as soon as a FIQ is triggered, but the reverse is not possible. So, starvation can happen, but only from the non-secure world side. However, despite this being a design decision to ensure the real-time needs, it is well-justified by the fact typical real-time applications have frequent idle times, which ensures the non-secure guest OS has enough CPU slices for execution. Ultimately, the scheduling policy can be designed accordingly to the applications needs, ensuring enough scheduling points that adequately meet the needs of both OSes, without compromising any real-time deadline; or either multicore platforms can be exploited to implement asymmetric multiprocessing (AMP) support, as we already did.

One of the main identified limitations is related to the number of supported virtual machines. LTZVisor supports the coexistence of two VMs, one running in the secure world and one running in the non-secure world. Although this is almost sufficient for a huge amount of current embedded real-time applications, some researchers still rely on this premise to consider TrustZone as an ill-guided virtualization technique. We demonstrated this is not completely true, and that is possible to overcome this limitation by multiplexing more guest OSes inside the non-secure world side [18, 19]. It requires careful handling of shared hardware resources, such as processor registers, memory, caches and MMU. Processor registers can

be easily saved and restored into/from a specific VMCB, while memory isolation can be achieved through the dynamic memory configuration feature of TZASC.

Spatial isolation is a major requirement for virtualization. LTZvisor implements memory isolation relying on the TZASC, which is an optional and implementation-specific component on TrustZone specification. The granularity of access restrictions depends on the SoC. Some outdated TrustZone-based SoCs are not equipped with this memory controller, and on many other the TZASC can only control some portions of the memory. For example, the Versatile Express platform provides no means to partition the DDR RAM into secure and non-secure areas. Nevertheless, when regarding the most modern TrustZone-based SoC, this is completely different, because they are totally equipped with fully featured TrustZone-aware memory controllers. This is the case of Xilinx Zynq SoCs and also the Freescale i.MX53 QSB. For example, Sun et al. [25] explains the use of the same functionality to create TrustICE, a framework that uses the hardware-assisted Watermark technique to dynamically protect the memory regions of the suspended secure code (ICEs).

Another identified limitation on the memory subsystem is related to non-existence of a second level memory translation. There is no way to virtualize the physical memory as used by the guest OSes. The guest-physical memory always corresponds to the host-physical memory, which means all guest OSes have to co-operate with respect to the address space being used, requiring relocation and consequent recompilation of the guest OS. This means the chance to use multiple closed-source guest OSes (only available as binary image) is very reduced, because different OS providers typically compile their software to run on the same memory address space of a specific platform. What is seen as a limitation to the system from a non-real-time perspective, is somewhat seen as an advantage from a real-time perspective. It is well-established the use of MMU and other components which introduce some non-linearities are seen with some scepticism regarding determinism and worst-case performance requirements of many real-time systems. An important argument that supports our vision is the recent decision of ARM in introducing support for virtualization in the new ARMv8-R architecture relying on a double-stage MPU [27]. In the ARMv8-R architecture, operating systems running at PL1 (IRQ, FIQ, SVC, System, etc) are able to use an MPU, as well as the hypervisor running at PL2 (Hypervisor). The MPU controlled by the hypervisor restricts access of memory regions or peripherals to an individual guest, or shared between guests. This is a similar strategy to the one we use with TrustZone, and was adopted by ARM to meet the strict requirements of real-time environments.

The existence of two distinct MMU interfaces as well as secure and non-secure cache entries is also seen as an advantage due to the performance gains achieved during the partitions-switch. From a real-time perspective, the use of these features is not always desirable, which means that in many potential embedded applications the use of MMU and cache will only be exploited by the non-secure guest OS. However, if the idea is to consolidate a soft-real-time system with a general purpose, the use of these features can be helpful in terms of context-switch time and performance. The only disadvantage that arises with the TrustZone-awareness in this components, is the need of minimal hypervisor support on their initialization, as well as their inaccessibility during runtime. In this case, one possible strategy to deal with this limitation is to implement some paravirtualization support, by statically analyzing the non-secure guest OS image file, identify the opcode of the instruction, and replace the instruction by hypercalls that request the access to those components mediated by the hypervisor.

Current device virtualization approach goes towards a pass-through model without any sharing device access support. Device isolation relies in a virtual form of IOMMU provided

by means of the TZPC. Similar to the limitation identified in the TZASC, the TZPC is also an optional and implementation-specific component on TrustZone specification. This means the number and type of devices that can be configured as non-secure vary from platform to platform and from vendor to vendor. For example, in Xilinx ZC702, the TTC0 is always secure and there is no way to configure its access directly form the non-secure guest OS. Despite the identified limitation on the TZPC, the pass-through policy without any support for shared devices is also somehow limiting. This kind of implementation makes sense in the case of the secure VM, to promote real-time characteristics, but is very limiting in a system there is a need to share devices among VMs and disregards one of our main design principles: the principle of least privilege. We plan to implement an hybrid approach in-between a pass-through and a paravirtualization strategy: the secure guest OS has direct and full control over the devices (pass-through model), but the non-secure VM requests access to devices via hypercalls, and the hypervisor mediates the access (paravirtualization). This model guarantees the timing requirements of the real-time environment, promotes the principle of less privilege by controlling the non-secure guest OS devices' access while overcoming the dependency of the TZPC for configuring devices as non-secure.

One of the main advantages of TrustZone resides on the interrupt subsystem. The direct assignment of interrupts to each world, without intervention of the hypervisor, is a plus, but, most importantly, it does not increase the interrupt latency of the secure world once the RTOS gets executed. One small disadvantage that comes with this model is that slight modifications need to be introduced in the secure guest OS, in order to use interrupt handlers as FIQs instead of IRQs. In doing so, another problem on this specific platform arises: the decrease of performance on yield-intensive workloads. However, since this problem is very specific to this platform and precise workloads, we believe it should not be generalized.

Last, but not least, another considerable advantage of the presented solution is its scalability. The recent ARM decision of introducing TrustZone technology in the new Cortex-M processors series opens up a number of opportunities for implementing cost-effective virtualization for future low-end real-time systems. We strongly believe that it will be possible to consolidate a hard real-time environment (as a secure guest OS) with a soft real-time environment (as a non-secure guest OS), at the cost of minimal engineering effort. The enhancements introduced in TrustZone specification for ARMv8-M architecture will definitively ensure better timing and performance guarantees, since the new specification implements more hardware support for world switching while guaranteeing faster transitions and greater power efficiency.

7 Related Work

The idea of using TrustZone technology to assist virtualization in embedded systems is not new, and the first works exploiting the intrinsic virtualization capabilities of TrustZone were proposed some years ago.

The work presented by Johannes Winter [30] was the first scientific public attempt to exploit the TrustZone technology to assist virtualization. The paper introduces a virtualization framework for handling non-secure world guests, and presented a prototype based on a secure version of the Linux-kernel that was able to boot only an adapted Linux kernel as non-secure world guest. Later, Cereia et al. [3] described an asymmetric virtualization layer implemented on top of the TrustZone technology in order to support the concurrent execution of both an RTOS and a GPOS on the same processor. The evaluation process was conducted only on an emulator, and presenting limited results regarding the virtualization overhead and

the hypervisor interference in the real-time characteristics. In [5] Frenzel et al. presented a minimal adapted version of Linux-kernel (as normal world OS) on top of a hypervisor running on the secure world side. SafeG [22], from TOPPERS Project, is a dual-OS open-source solution that takes advantage of ARM TrustZone extensions to concurrently execute an RTOS and a GPOS on the same hardware platform. ViMoExpress [13] is a lightweight virtualization solution, proposed by Oh et al., which exploits the TrustZone technology to accelerate the execution of two guest OSes. Both works do not conducted any evaluation neither reported any experiments. Schwarz et al. [23] proposed an alternative system virtualization approach based on TrustZone which allows the switch between a virtualized and non-virtualized execution mode through soft reboots.

8 Conclusion

Embedded real-time systems are proliferating at rapid pace in our everyday life, representing a huge part of our key infrastructures. The trend nowadays goes towards the consolidation of a wide range of functions into the same hardware platform, leading real-time requirements to coexist with non-real-time characteristics. Virtualization has been used as an enabler for platform consolidation whilst guaranteeing a robust functionality isolation, but the penalties incurred by existent software-based approaches, altogether with timing requirements imposed by real-time virtualization bring forth the need of hardware-assisted virtualization solutions. Among existing COTS technologies, ARM TrustZone is attracting particular attention, due to its exclusive applicability on those ARM processors where VE are not available, while offering the best cost-benefit trade-off. The problem is that this technology is still seen with a lot of scepticism, which rised an urgent need to comprehensively examine the hype, myths, and realities of the use of this technology for virtualization, especially because ARM continues to spread TrustZone across the different processors families. LTZVisor provides a tool to understand, evaluate and discuss the benefits and limitations of using this security-oriented technology to assist virtualization. We conducted an extensive set of experiments which demonstrated that this technology can effectively satisfy the strict requirements for virtualizing a real-time environment, while offering a low performance cost on running an unmodified guest GPOS. Evaluation over the non-secure guest OS also helped us understand the expected penalties over a soft-real-time guest OS, regarding the consolidation of a soft and hard real-time environment in future Cortex-M processors. With LTZVisor we want to share the experience gained over the last years, to encourage research for next generation of TrustZone-assisted virtualization solutions.

Current research aims at multicore extension. We have already implemented support for asymmetric multiprocessing (AMP), but we also want to explore other multicore configurations. Work in the near future will focus on an extensive and exhaustive evaluation of real-time aspects with short-term and long-term tests, as well as studying the timing interferences and sources of non-determinism which arise from the multicore approach. Extension of LTZVisor for new generation Cortex-M platforms is also at the top of our goals, but we still need to wait until the release of the first ARMv8-M boards.

Acknowledgements. We would also like to thank the anonymous reviewers for their helpful comments on the first version of this paper.

References

1 Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003. doi:10.1145/1165389.945462.

2 F. Baum and A. Raghuraman. Making Full use of Emerging ARM-based Heterogeneous Multicore SoCs. In *Proceedings of the 8th European Congress on Embedded Real Time Software and Systems*, Jan 2016.

3 M. Cereia and I. Bertolotti. Virtual Machines for Distributed Real-time Systems. *Comput. Stand. Interfaces*, 31(1):30–39, January 2009. doi:10.1016/j.csi.2007.10.010.

4 C. Dall and J. Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. *SIGPLAN Not.*, 49(4):333–348, February 2014. doi:10.1145/2644865.2541946.

5 T. Frenzel, A. Lackorzynski, A. Warg H., and Härtig. ARM TrustZone as a Virtualization Technique in Embedded Systems. *Twelfth Real-Time Linux Workshop*, 2010.

6 G. Heiser. Virtualizing Embedded Systems: Why Bother? In *Proceedings of the 48th Design Automation Conference*, DAC'11, pages 901–905. ACM, 2011.

7 H. Joe, H. Jeong, Y. Yoon, H. Kim, S. Han, and H. W. Jin. Full virtualizing micro hypervisor for spacecraft flight computer. In *2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*, pages 6C5–1–6C5–9, Oct 2012. doi:10.1109/DASC.2012.6382393.

8 Genode Labs. An Exploration of ARM TrustZone Technology. URL: https://genode.org/documentation/articles/trustzone.

9 C. Lee, S. W. Kim, and C. Yoo. VADI: GPU Virtualization for an Automotive Platform. *IEEE Transactions on Industrial Informatics*, 12(1):277–290, Feb 2016. doi:10.1109/TII.2015.2509441.

10 Miguel Masmano, Ismael Ripoll, Alfons Crespo, and J. Metge. Xtratum: a hypervisor for safety critical embedded systems. In *11th Real-Time Linux Workshop*, pages 263–272. Citeseer, 2009.

11 L. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996.

12 B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin. TrustZone Explained: Architectural Features and Use Cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pages 445–451, Nov 2016. doi:10.1109/CIC.2016.065.

13 S. Oh, K. Koh, C. Kim, K. Kim, and S. Kim. Acceleration of dual OS virtualization in embedded systems. In *2012 7th International Conference on Computing and Convergence Technology (ICCCT)*, pages 1098–1101, Dec 2012.

14 S. Patni, J. George, P. Lahoti, and J. Abraham. A zero-copy fast channel for inter-guest and guest-host communication using VirtIO-serial. In *2015 1st International Conference on Next Generation Computing Technologies (NGCT)*, pages 6–9, Sept 2015. doi:10.1109/NGCT.2015.7375072.

15 S. Pinto, T. Gomes, J. Pereira, J. Cabral, and A. Tavares. IIoTEED: an enhanced Trusted Execution Environment for Industrial IoT Edge Devices. *IEEE Internet Computing*, 21(1):40–47, Jan-Feb 2017. doi:10.1109/MIC.2017.17.

16 S. Pinto, D. Oliveira, J. Pereira, J. Cabral, and A. Tavares. FreeTEE: When real-time and security meet. In *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, pages 1–4, Sept 2015. doi:10.1109/ETFA.2015.7301571.

17 S. Pinto, D. Oliveira, J. Pereira, N. Cardoso, M. Ekpanyapong, J. Cabral, and A. Tavares. Towards a lightweight embedded virtualization architecture exploiting ARM TrustZone. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pages 1–4, Sept 2014. doi:10.1109/ETFA.2014.7005255.

18 S. Pinto, J. Pereira, T. Gomes, M. Ekpanyapong, and A. Tavares. Towards a TrustZone-assisted Hypervisor for Real Time Embedded Systems. *IEEE Computer Architecture Letters*, PP(99):1–1, 2016. doi:10.1109/LCA.2016.2617308.

19 S. Pinto, A. Tavares, and S. Montenegro. Space and time partitioning with hardware support for space applications. *Data Systems In Aerospace (DASIA), European Space Agency, (Special Publication) ESA SP*, 2016.

20 D. Reinhardt and G. Morgan. An embedded hypervisor for safety-relevant automotive E/E-systems. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, pages 189–198, June 2014. doi:10.1109/SIES.2014.6871203.

21 Rusty Russell. Virtio: Towards a De-facto Standard for Virtual I/O Devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008. doi:10.1145/1400097.1400108.

22 D. Sangorrin, S. Honda, and H. Takada. Dual operating system architecture for real-time embedded systems. In *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, Brussels, Belgium*, pages 6–15, 2010.

23 O. Schwarz, C. Gehrmann, and V. Do. Affordable Separation on Embedded Platforms. In *Proceedings of the 7th International Conference on Trust and Trustworthy Computing*, volume 8564 of *LNCS*, pages 37–54. Springer-Verlag New York, Inc., 2014. doi:10.1007/978-3-319-08593-7_3.

24 Udo Steinberg and Bernhard Kauer. NOVA: A Microhypervisor-based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys'10, pages 209–222. ACM, 2010. doi:10.1145/1755913.1755935.

25 H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang. TrustICE: Hardware-Assisted Isolated Computing Environments on Mobile Devices. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 367–378, June 2015. doi:10.1109/DSN.2015.11.

26 A. Tavares, A. Dídimo, T. Lobo, P. Cardoso, J. Cabral, and S. Montenegro. Rodosvisor – An ARINC 653 quasi-compliant hypervisor: CPU, memory and I/O virtualization. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies Factory Automation (ETFA 2012)*, pages 1–10, Sept 2012. doi:10.1109/ETFA.2012.6489588.

27 J. Taylor. Security for the next generation of safe real-time systems. In *Proceedings of Embedded World Conference, Nuremberg, Germany*, March 2016.

28 Prashant Varanasi and Gernot Heiser. Hardware-supported Virtualization on ARM. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys'11, pages 11:1–11:5. ACM, 2011. doi:10.1145/2103799.2103813.

29 P. Wilson, A. Frey, T. Mihm, D. Kershaw, and T. Alves. Implementing Embedded Security on Dual-Virtual-CPU Systems. *IEEE Design Test of Computers*, 24(6):582–591, Nov 2007. doi:10.1109/MDT.2007.196.

30 J. Winter. Trusted Computing Building Blocks for Embedded Linux-based ARM Trustzone Platforms. In *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*, STC'08, pages 21–30. ACM, 2008. doi:10.1145/1456455.1456460.

31 S. Zampiva, C. Moratelli, and F. Hessel. A hypervisor approach with real-time support to the MIPS M5150 processor. In *Sixteenth International Symposium on Quality Electronic Design*, pages 495–501, March 2015. doi:10.1109/ISQED.2015.7085475.

VCDC: The Virtualized Complicated Device Controller

Zhe Jiang[1] **and Neil Audsley**[2]

1 **Department of Computer Science, University of York, York, UK**
zj577@york.ac.uk
2 **Department of Computer Science, University of York, York, UK**
neil.audsley@york.ac.uk

Abstract

I/O virtualization enables time and space multiplexing of I/O devices, by mapping multiple logical I/O devices upon a smaller number of physical devices. However, due to the existence of additional virtualization layers, requesting an I/O from a guest virtual machine requires complicated sequences of operations. This leads to I/O performance losses, and makes precise timing of I/O operations unpredictable.

This paper proposes a hardware I/O virtualization system, termed the Virtualized Complicated Device Controller (*VCDC*). This I/O system allows user applications to access and operate I/O devices directly from guest VMs, and bypasses the guest OS, the Virtual Machine Monitor (VMM) and low layer I/O drivers. We show that the VCDC efficiently reduces the software overhead and enhances the I/O performance and timing predictability. Furthermore, VCDC also exhibits good scalability that can handle I/O requests from variable number of CPUs in a system.

1998 ACM Subject Classification C.3 Real-time and Embedded Systems

Keywords and phrases many-core system, I/O virtualization, real-time I/O, hardware manager

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.5

1 Introduction

In the last decade, virtualization technology has been widely used not only in server and desktop platforms, but also in embedded systems [23]. Using virtualization brings superior benefits for the whole system, including increased resource utilization, reduced volume and cost of hardware, and a better load balance in cores [23, 1, 19].

In real-time systems, the primary benefits offered by virtualization are isolation and security. Specifically, guest virtual machines (VMs) are logical isolated, which means the applications executed in one guest VM can never affect the other virtual machines, even if it breaks down. The feature of isolation also brings significant support for the timing analysis of the tasks in a virtual machine [8].

In real-time systems, the I/O performance is often a bottleneck of an I/O-bounded system [3], which mainly results from the very slow processing speed of normal I/O devices compared to CPUs. This results in a performance reduction for the whole system.

When it comes to multi-core and many-core systems, these issues are magnified, because of CPU scheduling and contention over I/O resources. For example, in a traditional bus-based multi-CPU system, if an I/O operation is requested by a user application, the system should deal with the scheduling of cores inside one CPU as well as the I/O resource scheduling among all the CPUs.

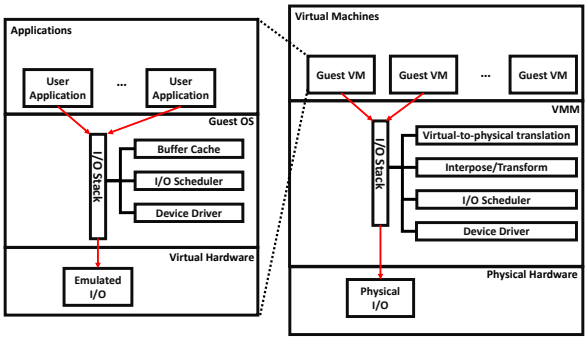


Figure 1 Flow of I/O Request in Traditional Virtualization System.

These issues are magnified with virtualization technology. When an application invokes an I/O request from a guest *Virtual Machine (VM)*, this I/O request will be transmitted via low layer drivers to the guest OS, *Virtual Machine Monitor (VMM)* and Host OS, which results in a serious loss of the system and I/O performance, see Figure 1.

In real-time systems, it is often vital for applications to access I/O devices at specific times in order to achieve the accurate control over I/O required. For example, the control of an automotive engine often requires I/O at accurate times in order to inject fuel at the optimal time [18]. Also, in a 3D printer, precise control of I/O is required [33]. This I/O operation must be occur at an exact time, i.e. be *timing-accurate* – it can be neither late nor early (within a small error bound). In a single-core system, latencies caused by device drivers and application process scheduling make timing-accurate I/O control problematic. In many-core systems, these issues are magnified: the transmission latencies from a processor to an I/O controller can be substantial and variable due to the communication bottlenecks and contention.

These issues are magnified even further with virtualization technology. Virtualizing one physical I/O to multiple virtual I/Os, complex I/O resource management (e.g. scheduling and prioritization) and the complicated path of an I/O request worsen the transmission latencies from a processor to an I/O controller. Hence, it is difficult for an application from a guest VM to issue an I/O operation that will result in a timing-accurate device level I/O operation.

Virtualization relies on hardware support, therefore today's chip manufacturers have promoted different technologies for I/O virtualization in order to mitigate these issues. Intel's Virtualization Technology for Directed I/O (*VT-D*) [12], which can provide a direct I/O access from guest VMs, is one example of this. The IOMMU [2] is applied to commercial PC-based systems to offload memory protection and address translation, in order to provide a fast I/O access from guest VMs. However, even with hardware assistance, the I/O performance from the guest VMs cannot reach the original I/O performance in a system without virtualization, let alone improve on it. Achieving timing accuracy of I/O operations in a virtualized system, even with hardware support is difficult [33]. Additionally, these commonly used hardware assists on I/O virtualization cannot help the predictability and timing-accuracy of the I/O operations requested from guest VMs.

To overcome these issues, we designed a hardware I/O system for multi-core and many-core systems. The contribution of this paper is the designed virtualized complicated device controller, termed the *VCDC*, that integrates the VMM and I/O drivers into the hardware layer, thus achieving significant improvements of I/O performance in guest VMs. The VMM

in VCDC virtualizes a physical I/O device to multiple virtual I/O devices for guest VMs. For example, in a 16-core system, the VMM can separate a single monitor into 16 individual partitions and provide access interfaces for each guest VM. In addition, the I/O drivers in VCDC provide high layer control interfaces for the guest VMs. With VCDC, the user applications in a guest VM are able to operate an I/O via very simple requests. Furthermore, if a user application is going to request the VGA controller to display a character from a guest VM, such as *'A'*, at coordinate *(2, 1)*, the user application is only required to transfer the ASCII of the character followed by its coordinates to the VGA part inside VCDC, that is *'0x41', '0x02', '0x01'*.

The VCDC utilises a timing-accurate I/O controller [32] to provide clock cycle level accurate I/O operations.

The paper is organized as follows: Section 2 presents our motivation. Section 3 presents the design and implementation of the VCDC, respectively. Section 4 evaluates the performance of a many-core system with VCDC. Section 5 presents some related work, with conclusions offered in Section 6.

2 Motivation

The most significant challenge in I/O virtualization is the loss of I/O performance. In conventional I/O virtualization, the potential overhead is associated with the indirection and interposition of an I/O request, as well as the complex resource management (e.g. scheduling and prioritization) [25].

2.1 Complicated Path of I/O requests

Figure 1 shows the flow of I/O requests handled in a traditional virtualization system. When an application running within a VM issues an I/O request, typically by making a system call, it is initially processed by the I/O stack in the guest OS, which is also running within the VM. A device driver in the guest OS issues the request to a virtual I/O device, which the VMM then intercepts. The VMM schedules requests from multiple VMs onto an underlying physical I/O device, usually via another device driver managed by the VMM or a privileged guest VM with direct access to the physical hardware.

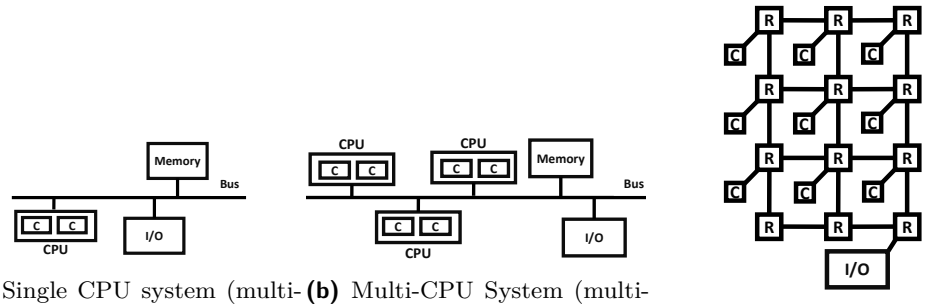
This complicated path of I/O requests poses three main drawbacks for the whole system [25]:

- *Significant software overhead.* Most of these operations are processed in software, which causes significant CPU overheads.
- *Longer response time of I/O operations.* Compared with the original system, virtualization requires more time to handle the same I/O request from the guest OS. This also causes a decline in I/O throughput.
- *Worse timing accuracy of I/O Operations..* It is very difficult for an I/O operation (e.g. read) to occur at a particular time point [32].

2.2 Complicated I/O Resource Management

In multi-core and many-core systems, in addition to the complicated path of I/O requests, complex I/O resource management is another bottleneck in virtualization:

- *Single CPU (Multi-core) System (Figure 2a).* In a bus-based single CPU system, user applications can normally request and operate I/O devices by modifying memory-mapped registers. The overhead of I/O resource management mainly comes from the scheduling of



(a) Single CPU system (multi-core) **(b)** Multi-CPU System (multi-core) **(c)** Many-core System

Figure 2 Structure of Multi-CPU and Many-Core Systems. C – Core; R – Router / Arbiter.

the CPU – deciding which core has the priority to access the I/O device. This procedure is normally handled by the OS.

- *Multi-CPU System (Multi-core) (Figure 2b).* In bus-based multi-CPU systems, apart from the CPU scheduling, the contention over I/O devices is unavoidable when a shared I/O is to be accessed. To solve the issue of I/O contention among CPUs, hardware mutexes are normally added in multi-CPU systems, which causes extra hardware overhead as well as high bus workload (frequent communication is required between CPUs and hardware mutex).
- *Many-core System (Figure 2c).* In many-core systems, all arbitrations among cores are turned over to the system arbiter (e.g. the routers in a NoC-based system), therefore CPU scheduling is not required. However, many-core systems still suffer from I/O contention when different cores need to access I/O devices at the same time.

In general, complicated I/O resource management poses the main three drawbacks for the whole system:

1. *Significant system overhead.* CPU scheduling is mostly implemented at the software level, and I/O contention is mostly handled at the hardware level, which both consume significant system overhead.
2. *Unpredictable I/O operations.* The complexity of I/O management makes I/O operations difficult to predict.
3. *Bad scalability.* With the number of cores and CPUs increasing in a system, the workload of resource management will be also increased, which causes more serious performance reduction of the whole system.

3 Virtualized Complicated Device Controller (VCDC)

Having presented the I/O problems suffered by virtualization technology in many-core and multi-core real-time systems, in this section we proceed by introducing our proposed Virtualized Complicated Device Controller (VCDC), which enables:

- *Better I/O performance.* Includes the lower response time of I/O operations and higher I/O throughput.
- *Predictability.* I/O operations requested from a guest OS are more predictable, than under conventional virtualization.
- *Lower software overhead.* Moves the VMM and low level I/O drivers from kernel mode (at the software level) to the VCDC.

- *Abstracted high layer access.* The user application in a guest virtual machine is able to request and operate an I/O device via invoking simple high layer drivers. For example, a user application can request to read a series of data from a SPI-Flash by sending a request with parameters to the VCDC: "*Read SPI-Flash* (instruction), from the *start address* to the *end address* (parameters)".
- *Scalability.* We propose a distributed implementation. When the VCDC is employed, to add one more CPU into a system, the users are only required to add one group of dedicated CPU FIFO, which aims to provide an interface between the added CPU and the VCDC.
- *Global arbitration.* We propose a modularized implementation, whereby the scheduling policy of the arbiter can be switched easily between round robin, fixed priority and customized scheduling policies [14].
- *Cycle level timing-accuracy.* All I/O operations over the GPIO pins can be issued with an accuracy of a single cycle via being integrated with our clock cycle level timing-accurate I/O controller [32].

3.1 Virtualization in the VCDC Systems

VCDC provides I/O virtualization for guest VMs, such that a physical I/O device can be virtualized to multiple virtual I/Os for each virtual machine. In a system with VCDC, the I/O virtualization has the following features:

- *Bare-metal virtualization [23].* Host OS is not required. A guest OS can be executed on a processor, directly.
- *Para-virtualization [17].* The I/O management module of a guest OS should be replaced by our high layer I/O drivers, which can significantly reduce the software overhead.

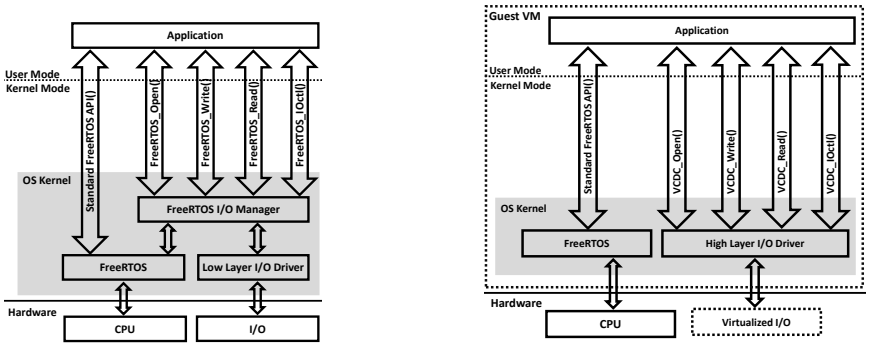
For the purposes of the discussion, in this paper, we define the following terms in this way:

- *I/O request..* Sent directly from a user application. It could be a high level abstracted command, which cannot be used directly on an I/O controller.
- *I/O instructions.* Can be used to control an I/O device controller directly.

The VCDC transforms each high level I/O request to single or multiple I/O instruction(s), that can be used on the physical I/O directly. For example, in our prototype implementation, a physical monitor (VGA controlled) is virtualized into four sections. The screen of the monitor is separated into four sections by VCDC, which is used to display the content sent from each guest VM. In each VM, the initial coordinate of the (virtual) screen is (0, 0), which is respectively mapped to the following physical coordinates of the screen: (0, 0), (0, 100), (0, 200) and (0, 300). When a user application in the guest VM #3 sends an I/O request "*Display 'Hello World' at coordinate (0, 0)*", the VCDC will transform this request to "*Display 'Hello World' at coordinate (0, 300)*" and send corresponding instructions to the VGA controller.

3.2 Guest Virtual Machine and Guest OS

In our approach, each processor has an individual guest VM. As bare-metal virtualization is deployed (no host OS required), in each guest VM, a guest OS is able to execute in kernel mode to achieve full functionality. Given that the VCDC provides part of the device driver, we also employ para virtualization (modified OS kernel) to reduce software size, which we build using some high layer I/O drivers to replace the original I/O manager. Currently, we



(a) FreeRTOS Kernel in a non-VCDC system) **(b)** FreeRTOS Kernel in a VCDC system

Figure 3 FreeRTOS Kernels in Non-VCDC and VDCD systems.

have provided three modified OS to support the I/O virtualization [14], which are FreeRTOS [7], ucosII [16] and Xilkernel [31]. In Figure 3, we use FreeRTOS as an example to illustrate the modification of a guest OS kernel in VCDC systems.

Compared with the original FreeROTS kernel (Figure 3a), the user application in a guest VM in VCDC system (Figure 3b) is able to access and operate I/O via the high layer I/O drivers, which are independent of the core module of the FreeRTOS.

Additionally, user applications running on the original FreeRTOS kernel can be ported to the modified kernel directly in a VCDC system (without any modification), since we have not modified the OS interfaces.

3.3 System Model

Typical use of the VCDC within a NoC architecture is shown in Figure 4 – all the I/O functions are performed by the VCDC rather than remotely by software.

At run-time an application in a guest VM can invoke a high layer I/O driver on the VCDC to achieve the required I/O. The communications packets are transferred between the CPU and the VCDC via routers in the NoC. As an example, the path of such an I/O request message is shown in Figure 4 as a red line.

Note that use of a NoC is not required by our system – a shared bus could be used alternatively. However, in our experiments we use a NoC.

3.4 Overall Architecture

The architecture of the VCDC consists of the following main parts (see Figure 5):

- *Hardware Manager.* Provides the interface to/from application CPUs via the NoC mesh.
- *I/O Virtual Machine Monitor (I/O VMM).* Provides the functionality of virtualization for I/O devices.
- *I/O Low Layer Drivers.* Encapsulates the corresponding drivers of the specific I/O controllers (via I/O instructions).
- *I/O Controllers.* Controls the I/O devices, and can be driven by the low layer drivers directly.
- *Memory Access Module.* Provides the memory access interfaces for I/Os.

These architectural elements are detailed in the following subsections.

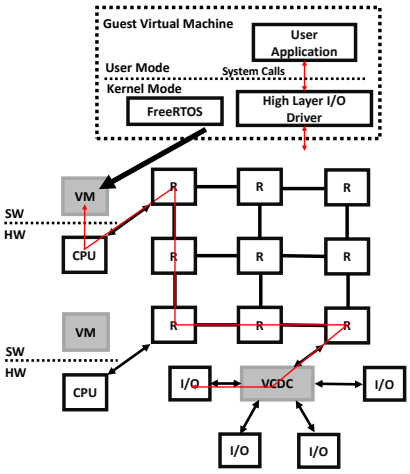


Figure 4 System Model of a NoC with VCDC. VM – Virtual Machine; R – Router / Arbiter.

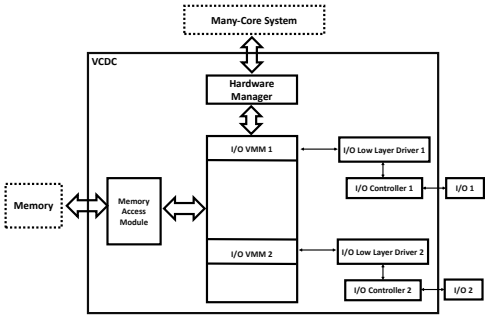
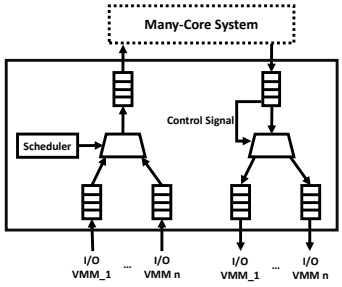
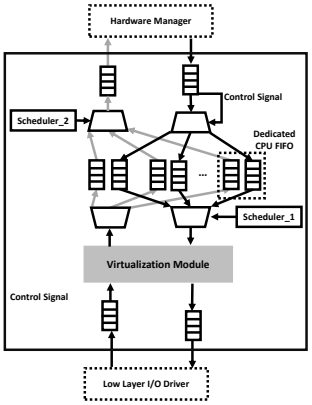


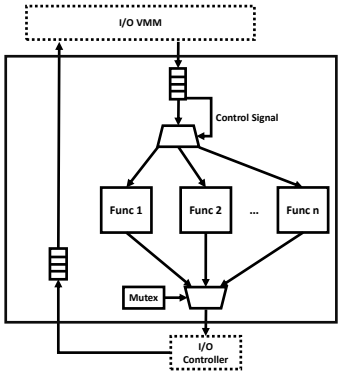
Figure 5 Architecture of VCDC.



(a) Architecture of Hardware Manager



(b) Architecture of I/O VMM



(c) Architecture of I/O Low Layer Driver

Figure 6 Architectures of inner modules of VCDC.

3.5 Detailed Architecture

3.5.1 Hardware Manager

The hardware manager is responsible for communicating with application CPUs, allocating incoming messages (I/O requests) from different CPUs to corresponding I/O VMMs, as well as allocating response messages (I/O responses) from I/O VMMs back to CPUs. The architecture of the hardware is shown in Figure 6a, with the right hand part allocating incoming requests from the NoC; and the left hand part taking ending data back to CPUs from VCDC.

The right hand part of the hardware manager is mainly comprised of one input FIFO, a multiplexer and multiple output FIFOs (dependent on the number of I/O VMMs). The output FIFOs are connected to the different I/O VMMs. Similarly, the left hand part of the hardware manager is mainly comprised of multiple input FIFOs (dependent on the number of I/O VMMs), a multiplexer, an output FIFO and a scheduler. The input FIFOs are connected to the I/O VMMs, in order to receive the data to be sent back to the CPUs. The scheduler

controls the multiplexer to choose which input FIFO can transmit data into the output FIFO (if neither input FIFO is empty the FIFOs are chosen in a round-robin manner).

Additionally, the FIFOs used to connect with I/O VMMs can be connected to I/O controllers directly, which assists in supporting different I/O devices.

3.5.2 I/O VMM

I/O VMM maintains the virtualization of I/O devices. Considering that the functionalities and features of I/O devices are different, it is very difficult to build a general-purpose module to achieve virtualization for all kinds of I/O devices. Therefore, we create some specific-purpose I/O VMM for those commonly used I/O devices, including UART, VGA, DMA, Ethernet, etc. Users can also easily add their customized I/O VMM into VCDC via our provided interfaces [14]. All of these I/O VMMs have a general architecture, see Figure 6b.

The general architecture of the I/O VMMs are the same, except for the virtualization module. The I/O VMM is comprised of two groups of communication FIFOs, four multiplexers, two schedulers, groups of dedicated CPU FIFOs and a virtualization module.

The two groups of communication FIFOs are connected with the hardware manager and a low layer I/O driver respectively, providing the communication interfaces between the hardware manager and the low layer I/O drivers. The dedicated CPU FIFOs are built to store the I/O requests sent from different CPUs and I/O response messages sent back from the I/O (as buffers); one CPU owns an individual group of dedicated CPU FIFOs. The number of groups of dedicated FIFOs are generic, so that users can add any number of dedicated CPU FIFOs into the VCDC [14], which provides for scalability. The two schedulers take charge of the scheduling of I/O requests and I/O response. Specifically, *Scheduler_1* determines which I/O request can be served by the virtualization module first, and *Scheduler_2* determines which I/O response can be sent back to the hardware manager first.

The virtualization module transforms I/O requests (sent to an virtual I/O) to I/O instructions (can be used to control a physical I/O). The implementation of this virtualization module depends on the specific I/O devices to be controlled. Currently, we have provided the virtualization module for some commonly used I/O, including UART, VGA, DMA, Ethernet and an SPI NOR-flash. Due to limitations of space here, in Section 4.3.1 we will only introduce the virtualization module for the Ethernet as an example.

3.5.3 Low Layer I/O Driver

Low layer I/O drivers takes charge of encapsulating the specific I/O drivers for an specific I/O controller (shown in Figure 6c). We encapsulate the functions of I/O drivers into separate hardware modules, e.g. read the data from a specific address of the SPI NOR-flash.

As shown, a low layer I/O driver is comprised of two FIFOs (one input and one output), two multiplexers, one mutex and multiple functions of I/O drivers. Specifically, the input FIFO is responsible for receiving I/O instructions from I/O VMM, and the output FIFO takes charge of receiving I/O responses from the I/O controller. In order to guarantee that the low layer I/O driver is able to execute the I/O instructions in the same sequence as they are sent by the I/O VMM, a mutex is added. While instructions are being carried out by one of the hardware functions, other I/O instructions must be blocked to wait to access the I/O controller.

3.5.4 Memory Access Module

VCDC also provides an interface to access the external memory (DDR), which is named BlueTree [11]. I/O devices are able to use this interface to read and write the external memory, such as the DMA. We will not introduce the implementation of the memory access module in this paper; for more details please see [11], [9] and [10].

3.5.5 Timing-accurate I/O Controller

Clock cycle level timing-accurate I/O operations can be achieved by connecting the GPIO Command Processor (GPIOCP) [32].

The GPIOCP is a resource efficient programmable I/O controller, which permits applications to instigate complex sequences of I/O operations at an exact time, so achieving timing-accuracy of a single clock cycle. This is achieved by loading application specific programs into the GPIOCP which generates a sequence of control signals over a set of General Purpose I/O (GPIO) pins, e.g. read / write. Applications then are able to invoke a specific program at run-time by sending the GPIO command, for example Run command X at time t (at a future time). This achieves cycle level timing-accuracy as the latencies of the I/O virtualization and communication bus are removed. As an example, a periodic read of a sensor value by an application can be achieved by loading the GPIOCP with an appropriate program, then at run-time the GPIOCP issues a command such as run command X at time t and repeat with period Z – the values are read at exact times, with the latency of moving the data back to the application considered within that application's execution time.

In [32], we have shown that deployment of GPIOCP can guarantee the clock cycle level granularity of I/O operations. In this paper, GPIOCP is integrated with the VCDC as a controller, and provides I/O virtualization as well as cycle level timing-accurate I/O operations.

4 Evaluation

The VCDC was implemented using Bluespec [13] and synthesised for the Xilinx VC709 development board [28] (further implementation details are given in technical report [14]). The VCDC is connected to a 4 x 5 size 2D mesh type open source NoC[21] containing 16 Microblaze CPUs [27] running the modified guest OS FreeRTOS (v9.0.0) in the guest VM. The modification of the FreeRTOS is described in Section 3.2. The architecture is shown in Figure 7.

To enable comparison, a similar hardware architecture was built, but without the VCDC and I/O virtualization – note that this architecture requires I/O operations requested by Mircoblaze to pass through the mesh to the I/O rather than being controlled by a VCDC. The OS running on each Microblaze is FreeRTOS (v9.0.0) with its official I/O management module [6]. Both architectures run at 100 MHz.

4.1 Response Time of I/O Operations

This experiment aims to evaluate the performance of the I/O system while CPU and I/O are fully loaded in a VCDC and non-VCDC system. In both architectures, 9 CPUs are active, whose coordinates are from (0, 0) to (0, 2), (1, 0) to (1, 2) and (2, 0) to (2, 2). In both architectures, all the active CPUs have an independent application that is set to be running, which continuously reads data from a SPI NOR-flash (model: S25FL128S). Specifically, the experiments are divided into four groups, depending on the read bytes in each I/O request:

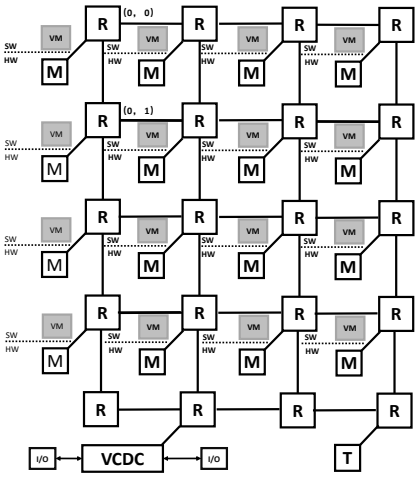


Figure 7 Experimental Platform. R – Router / Arbiter; M – Microblaze; VM – Guest Virtual Machine; T – Timer.

1, 4, 64 and 256. All the experiments are implemented 1000 times and recorded in tables. A lower I/O response time indicates a higher performance of the corresponding I/O system. We name the experiments according to the global scheduling policy and bytes of read data in one I/O request. For example, *non-VCDC-RR-4B* stands for a non-VCDC system with round-robin global scheduling policy; and 4 bytes of data read from the NOR-flash in one I/O request.

In the non-VCDC architecture, we modify the I/O management of FreeRTOS to be suitable for many-core systems[1]. While the user applications on different CPUs are requesting the I/O at the same time point, the scheduling policy can be set as FIFO (non-VCDC-FF) and Round-Robin (non-VCDC-RR) respectively.

Results of 1000 experiments are given in Table 1, showing that the response time of I/O requests in the non-VCDC architecture is significantly higher for the reading of 1 byte, 4 bytes, 64 bytes or 256 bytes from the NOR-flash, especially while Round-Robin scheduling policy being employed. For example, the average response time of non-VDCD-RR-1B is higher than $360,000$ ns ($36,000$ clock cycles). In contrast, in VDCD-1B, the worst I/O response time is lower than $4,000$ ns (400 clock cycles). The high I/O response time in non-VCDC-RR is mainly caused by the software implementation of round-robin I/O scheduling policy (complicated on-chip communication is required). In experiments with more bytes being read, the VCDC system maintains its superior performance. For example, in VCDC-256B, the I/O response time is lower than $900,000$ ns ($90,000$ clock cycles), which is similar to the worst case of the I/O response time in non-VCDC-RR-1B – $658,850$ ns ($65,885$ clock cycles).

Additionally, when it comes to the variance of I/O response time in 1000 experiments, the VCDC systems have a better performance than the non-VCDC systems. For example, in the non-VCDC-FF-1B, the highest variance of I/O response time is greater than $15,000$ ns ($1,500$ clock cycles). When it comes to the non-VCDC-RR-1B, the situation becomes worse: the highest variance of I/O response time reaches $600,000$ ns ($60,000$ clock cycles). Conversely, in the VCDC-1B, the highest variance of I/O response time is less than 500 ns (50 clock

[1] The I/O management in FreeRTOS is designed for a single-core system; in our experiments, we modify it to be suitable for many-core systems.

Table 1 I/O response time in VCDC and non-VCDC systems (unit: clock cycle).

	Non-VCDC System Scheduling Policy: FIFO			Non-VCDC System Scheduling Policy: RoundRobin			VCDC System		
CPU Index	Min	Max	Mean	Min	Max	Mean	Min	Max	Mean
Read 1 Byte									
(0, 0)	9357	9357	9357	6149	65885	36060	285	285	285
(0, 1)	7425	8989	8915	7073	65849	35860	380	403	396
(0, 2)	7057	8598	8415	7096	65849	36049	380	403	395
(1, 0)	7057	8207	8203	7096	65826	36237	357	403	391
(1, 1)	9748	9748	9748	7073	65826	36410	403	403	403
(1, 2)	7425	8966	7476	7073	65826	36576	334	334	334
(2, 0)	7034	8598	7467	7073	65826	36741	357	403	366
(2, 1)	7057	8207	7576	7096	65826	36930	357	403	377
(2, 2)	6121	6121	6121	7073	65803	37102	334	334	334
Read 4 Bytes									
(0, 0)	58002	58477	58021	29515	316248	173091	1066	1123	1093
(0, 1)	29611	36281	34908	33243	309490	168542	1247	1408	1356
(0, 2)	29657	37017	36191	34770	322660	176642	1293	1569	1398
(1, 0)	28875	36258	35264	34770	322547	177561	1362	1569	1412
(1, 1)	58361	58844	58381	33243	309382	171130	1316	1385	1325
(1, 2)	29588	35499	30208	34657	322547	179222	1247	1408	1270
(2, 0)	29979	37040	31290	35223	327813	182972	1247	1569	1322
(2, 1)	28139	36235	34785	32641	302799	169881	1293	1431	1369
(2, 2)	57579	58062	57599	32535	302693	170670	1247	1270	1249
Read 64 Bytes									
(0, 0)	907744	929955	918905	408908	4381352	2398035	18770	19245	18935
(0, 1)	450935	478696	460279	393536	4216640	2307883	19007	20272	19521
(0, 2)	479501	579758	538170	476993	4426369	2423243	19053	22549	20808
(1, 0)	473268	571294	520525	476993	4424823	2435851	19145	23032	21203
(1, 1)	909739	936166	921822	488037	4541994	2512343	19076	19398	19188
(1, 2)	449348	473636	456782	475305	4423507	2446804	19007	20157	19418
(2, 0)	474027	579068	535487	475305	4423507	2469029	19007	22043	20535
(2, 1)	472095	565429	518137	489451	4555159	2542512	19007	22549	20895
(2, 2)	900332	920618	907492	468232	4356158	2456170	19007	19237	19073
Read 256 Bytes									
(0 ,0)	3628902	3702565	3674076	1586442	16998330	9303655	75609	78231	76046
(0, 1)	1810819	1897023	1826232	1848174	17206343	9370227	75839	79841	77648
(0, 2)	1897828	2181970	2119170	1830492	17041721	9280577	75885	88305	83101
(1, 0)	1890399	2132060	2046512	1862700	17279325	9512215	75997	89708	84212
(1, 1)	3631085	3708365	3679649	1848508	17147673	9620444	75908	78484	76336
(1, 2)	1808220	1897000	1823103	1842516	17147673	9528055	75839	79542	77494
(2, 0)	1897391	2180659	2116159	1828370	17016021	9497681	75839	87040	82616
(2, 1)	1890422	2131301	2044241	1869796	17345151	9731236	75839	89202	83631
(2, 2)	3616296	3682191	3641053	1826248	16990334	9579806	75839	78346	76212

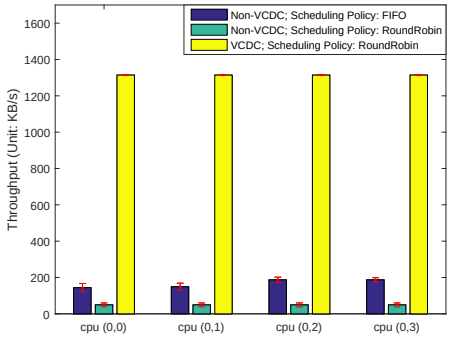


Figure 8 I/O Throughput.

cycles). For experiments with more bytes being read, VCDC systems still have a better performance. For example, in non-VCDC-RR-256B, the maximum variance of the I/O response time reaches 154, 118, 880 ns (15, 411, 888 clock cycles). Conversely, in VCDC-256B, the maximum variance of the I/O response time is only 137, 310 ns (13, 731 clock cycles), which is 1/1000 of variance in the non-VCDC-RR-256B.

Therefore, the evaluation results shows that a system with VCDC can provide more predictable I/O operations with lower response time.

4.2 I/O Throughput

We evaluated the I/O throughput in two architectures (with VCDC and without VCDC). In the experiment, we use the same NOR-flash illustrated in Section 4.1 connected to the VCDC as our evaluation object.

In both architectures, one independent application is set to be running on each of four Microblaze CPUs (coordinates are from (0,0) to (0,3)) and continuously writing to the NOR-flash – one byte can be written during one I/O request. We record the written bytes from each CPU within 1 second as the I/O throughput. The result of higher I/O throughput implies a better performance of the I/O system. All the evaluations are implemented 1000 times. The evaluation results are shown in Figure 8.

In the figure, four groups of bar charts present the average I/O throughput in the VCDC system and the non-VCDC system; and the error bar on each bar chart presents the variance of the I/O throughput in these 1000 experiments. As shown, on all CPUs considered, the VCDC system always provides a better performance on I/O throughput. Specifically, the I/O throughput from any of the CPUs in the VCDC system is nearly 7 times higher than the non-VCDC system with FIFO scheduling policy, and 20 times higher than the non-VCDC system with round robin scheduling policy. Additionally, when it comes to the variance of I/O throughput, the VCDC system has a better performance than the non-VCDC systems.

In general, the evaluation results in this section show that a system with VCDC can provide higher I/O throughput with smaller variance.

4.3 Scalability

In this section, we evaluate the scalability of the VCDC by measuring the I/O response time of Ethernet packets sent from different CPUs in single-core, 4-core, 8-core and 16-core systems, respectively.

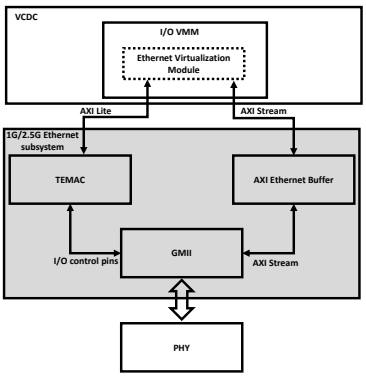


Figure 9 Connection between VCDC and Ethernet System.

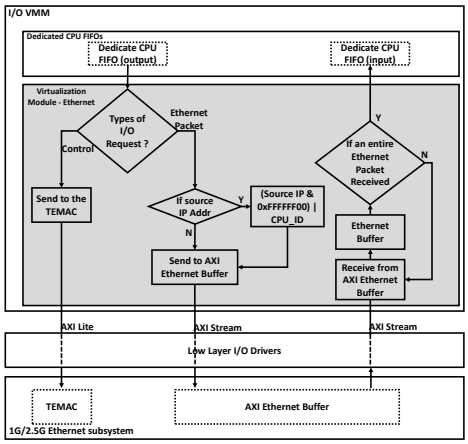


Figure 10 Virtualization Module of Ethernet I/O VMM.

4.3.1 Ethernet Virtualization

A full Ethernet packet comprises an Ethernet header, an IP header, a TCP header and the payload [22]. The virtualization of Ethernet is implemented by virtualizing the IP address of Ethernet packets sent from each processor.

In a many-core or multi-core system, all the Ethernet packets sent from different CPUs should have the same IP address. In a system with VCDC, the virtulization module sets the last 8 bits of the source IP address as the CPU ID, so that the Ethernet packets sent from each CPU can have a unique source IP address. With VCDC employed, one CPU is able to communicate with a dedicated destination without interference from other CPUs.

In our approach, VCDC is designed to connected with Xilinx 1G/2.5G Ethernet subsystem [29], which comprises three IP cores: a Tri-mode Ethernet MAC (*TEMAC*) [24], a Gigabit MII (*GMII*) [24] and an Axi Ethernet buffer [30], see Figure 9.

In Xilinx 1G/2.5G Ethernet subsystem, the GMII provides an interface between MAC and PHY, which is controlled by the TEMAC and the AXI Ethernet buffer. Specifically, the TEMAC takes charge of the control parts of the GMII, such as initialization and settings of communication speed. The AXI Ethernet buffer takes charge of transmission of Ethernet packets. When an Ethernet packet is received by the AXI Ethernet buffer, the packet will be sent to the GMII directly via an AXI stream interface, then sent to the physical layer.

As described in Section 3.5.2, inside I/O VMM, the virtualization module is responsible for the virtualization of a specific I/O. Figure 10 describes the inner architecture of the virtualization module inside the I/O VMM for Ethernet.

The virtualization module inside the Ethernet I/O VMM has two parts: down and up. The down part takes charge of the analysis and allocation of incoming I/O requests from the dedicated CPU FIFOs. Specifically, the I/O requests received by the virtualization module are divided into the control operations and the Ethernet packets. If the incoming I/O request is the control operation, the virtualization module will allocate it to the TEMAC inside the Ethernet subsystem via the low layer I/O drivers (AXI lite interface). If the incoming I/O request is an Ethernet packet, the virtualization module will virtualize its IP address according to its corresponding CPU IP; and send it to the AXI Ethernet buffer via the low layer I/O drivers (AXI Stream interface). Additionally, the up part takes charge of receiving Ethernet packets from the physical layer (PHY). It buffers and sends an entire Ethernet

packet back to the corresponding dedicated CPU FIFO according to the destination IP address of this Ethernet packet.

4.3.2 Experiment

The experiment is divided into two groups, which depends on the global scheduling policy of the VCDC: round-robin (named VCDC-RR) and fixed priority (named VCDC-FP). In VCDC-RR and VCDC-FP, the experiments can be further divided into four parts, according to the number of active CPUs. In these four parts of the experiments, we activate 1, 4, 8 and 16 Microblazes respectively. We name these experiment parts according to the label of the experiment plus the number of active CPUs. For example, in a 4-core VCDC system with round-robin global scheduling policy, the experiment is labelled VCDC-RR-4.

The software application running on each active CPU is the same, and is designed to continuously send 1 KB Ethernet packets via VCDC to a dedicated component. The 1 KB Ethernet packets sent from different CPUs are exactly the same, including the MAC header, the IP header, and the payload. However, the VCDC will virtualize the source IP address of each Ethernet packet based on the rules in Section 4.3.1. Additionally, the dedicated component is designed to monitor the response time of these Ethernet packets by recording the reach time and analysing the virtual source IP address of the packets. All the experiments were implemented 1000 times.

The experiment is divided into two groups, which depends on the global scheduling policy of the VCDC: round-robin (named VCDC-RR) and fixed priority (named VCDC-FP). In VCDC-RR and VCDC-FP, the experiments can be further divided into four parts, according to the number of active CPUs. In these four parts of the experiments, we activate 1, 4, 8 and 16 Microblazes respectively. We name these experiment parts according to the label of the experiment plus the number of active CPUs. For example, in a 4-core VCDC system with round-robin global scheduling policy, the experiment is labelled VCDC-RR-4.

The software application running on each active CPU is the same, and is designed to continuously send 1 KB Ethernet packets via VCDC to a dedicated component. The 1 KB Ethernet packets sent from different CPUs are exactly the same, including the MAC header, the IP header, and the payload. However, the VCDC will virtualize the source IP address of each Ethernet packet based on the rules in Section 4.3.1. Additionally, the dedicated component is designed to monitor the response time of these Ethernet packets by recording the reach time and analysing the virtual source IP address of the packets. All the experiments were implemented 1000 times; and the experiment results are depicted in tables.

In VCDC-FP, CPU (0, 0) is always set as the highest priority, followed by CPU (1, 0), (2, 0) , (3,0) and (1, 0) etc. The experiment results are shown in Table 2. As shown, for all multi-core systems, the I/O response time from the CPU with the highest priority is always fixed around 12 us; and the I/O requests from the CPUs with the lower priorities are always blocked by the the I/O requests with higher priorities, which guarantees the execution of the I/O requests with higher priorities. For example, in VCDC-FP-8, the average response time of the I/O requests from CPU (0,0) (the highest priority) is kept to 12 us, which means it can never be blocked by others. When it comes to the I/O requests from CPU (3, 1) (the lowest priority), the I/O response time is always around 96 us, which is 8 times the highest priority I/O requests. The I/O response time of the lowest priority I/O request is extended due to blocks from other CPU, which means that the VCDC system does not introduce extra delay for the lowest priority I/O request. In a 8-core system, the theoretical optimal response time of the lowest priority I/O request should be 8 times the highest priority I/O request, and our experiment results obtain this. Similarly, in VCDC-FP-16, the average response time of

Table 2 Average Response Time of Loop Back 1KB Ethernet Packets in VCDC System (Global Scheduling Policy: Fixed Priority; Unit: us).

	Number of CPUs			
CPU Index	1	2	3	4
(0, 0)	12.09	12.07	12.09	12.08
(1, 0)	–	25.50	25.51	25.50
(2, 0)	–	36.92	36.94	36.93
(3, 0)	–	48.35	48.36	48.35
(0, 1)	–	–	59.78	59.78
(1, 1)	–	–	71.21	71.19
(2, 1)	–	–	82.62	82.62
(3, 1)	–	–	94.06	95.06
(0, 2)	–	–	–	105.46
(1, 2)	–	–	–	116.90
(2, 2)	–	–	–	128.31
(3, 2)	–	–	–	139.74
(0, 3)	–	–	–	151.17
(1, 3)	–	–	–	162.58
(2, 3)	–	–	–	174.02
(3, 3)	–	–	–	185.44

Table 3 Average Response Time of Loop Back 1KB Ethernet Packets in VCDC System (Global Scheduling Policy: Round Robin; Unit: us).

	Number of CPUs			
CPU Index	1	2	3	4
(0, 0)	12.32	46.71	90.58	180.15
(1, 0)	–	47.20	90.88	180.71
(2, 0)	–	47.68	91.22	179.99
(3, 0)	–	48.19	91.58	180.66
(0, 1)	–	–	91.93	180.04
(1, 1)	–	–	92.27	180.71
(2, 1)	–	–	92.63	180.09
(3, 1)	–	–	92.98	180.77
(0, 2)	–	–	–	180.04
(1, 2)	–	–	–	180.71
(2, 2)	–	–	–	180.09
(3, 2)	–	–	–	180.77
(0, 3)	–	–	–	180.04
(1, 3)	–	–	–	180.71
(2, 3)	–	–	–	180.09
(3, 3)	–	–	–	180.77

the I/O request from CPU (3,3) (the lowest priority) is around 190 us, which is 16 times the response time of the highest priority I/O requests. The results still meet the theoretical optimal value. These experiments indicate a good scalability of the VCDC.

In VCDC-RR, the global arbiter is set to start from operating a random I/O request in each independent experiment. The experiment results are shown in Table 3. As shown, with an increase in the number of CPUs, the I/O response time of each CPU is proportional to the number of CPUs. Specifically, compared to the response time of an I/O request in VCDC-RR-1, the average I/O response time of an I/O request in VCDC-RR-4, VCDC-RR-4 and VCDC-RR-16 is respectively around 4, 8 and 16 times the average I/O response time in a single-core system. These results are close to the theoretical optimal values, which shows a good scalability of the VCDC.

4.4 Hardware and Software Overhead

This section can be mainly divided into two parts. In the first part, we compare the software overhead of a VCDC system and non-VCDC system with a software implementation of I/O

Table 4 Software Usage (object code).

Software Module	VCDC	Non-VCDC (FIFO)	Non-VCDC (Round-Robin)
I/O Manager (KB)	0	139.2	148.5
UART Driver (KB)	60.5	122.4	122.4
VGA Driver (KB)	70.2	105.2	105.2
Non-Flash Driver (KB)	90.2	135.8	145.6
Ethernet Driver (KB)	88.7	210.2	230.2

Table 5 Hardware Usage (Without GPIOCP).

Hardware Consumption	VCDC	Microblaze	SPI Controller
Look Up Tables	4812	1860	886
Registers	1413	2133	615
Block RAMs (KB)	0	8	0

management (i.e. I/O manager in FreeRTOS), see Table 4. In the second part, we compare the hardware overhead of a VCDC and a Microblaze CPU (running as a VMM), see Table 5.

4.4.1 Software Overhead

As shown in Table 4, the VCDC system significantly reduces software overhead. Specifically, the software I/O manager is not required and the size of I/O drivers is smaller in the VCDC system.

4.4.2 Hardware Overhead

As shown in Table 5, compared with a dedicated I/O controller (SPI controller), VCDC consumes more FPGA hardware resources, including look up tables and registers. When it is compared with a full-featured Microblaze, the VCDC consumes more look up tables but less registers and BRAMs.

It is a trade-off between software overhead and hardware overhead. However, the VCDC system brings significant improvements of the I/O performance, including I/O throughput, response time, variance and scalability.

4.5 On-chip Communication Overhead

In NoC-based many-core systems, all the I/O requests are transmitted as on-chip packets. A larger requirement for on-chip packets means a higher on-chip communication overhead. In this section, we compare the on-chip communication overhead while invoking commonly used I/O requests in a VCDC and non-VCDC system by recording the number of packets on the NoC. In the NoC [21], the width of all the on-chip packets are 32 bits. The evaluation results are demonstrated in Table 6.

As it is shown, while the invoked I/O request is simple, the on-chip communication overhead is similar in all the systems, e.g. displaying one pixel via the VGA in a single-core

Table 6 On-chip Communication Overhead.

I/O Device	I/O Operation		Number of on-chip Packets (Each Packet: 32-bit)		
			Non-VCDC FIFO	Non-VCDC Round-Robin	VCDC
VGA	Display 1 Pixel	1 CPU	6	6	3
		4 CPUs	24	33	12
		10 CPUs	60	87	30
	Display 10 Pixels	1 CPU	60	60	30
		4 CPUs	240	357	120
		10 CPUs	600	897	300
SPI Flash	Read 1 Byte	1 CPU	12	12	4
		4 CPUs	48	57	16
		10 CPUs	120	237	40
	Read 10 Bytes	1 CPU	120	120	40
		4 CPUs	480	597	160
		10 CPUs	1200	1497	400

system. When the I/O operations become complicated or the number of CPUs are increased, the on-chip communication overhead in non-VCDC architecture is significant; in contrast, the VCDC architecture has a lower on-chip communication overhead, for example, reading 10 bytes data from the SPI flash in 10-core systems.

4.5.1 Bottleneck of On-chip Communication

In the proposed design, a single channel interface is used for transmitting VCDC requests. It connects the many-core system and the VCDC, which has been explained in Section 3.5.1. Frequently invoked VCDC requests might cause traffic congestion at the entrance of the VCDC, which decreases the predictability of I/O operations. This traffic congestions can further affect the communication issues on the system level.

4.5.2 Discussion

In current stage, a provided solution is adding the number of communication channels in the interface between many-core system and VCDC. The multiple communication channels can alleviate communication traffic significantly. However, changing the number of communication channels requires to rebuild whole hardware, which is not suitable for a ready-built IC.

5 Related Work

Related approaches for I/O virtualization over a many-core or many-CPU architecture can be mainly divided into software virtualization and hardware virtualization. In this section, we review one software I/O virtualization (Quest-V [26]) and two hardware I/O virtualizations (VT-d [12] and SR-IOV [20]).

5.1 Quest-V

Quest-V is a virtualized multi-kernel [26]. It uses virtualization techniques to isolate kernels on different cores of a multi-core processor. Quest-V virtualizes the single CPU as two classes of VCPUs: (1) *main VCPUs* are used to schedule and track the conventional software threads; (2) *I/O VCPUs* are used to account for scheduling, execution of I/O requests and handling of I/O interrupts. The virtualization for the underlying hardware features are supported by the I/O VCPUs with corresponding I/O drivers (virtualized). Using the same

physical CPU for both software threads and for handling I/O can compromise the I/O accuracy.

5.2 Virtualization Technology for Directed I/O (VT-d)

VT-d is the hardware support for isolating and restricting device accesses to the owner of the partition managing the device, which is developed by Intel [12]. VT-d includes three key capabilities: (1) Allows an administrator to assign I/O devices to guest VMs in any desired configuration; (2) supports address translations for device DMA data transfers; and (3) provides VM routing and isolation of device interrupts. Generally speaking, VT-d provides a hardware VMM that allows user applications running in the guest VMs to access and operate the I/O devices directly. Compared with traditional software virtualization, VT-d offloads most of the overhead of virtualization to the hardware level. In a system with VT-d, in addition to I/O drivers, extra drivers for VT-d are also required in the software layer. Therefore the I/O performance in the guest VM can only reach about 70% [25], compared to the original I/O.

5.3 Single Root I/O Virtualization (SR-IOV)

Single Root I/O Virtualization (SR-IOV) is a specification, which proposes a set of hardware enhancements for the PCIe device. SR-IOV aims to remove major VMM intervention for performance data movement to I/O devices, such as the packet classification and address translation. A SR-IOV-based device is able to create multiple "light-weight" instances of PCI function entities (also known as VFs). Each VF can be assigned to a guest for direct access, but still shares major device resources, achieving both resource sharing and high performance. Currently, many I/O devices have already supported the SR-IOV specification, such as [4], [5] and [15]. Similarly to Intel VT-d, to support a SR-IOV-based I/O more drivers are needed in the software, which reduces the performance of the I/O.

6 Conclusion

In this paper, we have presented the concept of predictable hardware I/O virtualization for NoC many-core systems (VCDC). It enables applications to access and operate I/O devices directly from guest VMs, bypassing the guest OS, the VMM and low layer I/O drivers in software layer.

Evaluation reveals that VCDC can virtualize a physical I/O to multiple virtual I/Os with significant performance improvements, including faster I/O response time, greater I/O throughput, less on-chip communication overhead and good scalability. When it comes to the system overhead, the VCDC represents a trade-off between software and hardware, decreasing the software usage but requiring a greater consumption of hardware.

References

1 Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *SIGARCH Comput. Archit. News*, 34(5):2–13, October 2006. doi:10.1145/1168919.1168860.

2 Muli Ben-Yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, Alexis Bruemmer, and Leendert Van Doorn. The price of safety: Evaluating IOMMU performance. In *The Ottawa Linux Symposium*, pages 9–20, 2007. doi:10.1.1.716.7062.

3 Alan Burns and Andrew J Wellings. *Real-time systems and programming languages: Ada 95, Real-Time Java, and Real-Time POSIX*. Pearson Education, 2001.

4 Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. High performance network virtualization with SR-IOV. *J. Parallel Distrib. Comput.*, 72(11):1471–1480, November 2012. doi:10.1016/j.jpdc.2012.01.020.

5 Yaozu Dong, Zhao Yu, and Greg Rose. SR-IOV networking in Xen: Architecture, design and implementation. In *Proceedings of the First Conference on I/O Virtualization*, WIOV'08, pages 10–10, Berkeley, CA, USA, 2008. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1855865.1855875.

6 FreeRTOS. FreeRTOS I/O official website. http://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_IO/FreeRTOS_Plus_IO.shtml. Accessed September 27, 2016.

7 FreeRTOS. FreeRTOS official website. http://www.freertos.org/. Accessed September 27, 2016.

8 Marisol García-Valls, Tommaso Cucinotta, and Chenyang Lu. Challenges in Real-time virtualization and predictable cloud computing. *J. Syst. Archit.*, 60(9):726–740, October 2014. doi:10.1016/j.sysarc.2014.07.004.

9 Jamie Garside and Neil Audsley. Prefetching across a shared memory tree within a Network-on-Chip architecture. In *2013 International Symposium on System on Chip (SoC)*, pages 1–4, Oct 2013. doi:10.1109/ISSoC.2013.6675268.

10 Manil Gomony, Jamie Garside, Benny Akesson, Neil Audsley, and Kees Goossens. A globally arbitrated memory tree for mixed-time-criticality systems. *IEEE Transactions on Computers*, pages 1–1, 2016. doi:10.1109/tc.2016.2595581.

11 Manil Dev Gomony, Jamie Garside, Benny Akesson, Neil Audsley, and Kees Goossens. A generic, scalable and globally arbitrated memory tree for shared DRAM access in Real-Time systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*. IEEE Conference Publications, 2015. doi:10.7873/date.2015.0390.

12 Radhakrishna Hiremane. Intel virtualization technology for directed I/O (Intel VT-d). *Technology@ Intel Magazine*, 4(10), 2007.

13 Bluespec Inc. Bluespec System Verilog (BSV). http://www.bluespec.com/products/. Accessed September 27, 2015.

14 Zhe Jiang. VCDC technical report. https://github.com/RTSYork/BlueIO. Accessed January 27, 2017.

15 Jithin Jose, Mingzhe Li, Xiaoyi Lu, Krishna Chaitanya Kandalla, Mark Daniel Arnold, and Dhabaleswar K. Panda. SR-IOV support for virtualization on InfiniBand clusters: Early experience. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE, may 2013. doi:10.1109/ccgrid.2013.76.

16 Silicon Labs. UCOS official website. https://www.micrium.com/rtos/kernels/. Accessed September 27, 2015.

17 John A. Landis, Terrence V. Powderly, Rajagopalan Subrahmanian, Aravindh Puthiyaparambil, and James R. Hunter Jr. Computer system para-virtualization using a hypervisor that is implemented in a partition of the host system, July 19 2011. US Patent 7,984,108.

18 Jürgen Mössinger. Software in automotive systems. *IEEE Software*, 27(2):92–94, mar 2010. doi:10.1109/ms.2010.55.

19 Gil Neiger. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(03), aug 2006. doi:10.1535/itj.1003.01.

20 PCI-SIG. SR-IOV official website. http://pcisig.com/. Accessed September 27, 2016.

21 Gary Plumbridge, Jack Whitham, and Neil Audsley. Blueshell: a platform for rapid prototyping of multiprocessor NoCs and accelerators. *ACM SIGARCH Computer Architecture News*, 41(5):107–117, jun 2014. doi:10.1145/2641361.2641379.

22 D. Plummer. Ethernet address resolution protocol: Or converting network protocol addresses to 48.bit ethernet address for transmission on ethernet hardware, nov 1982. doi:10.17487/rfc0826.

23 Jyotiprakash Sahoo, Subasish Mohapatra, and Radha Lath. Virtualization: A survey on concepts, taxonomy and associated security issues. In *2010 Second International Conference on Computer and Network Technology*. IEEE, 2010. doi:10.1109/iccnt.2010.49.

24 Pang Wei Tsai, Hou Yi Chou, Mon Yen Luo, and Chu Sing Yang. Design a flexible software development environment on NetFPGA platform. In *Applied Mechanics and Materials*, volume 411-414, pages 1665–1669. Trans Tech Publications, sep 2013. doi:10.4028/www.scientific.net/amm.411-414.1665.

25 Carl Waldspurger and Mendel Rosenblum. I/O virtualization. *Communications of the ACM*, 55(1):66–73, 2012.

26 Richard West, Ye Li, and Eric S. Missimer. Quest-v: A virtualized multikernel for safety-critical Real-Time systems. *CoRR*, abs/1310.6349, 2013. URL: http://arxiv.org/abs/1310.6349.

27 Xilinx. Microblaze user manual. http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/mb_ref_guide.pdf. Accessed August 27, 2016.

28 Xilinx. VC709 official website. https://www.xilinx.com/products/boards-and-kits/dk-v7-vc709-g.html. Accessed August 27, 2016.

29 Xilinx. Xilinx 1G/2.5G Ethernet subsystem manual. https://www.xilinx.com/support/documentation/ip_documentation/axi_ethernet/v7_0/pg138-axi-ethernet.pdf. Accessed August 27, 2016.

30 Xilinx. Xilinx AXI FIFO user manual. https://www.xilinx.com/support/documentation/ip_documentation/axi_fifo_mm_s/v4_1/pg080-axi-fifo-mm-s.pdf. Accessed August 27, 2016.

31 Xilinx. Xilinx official website. https://www.Xilinx.com. Accessed July 5, 2015.

32 Neil Audsley Zhe Jiang. GPIOCP: Timing-accurate general purpose i/o controller for many-core Real-time systems. In *Proceedings of the 2017 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2017.

33 Richard West Zhuoqun Cheng and Ying Ye. Building Real-Time embedded applications on QduinoMC: A web-connected 3d printer case study. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE*. IEEE, 2017.

VOSYSmonitor, a Low Latency Monitor Layer for Mixed-Criticality Systems on ARMv8-A*

Pierre Lucas[1], Kevin Chappuis[2], Michele Paolino[3], Nicolas Dagieu[4], and Daniel Raho[5]

1 **Virtual Open Systems, Grenoble, France**
p.lucas@virtualopensystems.com
2 **Virtual Open Systems, Grenoble, France**
k.chappuis@virtualopensystems.com
3 **Virtual Open Systems, Grenoble, France**
m.paolino@virtualopensystems.com
4 **Virtual Open Systems, Grenoble, France**
n.dagieu@virtualopensystems.com
5 **Virtual Open Systems, Grenoble, France**
s.raho@virtualopensystems.com

Abstract

With the emergence of multicore embedded System on Chip (SoC), the integration of several applications with different levels of criticality on the same platform is becoming increasingly popular. These platforms, known as mixed-criticality systems, need to meet numerous requirements such as real-time constraints, Operating System (OS) scheduling, memory and OSes isolation.

To construct mixed-criticality systems, various solutions, based on virtualization extensions, have been presented where OSes are contained in a Virtual Machine (VM) through the use of a hypervisor. However, such implementations usually lack hardware features to ensure a full isolation of other bus masters (e.g., Direct Memory Access (DMA) peripherals, Graphics Processing Unit (GPU)) between OSes. Furthermore on multicore implementation, one core is usually dedicated to one OS, causing CPU underutilization.

To address these issues, this paper presents VOSYSmonitor, a multi-core software layer, which allows the co-execution of a safety-critical Real-Time Operating System (RTOS) and a non-critical General Purpose Operating System (GPOS) on the same hardware ARMv8-A platform. VOSYSmonitor main differentiation factors with the known solutions is the possibility for a processor to switch between secure and non-secure code execution at runtime. The partitioning is ensured by the ARM TrustZone technology, thus allowing to preserve the usage of virtualization features for the GPOS.

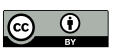
VOSYSmonitor architecture will be detailed in this paper, while benchmarking its performance versus other known solutions.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

Keywords and phrases VOSYSmonitor, ARM TrustZone, mixed criticality, real time

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.6

* This research work has been supported by the FP7 DREAMS project under the grant agreement number 610640.



29th Euromicro Conference on Real-Time Systems (ECRTS 2017).
Editor: Marko Bertogna; Article No. 6; pp. 6:1–6:18

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

1 Introduction

An important challenge in the design of embedded systems is the consolidation of software applications with different levels of criticality on a common hardware platform. In the automotive domain, a common practice to isolate safety critical applications is through the proliferation of multiple hardware Engine Control Units (ECUs) [8], which are dedicated to basic operations, such as lowering the windows, to critical tasks as Electronic Braking System (EBS), engine control and digital dashboard applications. This is a highly inefficient way of using the available processing power since many of these ECUs are typically not used at their full potential. However, recent multi-core architectures with new hardware extensions (e.g., virtualization, TrustZone) enable the execution of multiple applications on the same platform safely and securely, thus reducing costs and vehicle weight, helping to increase efficiency.

The consolidation of different OSes on the same platform implies the concurrent execution of a critical OS with stringent real-time requirements [7] with non-critical applications. As a matter of fact, connected cars are required to support safety-critical control functions, such as EBS and Electric Power Assist Steering (EPAS) that have to be securely isolated from the In-Vehicle Infotainment (IVI) system. Similarly in avionics, functions are usually classified either as *flight-critical* (necessary for ensuring a safe flight) or *mission-critical* (essential for business execution) [25]. In this context, the main challenges are to integrate real-time tasks execution with software applications of a GPOS.

In the past, virtualization has been presented as a solution to isolate OSes in a VM. This approach offers the advantages of reducing implementation costs by abstracting the host platform. Additionally, features provided by hypervisors, such as memory partitioning, CPU and interrupts abstraction help the OSes isolation. However, the use of a hypervisor may cause performance overheads, and therefore the critical execution path of real-time operations may not be ensured.

In this context, Virtual Open Systems has developed VOSYSmonitor, a software monitor layer, which enables the native concurrent execution of a safety critical RTOS (or another type of OS) along with a GPOS with the option to use virtualization extensions, such as Linux/KVM, in order to instantiate a variety of different VMs. The monitor layer is the highest secure operating mode available on ARM processors, designed with the hardware security extension ARM TrustZone [16], which manages the interaction between two execution worlds (see Section 2). In this context, VOSYSmonitor has been designed for the ARMv8-A architecture by guaranteeing peripherals and memory isolation between both OSes with ARM TrustZone. The main advantage of such a solution is to allow dynamically cores sharing between both applications, thus offering a close to native performance. To achieve this, VOSYSmonitor supports a context switch mechanism with a minimal overhead (see Section 3.3).

The rest of this paper is organized as follows. In Section 2, there is a brief introduction of the ARM TrustZone technology. Section 3 describes VOSYSmonitor architecture as well as its main features. Section 4 outlines the works related to this paper and emphasizes the advantages/drawbacks compared to VOSYSmonitor. Methods and benchmarks are explained and detailed in Section 5. Finally, Section 6 summarizes this work findings and directions for future works.

2 TrustZone

ARM TrustZone is a hardware security extension, which provides a system-wide security approach by integrating protective measures into ARM processors, bus and system peripherals.

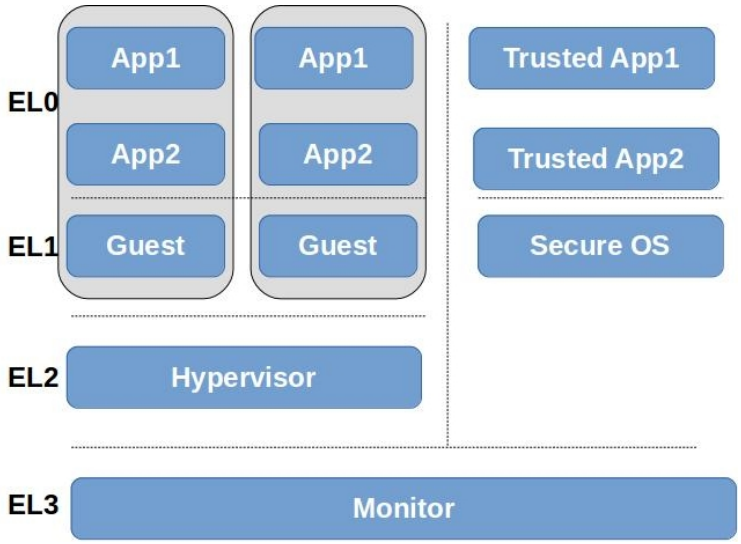


Figure 1 ARMv8-A Execution Level overview.

The security of the system is achieved by partitioning the hardware and software resources in two compartments: the Secure and the Normal worlds. The Secure world is usually used during the boot process in order to enforce a chain of trust. Indeed, starting with an implicitly trusted component, every other runtime binaries can be authenticated before their execution. In this context, some security specific configuration as well as sensitive data and peripherals can be only accessible from the Secure world. On the other hand, the Non Secure World is intended to host a rich operating system (e.g., Android or Linux). Security sensitive operations, such as the access to a private key or the interaction with a real time task, are provided to the non-secure application running in this compartment by the services run in the Secure World. Moreover, TrustZone enables a single core to safely and efficiently execute code from both worlds, allowing to save silicon area and power since a dedicated security processor is not needed.

While TrustZone is present since ARMv6 architecture [11], ARMv8-A provides a new architecture related to the TrustZone management. Indeed, a new highest Exception Level (EL) called EL3 (i.e., secure monitor mode) manages the interaction between these two compartments. This layer is always considered secure regardless of the current CPU state. Moreover, it manages the context switch between both worlds by triggering hardware exceptions, such as interrupts, synchronous and asynchronous events. The kernel level (EL1) and user level (EL0) are available in both worlds. This allows the execution of Trusted Execution Environment (TEE) in the Secure world, while the Normal world is expected to run a rich OS with the option to use virtualization extension, such as Linux-KVM, since the hypervisor mode (EL2) is only available in the Non-secure side.

The isolation provided by TrustZone is stronger than virtualization technology since this latter is restricted to the processor through the implementation of a hypervisor. Any other bus masters in the system, such as DMA peripherals or GPUs, can bypass the protections provided by the virtualization layer. In fact, ARM processors supporting security extensions along with TrustZone compliant Memory Management Unit (MMU) ensure the isolation of CPU execution mode, interrupts, hardware peripherals, memory and caches. However, the hypervisor can manage these peripherals for security reasons at the cost of introducing overhead.

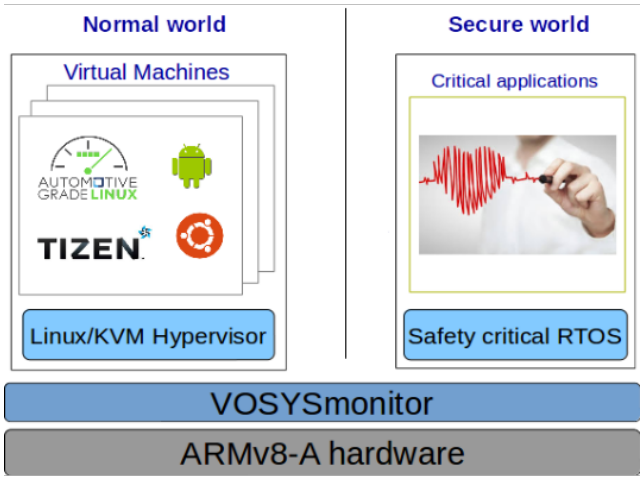


Figure 2 VOSYSmonitor overview.

3 VOSYSMonitor

In this section, VOSYSmonitor architecture as well as its interaction with the Secure/Normal worlds are described.

3.1 Architecture overview

VOSYSmonitor is a firmware that runs in the Secure Monitor mode (EL3) of ARMv8-A processors. As shown in Figure 2, it enables the native concurrent execution of two operating systems, such as for example a safety critical RTOS and a GPOS with the option to use virtualization extensions, such as Linux-KVM, in order to instantiate a variety of different VMs. VOSYSmonitor is a 64-bit monitor layer, which allows the execution of both 32-bit and 64-bit applications. Moreover, it ensures the isolation of each world by using the hardware security extension called TrustZone and provides, at the same time, functions to enable a safe and secure communication between them. Therefore the RTOS, running in Secure world, is totally isolated from applications executing in the Normal world. Finally, VOSYSmonitor manages the context switching between the two worlds by triggering a Secure Monitor Call (SMC) instruction or by hardware exception mechanisms, such as interrupts. VOSYSmonitor oversees these exceptions in order to ensure a correct operation for each world. The implementation of VOSYSmonitor is based on TrustZone to run another secure instance of an RTOS, while virtualization can be used as an additional option in the Normal world, such as XEN or Linux-KVM, giving the possibility to implement an additional layer of isolation between normal world applications.

It is important to note that VOSYSmonitor is not a boot loader. In this type of architecture, the boot process is usually split in different boot stages. Usually, the first two stages are platform specific and often provided by the board maker. As a matter of fact, the boot flow seen on Figure 3 is defined as follows:

- Primary Boot Loader(1) is the first stone of the chain of trust. It is generally implemented in the Read Only Memory (ROM), which is the only component that can not be modified or replaced by simple reprogramming attacks. It is responsible for initializing critical peripherals and authenticating the Secondary Boot loader located in external non-volatile storage.

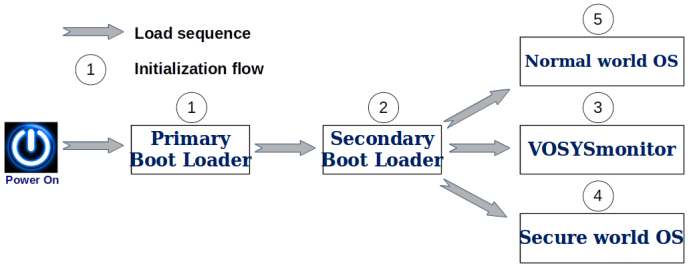


Figure 3 Trusted boot and load sequence on an ARMv8 platform.

- Secondary Boot Loader(2) is in charge to load and authenticate the different runtime binaries (e.g., Normal/Secure world applications and VOSYSmonitor), then it switches to VOSYSmonitor.
- VOSYSmonitor(3) performs basic initialization operations, such as ARM EL3 configuration, platform peripherals initialization and secure services setup, then, it gives the control to the Secure world(4) application. Once the latter has finished its own initialization, it requests a context switching through VOSYSmonitor in order to execute the Normal world(5) firmware/software (e.g., u-boot, UEFI, etc.).

VOSYSmonitor is designed to ease the support of new hardware platforms, as well as the integration of Trusted Operating Systems in the Secure world. Depending on the hardware requirements, VOSYSmonitor can reuse software components, such as common peripheral drivers, in order to minimize the integration effort. The top level architecture of VOSYSmonitor is seen in Figure 4, where different sub-systems of the firmware are depicted, including:

- EL3 Monitor Layer, mostly coded in ARM assembly, is specifically implemented targeting the ARMv8 architecture. It handles the world context switch operations triggered by SMC instruction or hardware exception mechanisms.
- Secure OS service layer is the interface between the Trusted OS, running in the Secure world (S-EL1), and VOSYSmonitor. The Secure OS service layer is in charge of the secure context initialization before jumping on the Trusted OS entry-point. Moreover, it is responsible for dispatching SMC services as well as to route interrupts reserved for the Trusted OS during its runtime.
- The Platform API is designed to abstract driver function calls from the core part of the monitor layer. Indeed, VOSYSmonitor requires access to peripherals, which can vary according to the hardware platform. Therefore, for modularity reasons the EL3 monitor layer uses these generic API functions, which, in turn, call specific driver functions.
- Drivers include all necessary drivers related to peripherals (e.g., interrupt controller, UART, power controller, etc.), which are used by the VOSYSmonitor runtime.

VOSYSmonitor is a low latency software monitor layer developed for the 64-bit ARMv8-A architecture and it already supports the last generation of ARMv8 platforms such as:

- ARM Juno Development board (2 Cortex-A57 + 4 Cortex A-53) [18],
- Renesas R-CarH3 board (4 x A57 + 4 x A53) – ISO-26262 (ASIL-B) compliant [29],
- Renesas R-CarM3 board (2 x A57 + 4 x A53) – ISO-26262 (ASIL-B) compliant [29],
- NVIDIA Jetson TX1 board (4 x Cortex-A57 in big.LITTLE configuration) [27].

Moreover, VOSYSmonitor has been designed to be compliant with the stringent ISO-26262 certification as well as to meet the following requirements:

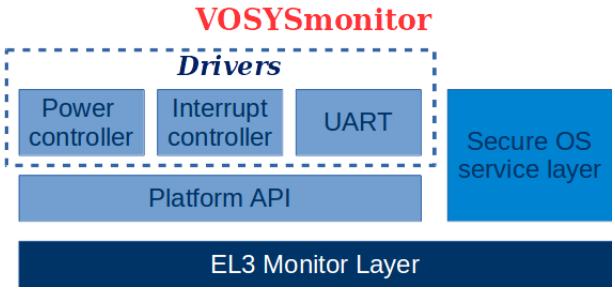


Figure 4 VOSYSmonitor top level architecture.

- Functional requirement. Safety critical OS (e.g., RTOS)/GPOS (e.g.,Linux-KVM) co-execution on the same platform.
- Functional requirement. Isolation of safety critical OS resources (e.g., Memory, Peripherals, etc.) from GPOS illegal access.
- Functional requirement. Preserve the context of each OS during a switching operation.
- Performance goal. A strong requirement for the RTOS in automotive is related to the boot time, which must be limited even in a worst-case scenario. Since VOSYSmonitor adds a software layer before the execution of RTOS, it directly impacts the RTOS boot time. In this context, VOSYSmonitor setup must be achieved in less than 1% of the full RTOS boot time. For instance, a RTOS boot time of 60 ms implies a VOSYSmonitor setup which has to be performed in less than 600us regardless of the platform.
- Performance goal. The overhead added by the co-execution of software applications must be optimized to meet real-time constraints. In this context, the overhead due to the context switching in order to forward an interrupt to the RTOS, running in the Secure world, has to be lower than 1us. This requirement is self-imposed and concerns a standard context switch where only the general-purpose registers and ARM systems registers are saved (see Section 3.3 for more details). This overhead can also be used for estimating the Worst Case Execution Time (WCET) of a task in the RTOS.

3.2 Secure/Normal world scheduling

On multi-core architectures, VOSYSmonitor is able to dynamically share a core between both worlds by operating under the assumption that the Secure world tasks should be prioritized over the Normal world execution. This means that once a core is assigned to the secure RTOS tasks, the normal world applications can use it only if the RTOS, running in the secure world, has decided to release the core resource; something that happens when there is no real-time task to schedule. For this implementation, a minor update of the RTOS is needed in order to call VOSYSmonitor service, to schedule the Normal world execution, when the RTOS workload is null. Generally, this could be achieved through the creation of an “idle task” with the lowest priority, which will be scheduled only if no other tasks need to be executed.

While giving the full priority to the Secure world may seem simple and have too much of an impact on the Normal world execution, VOSYSmonitor has been implemented under the assumption that tasks performing in the Secure world take precedence over the Normal world execution. If a task has no real-time requirement but requires to be isolated from others tasks (e.g. data encryption), it should be executed in the Normal world which can use the Virtualization Extensions to provide isolation.

For instance, tasks whose failure to meet a real-time requirement could arise to a life-critical situation (e.g. brakes control), should be executed in the Secure world. On the other hand, tasks whose failure to meet a real-time requirement does not lead to a life-critical situation (e.g. video decoding) should be performed in the Normal world.

Although multiple scheduling methods have been proposed for mixed-criticality systems [25], such a design has several benefits related to the execution of the safety critical RTOS. First of all, critical interrupts dedicated to the RTOS systematically preempt the Normal world execution in order to be handled by the RTOS with a minimum overhead (see Section 5), thus ensuring to meet real-time constraints. Moreover, there is no risk to preempt the RTOS execution during a critical operation since the RTOS releases the core only when it wants to. Finally, the core usage is optimized since VOSYSmonitor enables the scheduling of the Normal world application when the RTOS has no operations to execute. Indeed, real-time tasks are usually used to perform a brief action bounded in terms of time that could imply a low workload of the RTOS depending on the use case. However, this solution could lead the normal world to the impossibility to run its applications if the RTOS execution monopolizes the core. This is mitigated through the assignment of other cores to the Normal world GPOS.

3.3 Context switch

As ARMv8-A CPU execution is split in two parts (i.e., Secure/Normal world) and some registers are not banked, VOSYSmonitor has to preserve the world context during the switching operation. This part is the most performance related function of the system, as execution and RTOS interrupt latency are directly affected by the time consumed for the context save/restore operations. For this reason, most of code executed during the context switching is written in ARM assembly in order to reach the best performance. In the current implementation, VOSYSmonitor only saves the vital registers, such as general-purpose registers and some ARM system registers, needed for the correct RTOS/GPOS co-execution in each CPU operating mode. Indeed, for the test purpose, RTOS does not use advanced CPU features as the floating point context. However, a memory segment in the backup context structure is reserved to save additional registers during the context switching in order to extend VOSYSmonitor functionalities.

VOSYSmonitor periodically transfers the execution from one world to the other. As the RTOS/GPOS shares only a specific core, the context switch is currently tied to this processor and may take place in two main cases: an interrupt assigned to the Secure world is triggered during the execution of the Normal world application; one world requests a context switch (e.g., secure services, no RTOS workload, etc.) by calling VOSYSmonitor service through an SMC. In this context, it first saves the current state of the world suspended, then restores the state of the other world.

3.4 Interrupts management

ARMv8-A architecture including the hardware security extension TrustZone has been designed to support two interrupt types: the Fast Interrupt Request (FIQ) for low latency interrupt handling, and the more general Interrupt Request (IRQ), which is commonly available also in other architectures. The former has higher priority (IRQs are automatically masked by the CPU core when an FIQ occurs) and can directly use some banked registers without the overhead of saving/restoring them.

VOSYSmonitor takes advantage of the ARM architecture [20], which offers the ability to trap IRQ and FIQ directly in its exception layer (EL3) without intervention of code in

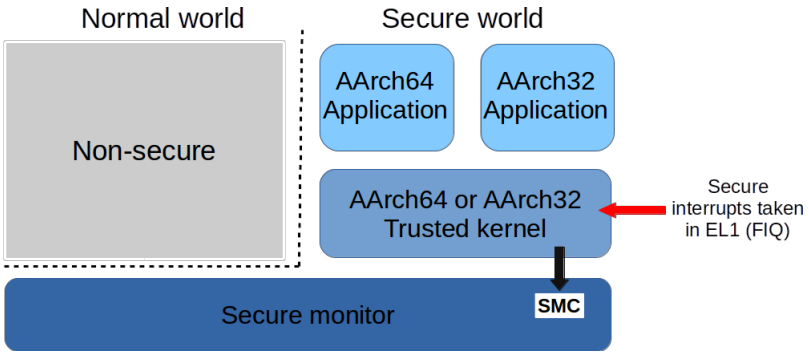


Figure 5 Secure world execution.

either world, thus allowing for the creation of a flexible interrupt management for safety critical RTOS tasks. Indeed, FIQs are considered as secure interrupts when TrustZone is supported, meaning that the configuration of FIQ cannot be altered by normal world accesses. Therefore, VOSYSmonitor sets the secure world (RTOS) to respond only to FIQs and the normal world (GPOS) to handle IRQs. This design allows critical applications to benefit from fast and high priority interrupts, while isolating them from the normal world.

3.4.1 Secure world interaction

During the Secure world execution, only FIQs are enabled in order to prevent the preemption of critical RTOS tasks by normal interrupts (IRQ) dedicated to the non-secure world. Indeed, on multi-core architecture, cores assigned only for Normal world applications can attempt to send normal interrupts for synchronization purpose, whereas the shared core executes RTOS tasks. This could generate untimely preemption of the RTOS during the critical path execution. Therefore, VOSYSmonitor prevents this side effect by masking all normal interrupts during the secure world execution.

FIQs are directly handled in the S-EL1 mode of the RTOS (see Figure 5. This minimizes the interrupts latency during the secure world execution since these are directly caught in the handler of the critical RTOS application, avoiding any context switch overhead. The Normal world only executes upon an explicit request by the Secure World that is achieved through an SMC instruction.

Although the execution of real-time tasks will not be impacted by non-critical applications, the main drawback of this implementation concerns the fact that VOSYSmonitor relies on the secure world application to induce a context switching. Indeed, VOSYSmonitor has no means to take back the control in case of the RTOS does not generate an SMC to release the core. With the current available ARMv8-A platforms equipped with the Generic Interrupt Controller (GIC) version 2 (e.g., Renesas R-Car H3), scheduling policies can be implemented (see Section 3.5) to allow the monitoring of the RTOS execution but such solutions either weaken the isolation of the safety critical RTOS or add an overhead during the context switch.

For these reasons, the current implementation depends on the safety critical RTOS to decide when the context switching should be performed. Moreover, safety critical RTOSes are generally compliant with the stringent requirements of an ISO standard (e.g., ISO-26262 Road vehicles – Functional safety [1, 2, 3]) in order to mitigate potential failures. However, last GIC version [15] (e.g., GICv3) enables the system to isolate interrupts addressed to the secure world from interrupts assigned to the monitor layer. In this context, VOSYSmonitor

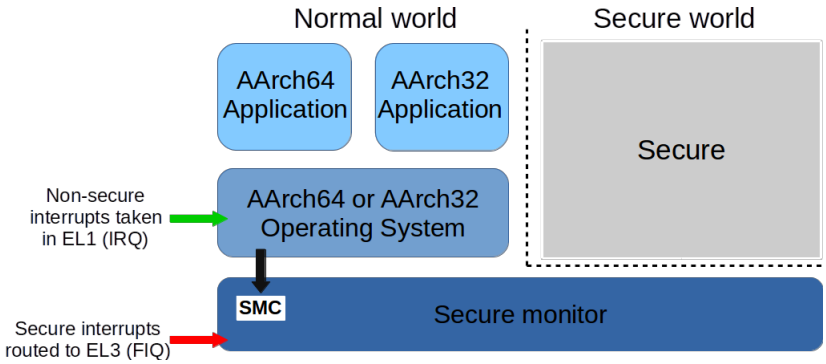


Figure 6 Normal world execution.

will be adapted and extended when platforms equipped with GICv3 will be available in order to implement a monitoring feature without compromising isolation, security and performance.

3.4.2 Normal world interaction

By contrast with the secure world, both FIQs and IRQs are enabled during the execution of the normal world: IRQs are directly handled in the NS-EL1 (see Figure 6) while FIQs are routed to VOSYSmonitor, which operates in EL3. When an FIQ occurs, the non secure OS is immediately preempted by VOSYSmonitor, which is responsible for an operating mode switch to save the normal world context and to restore the secure world one as fast as possible. This lowers latencies and helps the critical application to meet real time constraints. During the FIQ propagation into VOSYSmonitor, all interrupts (i.e., FIQ and IRQ) are masked in order to prevent any preemption by another FIQs having higher priority. Then, once the execution control is given to the Secure world, the FIQ management is handled by the trusted application. Therefore, the Secure world could decide to disable the FIQ mask when the critical part is over.

Finally, the normal world can initiate a context switch by generating an SMC instruction either to give back control to the secure world or to request a secure service.

3.5 Secure world failure handling

As mentioned in Section 3.2, the Secure world execution is always prioritized over the Normal world. However in some scenarios, the Secure RTOS might crash because of an internal failure and never giving the control back to the Normal world. To prevent such case, VOSYSmonitor proposes a Secure world failure handler, based on the ARM Physical Secure Timer. The handler execution is similar to a watchdog: if after a specific timeout the Secure world has not given back the control, VOSYSmonitor preempts the core and restart the Secure world. The handler timeout is reset at each context switch. However this feature is not present in the upstream version of VOSYSmonitor, for two reasons:

- Huge overhead of the context switch: when performing a context switch from the Secure world to the Normal world, the Secure Timer must be disabled. Likewise when returning to the Secure world, the Timer must be enabled and the timeout value calculated. It has been measured that adding the Secure Timer doubles the context switch latency.
- All interrupts in the Secure world are FIQs. Only the interrupt corresponding to the Secure Timer is configured as an IRQ, so it can be trapped by VOSYSmonitor. However this means IRQs are no longer masked during the Secure world execution.

When the system is mono-core, there are no issues as others IRQs can be disabled. Nevertheless, if the Normal world is running on a multi-core system, secondaries cores might forward IRQs to the primary core which is in the Secure world, causing undefined behaviour and possibly a crash of the RTOS. This could be avoided by updating the GPOS source code however this is against the policy of VOSYSmonitor which should be able to run a GPOS with minimal modifications. Furthermore, this would create a vulnerability exploitable from the Normal world.
In platforms supporting GICv3, interrupts are classified in three types, where one can be reserved to the VOSYSmonitor execution. In this configuration, it is possible to use the Secure Timer while retaining multi-core support for the Normal world. However all platforms currently supported are using GICv2. Thus for the time being the Secure Timer is only proposed as an optional feature.

3.6 ARM convention compliance

VOSYSmonitor is compliant with several ARM standards in order to ease the integration of this software component in a full system.

SMC Calling Convention (SMCCC) [19] specifies the calling procedure (e.g., registers used as parameters and return arguments, etc.) of the SMC instruction, used in the ARMv7 and ARMv8 architectures. This convention simplifies the integration between different software layers, such as operating systems, hypervisor and secure monitor. Moreover, it categorizes SMC service providers to allow the coexistence of services in the secure monitor firmware (e.g., ARM, OEM, Trusted OS, etc.)

Power State Coordination Interface (PSCI) [17] defines a standard interface for power management that can be used by software working at different ARM privilege levels. During a power management operation, rich OS, hypervisor, VOSYSmonitor and Trusted OS must not conflict each other. In this context, PSCI aims to standardize the communication between supervisory software to arbitrate power management requests. As a matter of fact, Linux kernel AArch64 relies on PSCI calls for powering up/down secondary cores avoiding the need of platform dependent drivers.

VOSYSmonitor is compliant with the PSCI convention v1.0, which requests the support of mandatory functions related to the power management such as power up/down a core, suspend core execution, etc.

4 Related work

VOSYSmonitor is a TrustZone based monitor, which enables the consolidation of a non-critical GPOS and a safety critical RTOS. It guarantees the isolation of critical real-time tasks, while minimizing the overhead on the global execution. Moreover, this software layer has been designed to meet certification requirements [6] induced by mixed-criticality systems.

Among existing solutions, TrustZone-assisted hypervisor [28] is able to run an arbitrary number of RTOSes in virtual machines. In this design, only one RTOS is executed at a time in the Normal world, while the context of the other guests is preserved in the Secure world. Therefore, if another RTOS needs to be scheduled, the current RTOS execution is stopped and its context is saved in the Secure world, then, the TrustZone-assisted hypervisor restores the second RTOS context in the Normal world. Real-time requirements of inactive RTOSes are met by setting their interrupts as secure, thus allowing to preempt the running RTOS, which uses normal interrupts. Such a implementation does not allow an efficient cache management since all RTOSes are scheduled in the Normal world, which requires

to clean/invalidate cache memory during context switches, thus increasing the overhead. Moreover, at the time of writing, the TrustZone-assisted hypervisor only supports single-core configuration. VOSYSmonitor is able to run on multi-core heterogeneous platform and take benefit of cache memory to reduce the context switch overhead. Indeed, since OSes are assigned to a specific world, VOSYSmonitor can rely on the isolation of caches lines provided by ARM TrustZone, thus avoiding the need of cache operations during a context switch.

Other solutions enable the co-execution of two or more OSes using ARM virtualization extensions. A design based on Erika OS [5] is executed along with Linux on top of XEN hypervisor [9]. This solution has been implemented on a cubieboard2 by assigning each OS on a different core. While it ensures the isolation, there is a risk of an inefficient use of computing power if Erika OS has a low workload. To prevent such a case, VOSYSmonitor is able to reallocate core resources to the GPOS if the RTOS has no real-time tasks to schedule. Others solutions based on virtualization extensions are Xtratum [24] and NOVA [31].

Furthermore, the isolation of Virtual Machines (VMs) against the host is not guaranteed. Indeed, some breach in the hypervisors can allow VMs to cause a denial service or access data from the whole system. As a matter of fact, a security vulnerability named VENOM [10], allows an attacker to escape from the isolation of a VM and get the access to the host and the others VMs.

ARM Trusted Firmware [22] hereafter referenced as ATF is a software layer able to host a Trusted Execution Environment alongside a Non-Trusted software. At the best of our knowledge, the current implementation of ATF only supports a dispatcher to execute OP-TEE OS [12] in the Secure world in order to provide trusted services to the Normal world application. VOSYSmonitor overcomes this limitation by giving the possibility to concurrently execute an RTOS and a rich OS. A further comparison between VOSYSmonitor and ATF is highlighted on Section 5.

A different approach, such as FLexPRET [32], introduces a new multi-threaded processor intended for mixed-criticality systems where threads are classified in two categories: *hard real-time thread (HRTT)* where deadlines must be met and *soft real-time thread (SRTT)* where a time constraint non-respected is acceptable. Another solution is IDAMC [26], which proposes a platform where multiples nodes are interconnected by a network-on-chip. These nodes includes processors based on the LEON3 technology connected through routers monitoring access to shared resources. While these solutions have been created for mixed-criticality systems, the main disadvantage is related to the new processor or cluster architecture, which implies, for instance, a higher workload for developers and increases the risk of unknown hardware/software bugs since the architecture is new and not widely used. On the other hand, VOSYSmonitor is based on ARMv8-A, one of the most popular embedded architectures for mobile, automotive, drone markets.

5 Evaluation

In this section, the performance of VOSYSmonitor is benchmarked with the ARM Trusted Firmware. The evaluation uses the ARMv8 Performance Monitoring Unit (PMU) [21] The tests have been performed on the ARM Juno board R1 and on the Renesas R-CarH3. Both boards have a Cortex A-57 and a Cortex A-53 cluster. As results may vary depending on the core where VOSYSmonitor is operating, all tests are performed on both A-53 and on A-57.

On top of VOSYSmonitor a bare-metal application is executed in the Normal world, which constantly loops a NOP instruction in order to have a minimal impact on the test execution. On the other hand, the Secure world is hosting a FreeRTOS version 8.2.3 modified

by Virtual Open Systems in order to use FIQ for interrupt processing. While it is not safety compliant, FreeRTOS is a widely used real-time kernel and is the base for an ISO 26262 certified RTOS called SAFERTOS [23].

Since VOSYSmonitor targets the automotive market, the compiler used to create the binary file should be qualified according to the ISO 26262 standard. For this reason, the ARM Compiler [13] with compilation optimisation (-O2) is used since functional safety support package [4, 14] will be provided by Q2 2017, thus avoiding further qualification activities when the recommendations and conditions are respected. Similarly, the upstream version of the ATF has been pulled from GitHub (v1.3) and also compiled with -O2 optimizations.

5.1 Context switch latency using PMU

VOSYSmonitor aims at having small impact on the co-execution of the Normal and the Secure worlds. The context switch is the most critical aspect in the VOSYSmonitor execution, therefore it is of utmost importance to implement it in a performant way. To assess the time induced by a context switch from the Normal world to the Secure, a timer in FreeRTOS is configured to generate an FIQ every 1us and then gives the control to the Non-Secure world. On Juno, the timer used is the SP804 Dual-Timer, on R-CarH3 the Compare Match Timer.

The timer interrupt is triggered while the processor execution is in the Normal world and thus it is trapped in VOSYSmonitor vector table. The PMU counter is started there and stopped before giving the control back to FreeRTOS, then the number of clocks cycles is displayed. The same test is performed with the ATF in order to compare the results. Adding the PMU breaks the code execution, therefore, the board has to be manually powered on/off after each measurement inducing a small sample size (< 10).

Although VOSYSmonitor includes the interrupt handler described in Section 3, the context switch latency has also been measured without this handler in order to gain time execution and assess the optimum value possible. This implementation makes that any FIQ trapped, when the processor execution is in the Non-Secure world, causes a request for a context switch. While lacking flexibility, this implementation may be of interest for scenarios where the RTOS response must be as fast as possible.

Figure 7 shows the context latency expressed in microseconds on the Juno board and Figure 8 for the Renesas R-CarH3 board. The results for the Juno board, shows that, in the worst case (VOSYSmonitor with interrupt handler on Cortex-A57), VOSYSmonitor is 118% faster than ATF and almost twice as fast if VOSYSmonitor without interrupt handler is executed on an A-53 core. On the R-CarH3, the comparison is in favor again of VOSYSmonitor, which can be more than thrice faster than ATF. However, it should be noted than the ARM Trusted Firmware used here is a software updated by Renesas, and thus may include features not present in the upstream version. VOSYSmonitor aims at having a context switch faster than 1us. Although on Cortex-A53 the requirement is met on VOSYSmonitor with or without an interrupt handler, a context switch on Cortex-A57 barely misses the prerequisite on Juno.

In this measurement, the interrupt handler called by VOSYSmonitor is a minimal implementation, which consists in fetching the pending interrupt ID, then after comparison, jumps to the context switch macro. Therefore, a decrease in performance can be expected if a more complex interrupt management is requested. However, the same can be said for ATF and the latency difference between both ATF and VOSYSmonitor should remain as it is.

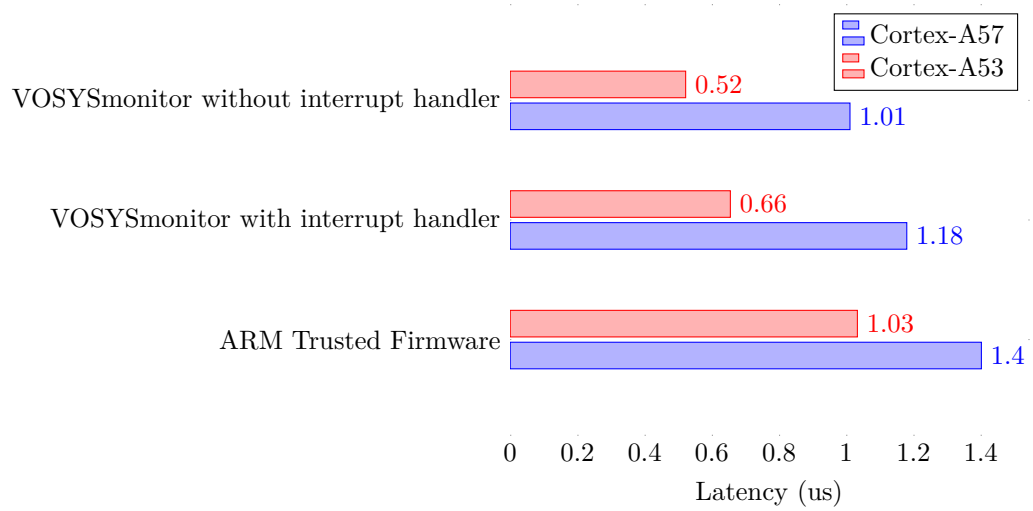


Figure 7 Juno context switch latency.

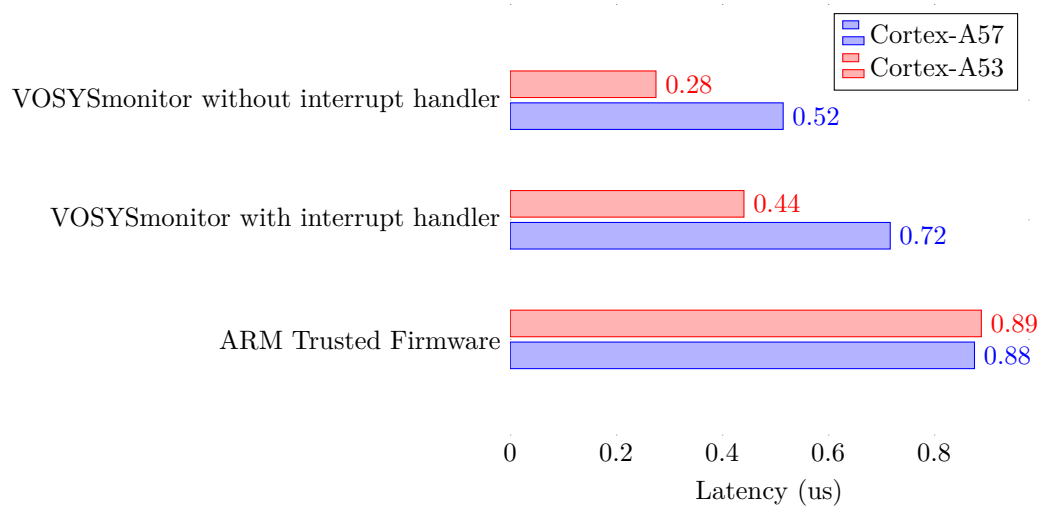


Figure 8 R-CarH3 context switch latency.

5.2 Context switch including the hardware latency

Although the previous measurement allows to measure the exact number of clock cycles consumed during a context switch, a second test has been performed to overcome some limitations of the first one. Indeed, VOSYSmonitor is running with both instructions and data caches enabled, which drastically improve performance. However, caches misses can cause a decrease in performance in worst-case scenarios, which can not be estimated unless the sample size is big enough.

In this context, a second performance test has been performed in order to take into account the FIQ latency induced by the hardware, i.e. the time between an FIQ triggering and the beginning of its processing by the software, on an important sample size. To achieve this, the ARM EL1 physical timer is used to trigger an FIQ while the core is in the Normal World as in the previous test.

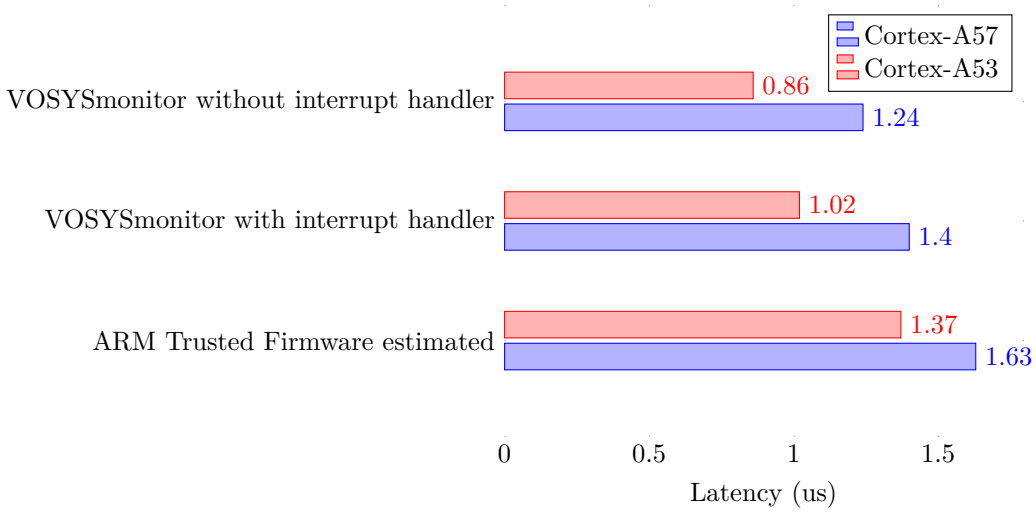


Figure 9 Juno board context switch with hardware latency.

Table 1 FreeRTOS FIQ latency.

Platform	Processor	Average (ns)	Min (ns)	Max (ns)
Juno	Cortex-A53	220	180	280
	Cortex-A57	220	200	260
R-CarH3	Cortex-A53	240	120	360
	Cortex-A57	240	120	360

The ARM timer is composed of three registers:

- CNTP_CTL_EL0 used to enable/disable the timer.
- CNTP_TVAL_EL0 contains the current value of the timer.
- CNTP_CVAL_EL0 holds the compare value. When equal to CNTP_TVAL_EL0, the interrupt is triggered.

When the context switch occurs, the timer interrupt trapped in VOSYSmonitor is forwarded to the FreeRTOS vector table. There, the very first instruction consists in reading the value of CNTP_TVAL_EL0, the timer is disabled and we jump to FreeRTOS handling routine. Finally, VOSYSmonitor latency between an FIQ triggering and the interrupt handler can be deduced by subtracting from CNTP_CVAL_EL0. Since VOSYSmonitor code has been untouched, we are able to run this test for as long as necessary and without any instrumentation of the code execution.

As before VOSYSmonitor is tested with and without the interrupt handler. The results with a sample size of 2048 contexts switch are presented on Figures 9 and 10. By taking into account the FIQ latency, it confirms that VOSYSmonitor is still faster than the ATF version in all scenarios. Moreover, it is better, in terms of performance, to have the RTOS executing on an A-53 core since the context switch is at least twice as fast as an A-57 for VOSYSmonitor. In this test, an issue has been the inability to run FreeRTOS and a Non-Secure OS in co-execution with ATF. However, the full context switch with hardware latency has been extrapolated by adding the previous results of ATF measured in Section 5.1 and the FIQ hardware latency shown in Table 1.

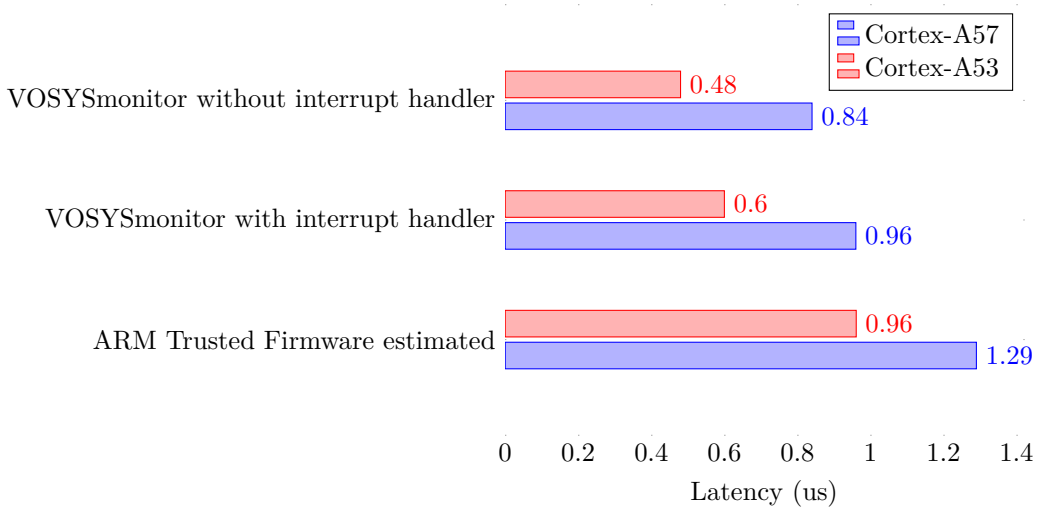


Figure 10 R-CarH3 board context switch with hardware latency.

Table 2 VOSYSmonitor and ATF setup time.

Platform	Processor	ARM Trusted Firmware (us)	VOSYSmonitor (us)
Juno	Cortex-A53	853.480	17.746
	Cortex-A57	1119.815	31.004
R-CarH3	Cortex-A53	1661.806	44.493
	Cortex-A57	1119.815	31.004

5.3 VOSYSmonitor and ATF booting time

The performance analysis also compared the setup time of VOSYSmonitor and ATF. The setup time corresponds to the time consumed between VOSYSmonitor/ATF entry point and the first jump to the Secure world. In this test, PMU has been used since the GIC and the timers are not yet configured. As for the previous tests, the measurements have been performed on both A-53 and A-57 cores. Table 2 shows that on Juno VOSYSmonitor is able to achieve its setup configuration within 18us, while ARM Trusted Firmware needs around 870us on Cortex-A53 and 920us on Cortex-A57. VOSYSmonitor is able to outperform ATF because the page tables, used by the Memory Management Unit, are defined by the developer and included statically during the compilation, whereas ATF generates the pages during the setup time.

Although it requires more effort and reduce flexibility, it is worth the trade-off since VOSYSmonitor impact is significantly reduced during the setup and meets the 600us boot time requirement presented in Section 3.1.

5.4 OSes co-execution workload and multicore support

RTOS/GPOS co-execution is ensured by sharing a core between these two OSes. The full priority is given to the RTOS, running in the Secure world, in order to meet real-time requirements. With this implementation, the GPOS execution can be impacted depending on the RTOS workload. In order to measure this impact, the *hackbench* [30] benchmark has been used in three scenarios:

Table 3 Juno board VOSYSmonitor hackbench result.

Linux one core			
Hackbench (s)	Linux standalone	Linux + FreeRTOS low workload	Linux + FreeRTOS 60% workload
	3.411	3.431	14.251
Linux multicore			
Hackbench (s)	Linux standalone	Linux + FreeRTOS low workload	Linux + FreeRTOS 60% workload
	0.500	0.498	0.596

- Linux standalone.
- Linux + FreeRTOS low workload. FreeRTOS requests a context switch every 1us and gives back the control immediately after.
- Linux + FreeRTOS 60% workload. The CPU will spend around 60% of the runtime executing FreeRTOS code.

Table 3 shows that adding FreeTOS with a low workload has no impact on the performance, whether using one or more cores. However, a deterioration can be noticed when FreeRTOS is executing 60% of the time on a core. By executing the test on a multi-core configuration (four Cortex-A53 and two Cortex-A57), we observe that the performance degradation is minimal between the Linux standalone and the 60% workload scenarios. Although a core is busy executing Secure world application, the others cores are nonetheless able to continue performing.

6 Conclusion

This paper presents VOSYSmonitor, a low latency monitor layer for mixed-criticality systems on multicore heterogeneous ARMv8-A platforms, providing architecture details and a performance analysis. VOSYSmonitor offers a strong isolation based on the ARM TrustZone technology, which gives the full priority to the safety critical RTOS, thus meeting real-time constraints as well as ensuring the execution of critical tasks.

The benefits of VOSYSmonitor are its modular and scalable architecture, which allows the system evolution according to the requirements. Indeed, it is possible to run, on top of VOSYSmonitor, a hypervisor in order to instantiate a variety of different VMs for multi-OS support (e.g., Linux, Android, etc.) in the Normal world. Moreover, the isolation provided by VOSYSmonitor is stronger than virtualization technology since this latter is restricted to the processor through the implementation of a hypervisor, whereas TrustZone can be extended to other master peripherals (e.g., DMA, GPU, etc.). Finally, the small footprint of VOSYSmonitor allows to mitigate the certification effort, thus reducing costs.

As for the analysis of the overhead introduced by the proposed solution, it is possible to claim that VOSYSmonitor is a perfect solution for the consolidation of real-time applications along with a rich OS whithout compromising hard real-time requirements. While it is not possible to compare with all technologies presented in Section 4, it has been proven that VOSYSmonitor is better than ARM Trusted Firmware in terms of setup time and context switching latency.

Finally, the future work includes the ASIL-C certification of VOSYSmonitor according to the stringent automotive standard ISO 26262. This will enable the usage of VOSYSmonitor technology in automotive use cases, such as the consolidation of Infotainment In-Vehicle

system along with safety vehicle cluster applications. With this in mind a full fledged software stack including VOSYSmonitor, an ASIL certified RTOS and GPU sharing support will be developed, tested and benchmarked. Related to the safety critical RTOS support, this work has given us the possibility to ensure the isolation of this software layer along with non-critical applications. However, only one safety critical RTOS can be executed with the current implementation. This problem will be investigated in future works, exploring design methods to overcome this limitation of the TrustZone hardware implementation. Regarding the world scheduling policy, the current architecture relies on the Secure world OS to initiate a context switch. Although this implementation ensures the execution of critical tasks without any preemption from another software layer, VOSYSmonitor lacks of a way to take back the control if the Secure world OS does not release the core resource. Software methods and new interrupt controller architecture, such as GICv3, will be explored to overcome this limitation due to the interrupt management.

References

1. International Standard ISO 26262-4. Road vehicles – functional safety – part 4: Product development at the system level. Standard, International Organization for Standardization, November 2011.
2. International Standard ISO 26262-6. Road vehicles – functional safety – part 6: Product development at the software level. Standard, International Organization for Standardization, November 2011.
3. International Standard ISO 26262-8. Road vehicles – functional safety – part 8: Supporting processes. Standard, International Organization for Standardization, November 2011.
4. Hopkins Andrew. The functional safety imperative in automotive design. Standard, ARM Ltd, September 2016.
5. Avanzini Arianna. Integrating Linux and the real-time ERIKA OS through the Xen hypervisor. *Industrial Embedded Systems (SIES)*, 2015. doi:10.1109/SIES.2015.7185063.
6. Sanjoy Baruah, Haohan Li, and Leen Stougie. Towards the design of certifiable mixed-criticality systems. In *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 13–22. IEEE, 2010.
7. Alan Burns and Rob Davis. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep*, 2013.
8. Helmut Fennel, Stefan Bunzel, Harald Heinecke, Jürgen Bielefeld, Simon Fürst, Klaus-Peter Schnelle, Walter Grote, Nico Maldener, Thomas Weber, Florian Wohlgemuth, et al. Achievements and exploitation of the AUTOSAR development partnership. *Convergence*, 2006:10, 2006.
9. Linux Foundation. The Xen Project, the powerful open source industry standard for virtualization. URL: https://www.xenproject.org/.
10. Jason Geffner. VENOM Virtualized Environment Neglected Operations Manipulation. URL: http://venom.crowdstrike.com/.
11. Richard Grisenthwaite. ARMv8 Technology Preview. In *IEEE Conference*, 2011.
12. Linaro. Op-tee. URL: https://wiki.linaro.org/WorkingGroups/Security/OP-TEE.
13. ARM Ltd. ARM Compiler 6. URL: https://developer.arm.com/products/software-development-tools/compilers/arm-compiler-6.
14. ARM Ltd. ARM Compiler Safety Package. URL: https://developer.arm.com/products/software-development-tools/compilers/arm-compiler/safety.
15. ARM Ltd. Programmable Interrupt Controllers: A New Architecture. URL: https://www.community.arm.com/processors/b/blog/posts/programmable-interrupt-controllers-a-new-architecture.

16 ARM Ltd. TrustZone. URL: https://developer.arm.com/technologies/trustzone.
17 ARM Ltd. *Power State Coordination Interface*, August 2012.
18 ARM Ltd. *Juno ARM Development Platform SoC*, r1p0 edition, June 2013.
19 ARM Ltd. *SMC Calling Convention*, June 2013.
20 ARM Ltd. *ARM Cortex – A Series*, March 2015. Programmer's Guide for ARMv8-A.
21 ARM Ltd. *ARM Architecture Reference Manual*, January 2016. ARMv8, for ARMv8-A architecture profile.
22 ARM Ltd. Github repository. https://github.com/ARM-software/arm-trusted-firmware, 2016.
23 HighIntegritySystems Ltd. SAFERTOS Safety Certified RTOS. URL: https://www.highintegritysystems.com/safertos/.
24 Miguel Masmano, Ismael Ripoll, Alfons Crespo, and J. Metge. Xtratum: a hypervisor for safety critical embedded systems. In *11th Real-Time Linux Workshop*, pages 263–272. Citeseer, 2009.
25 Malcolm S. Mollison, Jeremy P. Erickson, James H. Anderson, Sanjoy K. Baruah, and John A. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1864–1871. IEEE, 2010.
26 Boris Motruk, Jonas Diemer, Rainer Buchty, Rolf Ernst, and Mladen Berekovic. Idamc: A many-core platform with run-time monitoring for mixed-criticality. In *High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on*, pages 24–31. IEEE, 2012.
27 NVIDIA. *NVIDIA Tegra X1 Mobile Processor Technical Reference Manual*, November 2015.
28 Sandro Pinto, Jorge Pereira, Tiago Gomes, Mongkol Ekpanyapong, and Adriano Tavares. Towards a TrustZone-assisted Hypervisor for Real Time Embedded Systems. *IEEE Computer Architecture Letters*, 2016.
29 Renesas. *R-Car Series, 3rd Generation User's Manual: Hardware*, February 2016.
30 Russell Rusty. Ubuntu Manpage: hackbench – scheduler benchmark/stress test. URL: http://manpages.ubuntu.com/manpages/precise/man8/hackbench.8.html.
31 Udo Steinberg and Bernhard Kauer. NOVA: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*, pages 209–222. ACM, 2010.
32 Michael Zimmer, David Broman, Chris Shaver, and Edward A. Lee. PFlexPRET: A processor platform for mixed-criticality systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 101–110. IEEE, 2014.

A Hierarchical Scheduling Model for Dynamic Soft-Realtime Systems*

Vladimir Nikolov[1], Stefan Wesner[2], Eugen Frasch[3], and Franz J. Hauck[4]

1 **Institute of Information Resource Management, Ulm University, Ulm, Germany**
`vladimir.nikolov@uni-ulm.de`
2 **Instistute of Information Resource Management, Ulm University, Ulm Germany**
`stefan.wesner@uni-ulm.de`
3 **Institute of Distributed Systems, Ulm University, Ulm, Germany**
`eugen.frasch@uni-ulm.de`
4 **Institute of Distributed Systems, Ulm University, Ulm, Germany**
`franz.hauck@uni-ulm.de`

Abstract

We present a new hierarchical approximation and scheduling approach for applications and tasks with multiple modes on a single processor. Our model allows for a temporal and spatial distribution of the feasibility problem for a variable set of tasks with non-deterministic and fluctuating costs at runtime. In case of overloads an optimal degradation strategy selects one of several application modes or even temporarily deactivates applications. Hence, transient and permanent bottlenecks can be overcome with an optimal system quality, which is dynamically decided. This paper gives the first comprehensive and complete overview of all aspects of our research, including a novel CBS concept to confine entire applications, an evaluation of our system by using a video-on-demand application, an outline for adding further resource dimension, and aspects of our prototype implementation based on RTSJ.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems, D.4.1 Scheduling

Keywords and phrases real-time, scheduling, hierarchical, dynamic, ARTOS

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.7

1 Introduction

This work summarizes the scientific results of the ARTOS [5] project. Its objective was to develop and integrate adaptive resource-management mechanisms in a generic framework for soft real-time systems supporting a dynamic set of applications and tasks. As target platforms general purpose systems as traditional x86, ARM and PPC based platforms were supposed, which may range from small embedded and mobile devices, to desktop PCs and large scale servers. The result is an open and dynamic execution environment for soft real-time applications, which can be integrated on different system levels and used for a variety of scenarios.

* This research has been supported by Deutsche Forschungsgemeinschaft (DFG) under Grant No. HA2207/8-1 (ARTOS) and BMBF under Grant No. 01/H13003 (MyThOS).

The demand for such adaptive platforms is currently on the rise. A good example comes from the automotive sector. The consolidation of functions with mixed criticality on single powerful ECUs (electronic control units) is a prevailing requirement for manufacturers of head-unit and entertainment systems [14, 35]. For example, the central console of various brands already provides diverging functions concurrently, e.g. for telematics, connectivity, navigation, display, etc. In the near future the functional scope shall be dynamically extendable "over-the-air" without maintenance effort in a garage. Even passengers will be able to download and activate new third-party apps on their backseat entertainment devices. Many of them will be executed concurrently on single embedded devices and ECUs. The required resource-management mechanisms could be provided by the OS, a middleware or the application itself. Certainly, a working solution would fit a variety of further use-cases in the fields of embedded systems, robotics and even complex applications in data centres, as has been shown in [30].

Since applications and activities may freely arrive or depart during runtime, the system has to adapt itself to a varying computational load, while sustaining a steady application performance and best possible overall quality. Moreover, the exact resource requirements of the applications are assumed to be *unknown* until runtime and also may *fluctuate* during their execution. Reasons for such cost variations can be found not only in the non-deterministic behavior of the executional hardware (e.g. effects of non-uniform memory access, caching, virtual memory etc.). They are also caused by data dependencies and varying algorithmic complexity of tasks, or even on a software compositional level where applications' logical control flow may be strongly distracted by dynamic service discovery, qualification and execution. The latter particularly applies for dynamically composed apps in service-oriented environments. Such a system requires not only mechanisms for cost monitoring and admission control, but also adaptive resource reservation and degradation techniques for transient and permanent overload management.

The main scientific contribution of ARTOS is its approximation and scheduling framework which is subject of this paper. In ARTOS we primarily focused on CPU time management for a single processor. Anyway, plenty of the provided mechanisms are applicable to other resource types as well, e.g. network, memory or energy usage. We are currently working on an extension of our model for distributed and parallel applications (i.e. a multi-processor version) and multiple resource types.

However, the problem of optimal and adaptive resource distribution for a varying set of unknown real-time applications is hard even for the uniprocessor case and a single resource (i.e. CPU time). An appropriate cost approximation model for applications and tasks must hold as a decision basis independently from the time resolution of their events. An execution schedule has to be dynamically generated and updated on task arrival and departure, but (and even more important) also in case of cost fluctuations. On the other side, costs for (online) feasibility analysis have to be omitted and its occurrence clearly defined and planned. Furthermore, cost reservations for tasks and applications are crucial for their prioritization and isolation, especially in the case of overload and faults. On the other side, for optimal system utilization and admission the reservations have to be adjustable as well. However, unpredictable bottlenecks due to potential system over-provisioning are still possible and an appropriate overload management strategy is necessary in order to keep the system and the applications in a consistent state. Its reactiveness determines the probability of temporal faults and thus the overall system quality of service.

According to Buttazzo et al. [10] there are three basic mechanisms to handle overload situations, i.e. modify the periods of tasks (*elastic scheduling* [11]), omit certain jobs (*job*

skipping [20]) and degrade tasks workload on the algorithmic level. We developed a new generalized application model that supports different *operational modes*, and thus quality levels, of applications. The actual modes are centrally decided and adjusted by the system during runtime following priorities and their measured resource demands. Therefore, an optimization approach was developed which allows for a trade-off between the best possible system quality and its optimal utilization. This mechanism implements a sophisticated overload management strategy in our system based on a controlled and directed application degradation. The effective realization of the modes is finally up to the application developers, and can be implemented with any or as a combination of the previously mentioned techniques. In [28] we already presented a realization of multi-mode tasks with multiple versions.

The required monitoring, reservation and automatic degradation mechanisms for the applications were integrated as parts of a *hierarchical scheduling model*. In contrast to our previous publications on this topic (e.g. [31, 29, 27]), we present here the first comprehensive and complete overview of our approach, including recent results and improvements as well as several novel contributions. These will be explicitly accentuated in the paper. Our scheduler was integrated into a component-based and service-oriented software framework, which offers a generic platform for dynamic and open soft real-time systems [4]. Our framework was implemented based on the Real-Time Specification for Java (RTSJ) on top of a Linux system, a modified OSGi framework and Aicas's JamaicaVM [19]. The JVM offered us basic support for (a)periodic activities, deadlines, cost monitoring, events (e.g. overrun, deadline miss) and a fixed-priority–based scheduling. Our close cooperation with Aicas helped us to extend RTSJ 1.0.2 and its implementation in JamaicaVM where necessary [28], in order to improve the runtime support for our scheduler. However, our evaluation finally proves the functionality and adaptational capabilities of our system to unpredictable and varying load [26].

This paper is structured as follows: Section 2 introduces our system model and Section 3 gives a basic overview of our scheduler's functionality. Section 4 illustrates our cost approximation model and scheduling on the task-level. In Section 5 capacity reservation mechanisms for the applications and their modes are explained. Section 6 describes the decision of the optimal modes selection. Section 7 discusses our system implementation and in Section 8 different evaluation scenarios and results are presented. In Section 9 related work is assessed and discussed. Finally, in Section 10 we draw conclusions and present future work directions.

2 System Model and Scheduling Problem

We assume a dynamic set of n concurrent applications A_1, $A_2, ..., A_n$, and for each A_i there is a set of modes $M_{i,1}, M_{i,2}, ..., M_{i,m_i} \in \mathcal{M}_i$, which can be switched dynamically. Each application further consists of a set of k tasks $\tau_{i,k} \in \mathcal{T}_i$ also having different modes, e.g. with different periods $T_{i,k}$, deadlines $D_{i,k}$ and estimated costs $C_{i,k}$ with respect to each application mode $M_{i,j}$. We primarily focused on periodic tasks[1], which better suit our actual audio/video processing use-cases. Thus, an application mode manifests as a preconfigured *set of* particular *task modes*. In general, a higher application mode has a higher computational demand $B_{i,j}$ but delivers a better quality, e.g. a better video resolution. The quality is formalized by application-specific utility functions $\mathrm{u}_i(M_{i,j})$ given by developers. Their values are precomputed and normalized in the system as a relation of quality benefits between the different application modes, i.e. $U_{i,j} = \mathrm{u}_i(M_{i,j})/u_i(M_{i,|\mathcal{M}_i|})$. Each application A_i contains a

[1] Aperiodic tasks require a special treatment in periodic systems, e.g. aperiodic server mechanisms like Deferrable Servers [37].

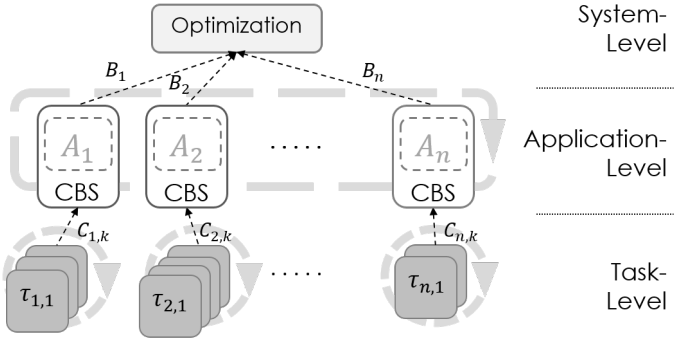


Figure 1 Scheduling Model Architecture.

mode $M_{i,0}$ with $B_{i,0} = \mathrm{u}_i(M_{i,0}) = 0$ which corresponds to the application being temporarily stopped[2]. Adding such a mode ensures the presence of an optimal resource distribution [31] and implements a fine-grained and dynamic admission control. Applications can be weighted by users or the system itself with importance factors a_i. These factors affect most obviously each applications prominence during the resource distribution process. For further discussion on conceptual and technical mechanisms for the realization of that model, we refer to [28].

2.1 The Scheduling Problem

Based on the previous system definitions our scheduler has to (a) monitor the actual resource requirements of the applications, (b) dynamically select an optimal modes configuration and (c) ensure schedulability of all application tasks. Obviously, a straight-forward approach would be to perform $|\mathcal{M}_1| \times |\mathcal{M}_2| \times \ldots \times |\mathcal{M}_n|$ feasibility tests for all application mode combinations and to select a feasible configuration that gains the maximal overall quality and optimally utilizes systems resources. However, since tasks may have deadlines not equal to their periods (i.e. $D_{i,k} \leq T_{i,k}$)[3] the complexity for solving the famously NP-hard feasibility problem escalates even further. Even with a fast approximative algorithm (e.g. [3]) solving the whole problem online may lead to inacceptable overhead. Also, it is not clear *when* and *how often* the feasibility of the actual configuration should be checked. For example, on each update of tasks (average) cost estimations? Without further precautions the latter may lead to permanent mode switches and reconfigurations of the system. Indeed, we assume that mode switches create additional *reconfiguration costs* for the apps, e.g. for data conversion, task adaptation, etc., and therefore they should be *omitted as far as possible*. Hence, a defensive reconfiguration strategy is needed, which tries to keep a stable system state as long as possible, but also is reactive enough to quickly detect and adapt on substantial state changes and errors.

3 General Scheduler Overview

Our hierarchical approximation and scheduling model is aimed to *temporarily* and *spatially* distribute the complexity of schedulability tests [29], i.e. the tests are performed only *when*

[2] Its tasks are blocked in a consistent state and temporarily removed from scheduler's control.
[3] $D_{i,k} \geq T_{i,k}$ was not considered due to implementational issues.

and *where* it is necessary in the system. Figure 1 depicts the general architecture of the scheduler. It follows abstractions of tasks, applications and operational modes on different levels. Application behaviour is monitored and statistically approximated on the lowest level. These approximations are forwarded to the upper levels, similar to a cascaded control system. Thereby, the information is further abstracted until on the uppermost level global decisions about the optimal configuration of application modes can be made. The basic idea is to assume schedulability as long as task approximations are valid and to postpone feasibility tests as long as their behaviour does not change considerably. If a task approximation gets invalid its cost estimate $C_{i,k}$ is updated and a local feasibility test is performed only for the concerned application A_i. This results in a new approximation of application's computational demand $B_{i,j}$ for its actual mode $M_{i,j}$. In this case, finding the optimal configuration of application modes on the system level is similar to filling a knapsack with items of different weights $B_{i,j}$ and values $U_{i,j}$.

On the lowest level, application-local schedulers control the execution of tasks and approximate their current costs. These cost estimations take also the observed jitter into account by allocating a dynamic resource buffer (cf. Section 4). As long as the approximations are accurate, no state update is forwarded to the upper scheduler layers. Only if a task bursts out of its current approximation bounds, a new cost estimation value $C_{i,k}$ is computed and notified on the application level.

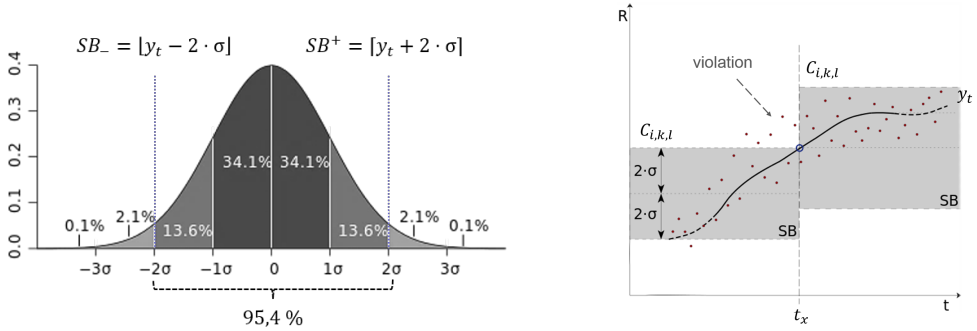
On the next higher level apps are isolated and scheduled with the help of *Constant Bandwidth Servers* (CBS) [1]. The CBS enforce resource reservations called *bandwidths*. We extended the original CBS-algorithm for encapsulation of whole applications. This requires a special methodology for dimensioning of the servers, which will be explained later in Section 5. In this paper, we present the latest version of our resource allocation mechanism. However, when a task cost estimation $C_{i,k}$ is updated, a local feasibility analysis results in a new bandwidth reservation $B_{i,j}$ for its application A_i. Thus, $B_{i,j}$ represents the required amount of resources in order to meet the deadlines of A_i's tasks. The new reservation is then forwarded to the optimization on the system level.

The objective of the optimization is to allocate the available system resources optimally. This is done by selecting and activating a set of app modes that maximize the system's overall utility constrained by its limited resource amount. For this purpose we developed a knapsack-based algorithm which is solved online via dynamic programming (see Section 6).

4 Task Approximation and Scheduling

As already explained, application-local schedulers control the activation of tasks $\tau_{i,k} \in \mathcal{T}_i$ according to a particular scheduling discipline. In fact, any arbitrary policy could be used, even different mechanisms for different applications at the same time. However, the task schedulers are an intrinsic part of our custom CBS mechanism. They determine the feasibility analysis mechanisms applied on the application level. In ARTOS we decided for EDF-based task schedulers. EDF allows for exact schedulability analysis even in case of a high system utilization. Since our framework was developed on top of a fixed priority scheduler (see Section 7), we used a modified version of the *priority shifting* mechanism proposed by Zerzelidis et al. [39, 41] in order to emulate EDF. Our current implementation with RTSJ is briefly discussed in Section 7.2 and further presented and evaluated in [28, 26].

Task schedulers also implement the approximation directives of our model. We instrument the thread-specific POSIX real-time clocks of the Linux kernel and sample the *pure* CPU costs C_t of each task job, i.e. scheduling and blocking effects are not included in C_t. These



(a) Standard deviation confidence intervals

(b) Safety Buffer with $2 \cdot \sigma$

Figure 2 Safety Buffers.

samples are then exponentially smoothed:

$$y_t = \alpha \cdot C_t + (1 - \alpha) \cdot y_{t-1} \tag{1}$$

The exponential smoothing implements a low-pass filter with an infinite impulse response. We chose a smoothing factor of $\alpha = 0.125$ which implies a stronger emphasis on the history $(1 - \alpha) \cdot y_{t-1}$ and therewith a relatively stable average value with respect to high-frequent jitter. However, for our actual task-cost estimation we need a measure for the observed jitter per task. The approximation is established with a dynamic *safety buffer* (SB) for guaranteed resources, which is defined as:

$$\mathrm{SB}_-^+ = [y_t \pm z \cdot \sigma_n] \tag{2}$$

where y_t is an expectation value based on the smoothed average from Equation 1 and σ_n is the standard deviation of the measured task costs C_t in a time window of n task periods. The actual task costs estimation is equal to the upper bound of the safety buffer $C_{i,k} = \mathrm{SB}^+$, while the lower bound SB_- prevents a constant overprovisioning of the costs. With this method fluctuations of the task costs can be algorithmically expressed and the factor z determines the quality of their approximation. According to the confidence intervals of normally distributed random variables a factor of for example $z = 2$ already covers ca. $95,4\%$ of the measured task costs (see Figure 2a). Hence, the actual task behaviour is approximated from the observed history of n periods with a quality of $95,4\%$.

Normal distribution is based on the central limit theorem which proves that the overlap of several independent stochastic processes is almost normally distributed [18]. Regarding the non-deterministic factors influencing the particular execution time of each task job, e.g. cache and TLB occupation, NUMA effects, potential blocking effects and spinning on OS-internal locks during system calls, this approximation model seems to be suitable. Thus, even if task costs are not normally distributed they are calibrated with the standard deviation leading to a certain deviation value σ_n.

In our model the cost estimations $C_{i,k}$ are not computed and updated on each task period. This would be costly because of the computation of σ_n for the last n period samples. In fact, only the exponentially smoothed values y_t are updated and periodically checked against their actual approximation bounds SB_-^+. These checks are part of the rescheduling process at the task level and have a low and constant complexity as well as memory demand. Consequently, each task verifies its own approximation limits on every periodic release. An established

SB is valid as long as the smoothed task costs y_t remain within its bounds (see Figure 2b). In case of $y_t \geq \mathrm{SB}^+$ or $y_t \leq \mathrm{SB}_-$ violations of the $2 \cdot \sigma$ SB-intervall must have occurred with a higher rate than expected, i.e. the task behaviour has changed so far, that it is not approximated correctly anymore. In this case a new deviation σ_n has to be computed and the buffer limits SB_-^+ must be updated (according to Equation 2). This results in a new cost estimation $C_{i,k} = \mathrm{SB}^+$ which is then notified to the application level. However, this strategy works on task-period granularity and approximates cost fluctuations precisely at the expense of a higher system reconfiguration probability. Other strategies may involve more seldom checks in order to reduce reconfigurational overhead and costs, however, at a risk for inaccurate task approximations and temporal faults.

5 CBS Extensions and Dimensioning

This chapter describes novel contributions of our work. As already explained, applications are isolated with separate constant bandwidth servers. A CBS is defined by the ratio of a reserved resource capacity R_i per server period P_i, i.e. a server *bandwidth* $B_i = R_i/P_i$. Since the bandwidths are automatically enforced by the CBS scheduling, applications are protected from overruns of other apps. Moreover, if an app needs more resources than actually reserved, it will not affect other apps but suffer alone from its deficiency. Hence, a mechanism is needed that 1) divides the available system bandwidth between the apps and 2) updates their reservations according to their actual demands. The first mechanism is implemented by the optimization, which decides the configuration of application modes on the system level. The second mechanism is conditioned by the app-local feasibility analysis which is triggered upon task cost updates (cf. Section 4).

Traditionally, the CBS were not designed to serve whole applications with multiple tasks and throughout literature they are mostly used for encapsulation of single tasks. Such a flat model would exacerbate the resource distribution between the applications since it requires additional application abstractions combining the reservations of their tasks. Moreover, it would also create a higher scheduling overhead for a considerable higher amount of servers. However, the original work of Abeni and Buttazzo [1] lists several clauses guiding the definitions and scheduling of the CBS. Indeed, we could not identify any reason or limitation why a CBS could not work well with multiple tasks in parallel. Consequently, we introduce the following specializations and enhancements of the algorithm:

1. **Internal Scheduling Discipline:** The server provides an internal queue for incoming jobs of *different* tasks. The ordering of the queue follows a particular scheduling strategy which does *not* have to be *non-preemptive*. At any time instant there is only one job of the same task in the queue – no interleaved job execution. The latter corresponds to the typical realization of tasks as threads within a particular OS.
2. **Server Dimensioning:** Regarding its capacity R_i and period P_i, a server must be dimensioned so that all internal tasks get enough computational time in order to meet their *own* deadlines. The feasibility of the internal task set must be approved for the allocated server bandwidth.

In ARTOS the first enhancement is fulfilled by the application local task schedulers. Each task scheduler maintains its own run queue ordered by EDF and the system maintains an absolute deadline for each task job. However, there are several points where the scheduling on task-level is interlinked with the CBS on application-level, i.e. on:

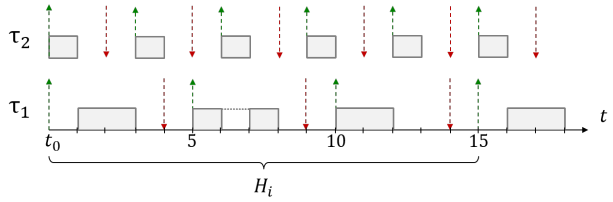


Figure 3 EDF activation example.

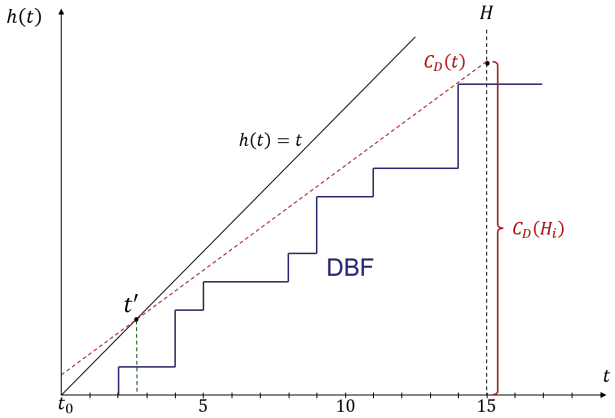


Figure 4 Example for app-local feasibility analysis.

- *Server Deactivation*: Each task scheduler tracks the termination of task jobs with no further pending jobs and notifies a server suspension (cf. Clause 8 in [1]) to the application level.
- *Server Activation*: On each task-job activation, the task scheduler tracks if the CBS is currently inactive and applies the *wake-up rule* if required (cf. Rule 7 in [1]).

Further details on the realization of the interconnection of task- and application-level scheduling can be found in [28, 26].

The second requirement is more complex and therefore described in more detail here[4]. It implies a relationship between the bandwidth of a server, i.e. its period P_i and capacity R_i, and the feasibility of its internal tasks with respect to their own relative deadlines. This relationship is established in our model based on tasks' processor demand [6]. The basic idea is to extract a required server capacity respecting tasks' actual cost estimations $C_{i,k}$ and internal deadlines $D_{i,k}$, which is then split over several server periods. Here we exploit the property of EDF that as long as task costs (estimations) remain unchanged their activation sequence remains identical within their hyperperiod H_i.

5.1 Dimensioning of the server capacity R_i

Figure 3 depicts an example of two tasks $\tau_{i,k} \in \mathcal{T}_i = (T_{i,k}, C_{i,k}, D_{i,k})$ with $\tau_{i,1} = (5, 2, 4)$ and $\tau_{i,2} = (3, 1, 2)$. Both tasks belong to the same application A_i and they are currently executed in a particular application mode $M_{i,j} \in \mathcal{M}_i$. The partial processor utilization of the task set is $U_i = \sum_k C_{i,k}/T_{i,k} = 0,73$ and the hyperperiod $H_i = 15$ is the LCM of the

[4] Again, we present here the latest version of our CBS dimensioning strategy.

task periods. In Figure 4 the cumulative demand bound function (DBF) [6] of the tasks is depicted as a step function. For simplicity we assume a synchronous task start, which is generally known as the worst case [17]. According to Baruah et al. [6] feasibility tests must be performed for all points of discontinuity of the DBF, i.e. the deadline points of the tasks, until $t' = U_i/(1 - U_i) \cdot max(T_{i,k} - D_{i,k}) = 2.75$ which is the intersection point of DBF's approximation tangent $C_D(t) = t \cdot U_i + U_i \cdot max(T_{i,k} - D_{i,k})$ and the angle bisector $h(t) = t$. The task set is feasible if for none of these points the according value of the DBF is greater than $h(t) = t$. As a required capacity for both tasks we use $C_D(H_i)$, which is the value of the approximation tangent in H_i. Hence, as long as the cost estimations of both tasks do not change (cf. Section 4), they need at least a capacity of $C_D(H_i) = 11.73$ in order to meet their deadlines in H_i and in every further repetition of H_i. The value of $C_D(H_i)$ is then normalized with the selected server period P_i and stored as R_i, i.e.:

$$R_i = P_i \cdot \frac{C_D(H_i)}{H_i} . \tag{3}$$

5.2 Dimensioning of the server period P_i

The choice of an appropriate server period is not as obvious as it might seem. In [26] we show based on several experiments, that the server period has to be less or equal the smallest task period within an application A_i, i.e. $P_i = min(T_{i,k})$. Only then enough server preemption points are created, so that their internal tasks can be activated timely in order to meet their internal deadlines. The normalization of the required capacity in Equation 3 can now be interpreted as follows:

- The system bandwidth is dissected over time in a capacity R_i per period P_i. The remainder capacity $P_i - R_i$ is available for all other apps in the system and their own server capacities.
- The capacity $R_i = c_g + c_r$ consist of a base part c_g for the task $\tau_{i,min}$ with the minimal period and a residual fragment c_r for all remaining tasks $\tau_{i,k}$ of A_i with greater periods.
- The base part $c_g = C_{i,min}$ guarantees the completion of task $\tau_{i,min}$ within each server period and, thus, within its own period and deadline.
- The residual fragment c_r consists of the sum of the partial costs of all remaining tasks of A_i, with respect to their own periods and deadlines, relative to the chosen server period, hence $c_r = \sum_{i=1}^{k} \frac{P_i \cdot C_{i,k}}{T_{i,k}} - C_{i,min}$.

Depending on the application-internal task activations and server preemptions during runtime, the actual job of $\tau_{i,min}$ or any other task of A_i may be split over multiple server periods. However, since R_i is warranted for each P_i, meeting of all task deadlines in A_i is guaranteed as long as its task costs do not change.

5.3 Interpretation of the Server Dimensioning

After setting $P_i = min(T_{i,k})$ the server capacity is computed and normalized according to Equation 3. Here the task hyperperiod H_i is inserted in the linear equation of the approximation tangent $c_D(t)$ of tasks DBF and normalized with the chosen period P_i. With the following transformation of the formula:

$$R_i = P_i \cdot \frac{c_D(H_i)}{H_i} \Rightarrow P_i \cdot \frac{(H_i \cdot U_i + U_i \cdot max(T_{i,k} - D_{i,k}))}{H_i} \Rightarrow$$

$$R_i = P_i \cdot U_i \cdot (1 + \frac{max(T_{i,k} - D_{i,k})}{H_i}) \tag{4}$$

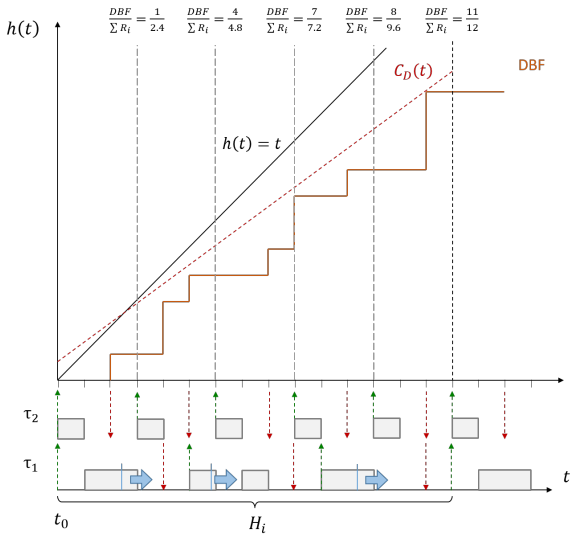


Figure 5 Visual interpretation of the server dimensioning.

it can be easily seen, that for tasks with deadlines equal to periods ($D_{i,k} = T_{i,k}$) the term $max(T_{i,k,l} - D_{i,k,l})/H_i$ gets zero and the required capacity is determined by the partial utilization $U_i = \sum_k \frac{C_{i,k}}{T_{i,k}} = B_i$ of A_i. According to Baruah et al. [6] in case of $D_{i,k} = T_{i,k}$ this necessary feasibility test is also sufficient. With $P_i \cdot U_i$ a base capacity for the $D_{i,k} = T_{i,k}$ case is reserved, which would be sufficient for the tasks in each hyperperiod. With $P_i \cdot U_i \cdot max(T_{i,k,l} - D_{i,k,l})/H_i$ a cumulative fraction of additional capacity per period P_i is reserved which in case of $D_{i,k} \leq T_{i,k}$ is necessary to meet the shorter task deadlines.

For our particular task example in Section 5.1 applying the defined server dimensioning rules leads to a server period $P_i = min(T_{i,k}) = 3$ and a server capacity $R_i = P_i \cdot \frac{c_D(H_i)}{H_i} = 2.4$ time units. Respectively, the allocated server bandwidth is $B_i = R_i/P_i = 0.8$. Figure 5 allows for a graphical interpretation of our reservation mechanism.

The time scale is subdivided in server periods of 3 time units. At each period an increase of 2.4 units of processor demand can be served. Figure 5 shows the relation between the cumulative processor demand increase DBF and the overall provided capacity $\sum R_i$ at the end of each server period. As a feasibility condition $\frac{DBF(t)}{\sum_t R_i} \leq 1$ must hold at the end of all server periods throughout H. For example, at the end of the second server period the task set demands for $DBF = 4$ time units, while the server guarantees so far $\sum R_i = 4.8$ TUs. Because of the capacity limitation of $R_i = 2.4$ per $P_i = 3$ the execution of some task jobs may be intercepted and postponed until the next server period. Since for all existing servers in the system the condition $\sum B_i \leq 1$ must hold true, the remainder capacity of $R_{rmd} = (1 - B_s) \cdot P_i = 0.6$ time units defines the longest possible preemption time of the server and its tasks in our scenario. For example, in Figure 5 in the first period a chunk of 0.6 time units of task costs is shifted to the second period, where afterwards a chunk of 0.2 is moved to the third, a.s.o. These costs are automatically reflected by the increase of tasks' processor demand at the respective server period. The actual preemption phases depend on the activation sequence of all servers and their internal tasks within the system. However, since the CBS scheduling (cf. [1]) automatically enforces the reserved bandwidths, meeting tasks' deadlines depends merely on the coverage of their DBF with sufficient capacity for each server period, independent of their actual activation and preemption sequence.

Our reservation mechanism automatically ensures that coverage by applying the presented processor demand approximation mechanism. However, other approximations such as the presented one by Albers et al. [3] and Chakraborty et al. [13] are possible and part of our future investigations.

6 Mode Decisions and Optimization

Mode decisions are reduced to a knapsack-based optimization problem, which is solved online and produces one of many exact solutions. Although the knapsack problem is NP-hard it can be solved via dynamic programming with pseudo-polynomial complexity. Our solution is described in detail in [31, 29, 26]. It computes an optimal mode selection in one pass and tends to distribute the resources uniformly, i.e. it prefers solutions with a higher admission rate and lower app modes instead of fewer active apps in higher modes gaining the same overall quality. The optimization is triggered when app's priorities or resource estimations $B_{i,j}$ change, or when new apps are activated or depart from the system. Our evaluation in [31] and [26] shows a linear increase of the computation time, when the amount of apps, modes, or system's resource resolution R increase. Thus, the overall computational complexity of the problem is $\mathcal{O}(n \cdot |R|) \cdot \mathcal{O}(|M_{i,j}|)$, where $\mathcal{O}(n \cdot |R|)$ denotes the amount of computed table entries and $\mathcal{O}(|M_{i,j}|)$ is the computational overhead per entry.

In the remainder of this chapter, as a novel contribution, we want to particularly focus a potential extension of our algorithm for further resource types with a multi-dimensional knapsack problem (MKP). Assuming one further dimension for application-specific network traffic[5] and demands, for example represented as mode specific bandwidths $N_{i,j}$, the problem could be defined as:

$$\begin{aligned} \text{Find a selection} \quad J = \; & \{M_{1,j_1}, M_{2,j_2}, ..., M_{n,j_n}\}, \\ \text{maximizing} \quad & \textstyle\sum_{i=1}^{n} a_i \cdot \mathrm{u}_i(M_{i,j_i}), \\ \text{subject to} \quad & \textstyle\sum_{i=1}^{n} B_{i,j_i} \leq R \\ \text{and} \quad & \textstyle\sum_{i=1}^{n} N_{i,j_i} \leq N. \end{aligned}$$

Since we work with bandwidth values, the numerical ranges of both resource dimensions R and N as also their knapsack size is assumed to be equal to 0..1 (i.e., 0 to 100 %). For the solution two matrices $\mathrm{d} \in \mathbb{Z}^3$ and $\mathrm{J} \in \mathbb{Z}^3$, are used for the computation. While d stores the values of partial iteration steps for all three dimensions, i.e. n-applications, R CPU resources and N network resources, J holds the optimal mode selection for each iteration step. For $i \in \{1, 2, ..., n\}$, $r \in \{0, 1, ..., R\}$, $k \in \{0, 1, ..., N\}$ and $M_{i,j} \in \mathcal{M}_i$ let

$$\mathrm{d}(i, r, k) = \textstyle\sum_i a_i \cdot \mathrm{u}_i(M_{i,j_i}) \tag{5}$$

denote the maximum utility such that $\sum_i B_{i,j_i} \leq r$ and $\sum_i N_{i,j_i} \leq k$. Let further

$$\mathrm{J}(i, r, k) = \{M_{1,j_1}, M_{2,j_2}, ..., M_{i,j_i}\} \tag{6}$$

be an optimal selection satisfying the given constraints. At each step the matrix $\mathrm{d}(i, r, k)$ holds the maximum possible utility for the first i applications and the resource boundaries of r and k, while $\mathrm{J}(i, r)$ stores the selection which led to the gained maximal utility.

[5] In practice, up- and down-link should be approximated separately, but we omit this here.

The matrices are initialized with $\mathrm{d}(0,r,k) = 0$ and $\mathrm{J}(0,r,k) = \emptyset$ for all $r = 0,1,...,R$ and $k = 0,1,...,N$. Then for $i \in \{1,2,...,n\}$, $r \in \{0,1,...,R\}$ and $k \in \{0,1,...,N\}$ the following recursion applies:

$$\mathrm{d}(i,r,k) = \max_{M_{i,j} \in \mathcal{M}_i} \{\mathrm{d}(i-1, r-B_{i,j}, k-N_{i,j}) + a_i \cdot \mathrm{u}_i(M_{i,j})\} \tag{7}$$

while with

$$\mathrm{J}(i,r,k) = \mathrm{J}(i-1, r-B_{i,j_i}, k-N_{i,j_i}) \cup \{M_{i,j_i}\} \tag{8}$$

an optimum realizing selection is given. Checking if $r - B_{i,j}$ and $k - N_{i,j}$ still reside within the matrix $\mathrm{d}(i,r,k)$ ensures observation of the constraints $\sum_{i=1}^{n} B_{i,j_i} \leq R$ and $\sum_{i=1}^{n} N_{i,j_i} \leq N$. In the following the pseudo code of the algorithm which was presented in [31, 29, 26] is extended with the new definitions:

Algorithm 1 Optimization Algorithm Pseudo Code

```
1: for r = 0 → R and k = 0 → N do
2:    d[0][r][k] ← 0, J[0][r][k] ← ∅
3: end for
4: for i = 1 → n do
5:    d[i][0][0] ← 0
6:    for r = 1 → R do
7:       for k = 1 → N do
8:          max ← 0
9:          for M_{i,j} ∈ M_i do
10:            b ← u[i−1][r − R_{i,j}][k − N_{i,j}] + a_i · U_i(M_{i,j})
11:            if b > max then
12:               max ← b
13:               J[i][r][k] ← J[i−1][r − R_{k,j}][k − N_{i,j}] ∪ {M_{i,j}}
14:            end if
15:         end for
16:         u[i][r][k] ← max
17:      end for
18:   end for
19: end for
20: return J[n][R][N]
```

As can be seen, we basically extended our previous algorithm with another dimension N. That leads to a further nested iteration loop over its value range (Line 7) and respective resource probing (Line 10). Consequently, with each new resource X the complexity of the algorithm is extended by a factor $|X|$, i.e. the numerical resolution of that resource. I.e., for our network resource based extension the computational complexity of our algorithm is $\mathcal{O}(n \cdot |R| \cdot |N|) \cdot \mathcal{O}(|M_{i,j}|)$, where $\mathcal{O}(n \cdot |R| \cdot |N|)$ denotes the amount of computed table entries and $\mathcal{O}(|M_{i,j}|)$ is the computational overhead per entry. MKPs generally suffer from the so called "curse of dimensionality", i.e. a substantial effort added with each further dimension. For that reason approximative dynamic programming approaches (like [33]) might help to solve the problem faster but at a risk of a suboptimal solution. However, such an approximative approach is part of our future work.

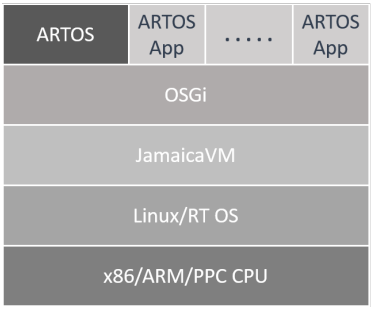


Figure 6 ARTOS Software Stack.

7 Implementation

We implemented our scheduling and application model and integrated them within a Java-based application framework and runtime system. In this Section, we describe the general hard- and software configuration, which was used also during our evaluation and experiments. We will then accentuate on some particular peculiarities of our implementation.

7.1 System Configuration

Figure 6 outlines the general software stack of our system. Our application and scheduling model were implemented with RTSJ 1.0.2 [8], which offers basic mechanisms for real-time application programming, i.e. periodic/aperiodic realtime threads, a FIFO priority-based scheduler, priority inheritance and ceiling monitor protocols, a bounded garbage collection interference, asynchronous events with controlled handler execution, cost monitoring and optional enforcement, server-like mechanisms and processing groups, etc. In order to support a dynamic provisioning and execution of applications and code, we decided for OSGi as a middleware. Its lightweightness, compositional abilities and service support allow for an easy integration of executable components (bundles) and Java-based code during runtime, without the necessity for a system restart. However, the ARTOS application API and scheduling code can be exported as bundles and, thus, integrated as an abstraction layer into any particular OSGi framework. In our experiments we used *Concierge* [15] which is a performance-optimized OSGi-R3-compliant framework for small and embedded devices. Applications which make use of our application and scheduling API *(ARTOS Apps)* are deployed as bundles within a running system and automatically registered within our scheduling primitives when activated.

JamaicaVM [19], which is developed by Aicas GmbH, is one of many RTSJ-compliant real-time JVMs. We chose this particular JVM for our experiments because it implements the full scope of the RTSJ spec, i.e. also its optional features like cost monitoring for threads and thread groups. These mechanisms were necessary for our custom CBS implementation on the application level of our scheduler. Beyond traditional interpreted execution JamaicaVM provides an *ahead-of-time* native compiler, which allows for better integration and runtime performance of Java programs and code especially on embedded devices. The compiled native binary of ARTOS contains a stripped version of the required RTSJ environment and runs as a root process on a particular OS and hardware platform. Thereby, RTSJ threads are mapped 1:1 to processes and threads of the underlying OS. Furthermore, the priority space of RTSJ can be mapped directly to a particular scheduling policy and respective priority

range[6] in a user-defined way. On this way, interference of other system and user processes can be avoided and controlled. On the other side, priority modifications of RTSJ real-time threads directly affect their native priorities and scheduling order within the underlying OS. JamaicaVM supports in general x86, ARM, and PPC platforms and a variety of operating systems, like Linux, VxWorks, QNX, etc. These configuration variants underline the broad range of applications supported alone by our system prototype. For our experiments we used a traditional Desktop PC with an Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz processor and 16 GB RAM. As an OS, we used a Linux 3.10.65 kernel, which was extended with the `CONFIG_PREEMPT_RT` [34] patch. Similar x86- and Linux-based configurations are applicable for a variety of embedded devices and they neatly integrate with JamaicaVM.

A traditional Linux kernel allows a process to preempt another process only at certain circumstances, i.e. (a) when the CPU executes user-mode code, (b) while leaving kernel-space, after a system call or interrupt has been handled, (c) when the kernel code blocks on a mutex or (d) explicitly yields control to another process. Thus, a long running low-priority process within the kernel may delay a request of a high-priority thread for hundreds of milliseconds, which would compromise the performance of any real-time system. With the `CONFIG_PREEMPT` option Linux allows kernel code to be entirely preemptable, except of particular spin-lock–protected regions and during interrupt handling. However, this lowers preemption latencies to a couple of milliseconds. For even lower latencies the `CONFIG_PREEMPT_RT` patch makes the kernel fully preemptable. For this, interrupt handlers are wrapped in kernel threads and processed with own fixed priorities next to user processes. Furthermore, the patch transforms particular kernel-intern locks to preemptive `rtmutexes` [36], which are enriched with priority-inversion avoidance protocols, like *priority inheritance*. JamaicaVM again builds on these mechanisms for its object monitors, threading system and internal scheduling.

7.2 Implementation Peculiarities

The implementation of our model based on RTSJ is presented more detailed in [28, 26]. However, for periodic tasks we used `RealtimeThread`s bound to special `PeriodicParameters` containing start, period, costs[7] and a relative deadline ($D_{i,k} \leq T_{i,k}$) definitions for each task. Task modes were realized with an RTSJ technique called Asynchronous Transfer of Control (ATC). ATC allows the definition of dynamically selectable and asynchronously interruptible code instances. Thus, multiple versions of the same task (i.e. modes) can be provided in parallel and are decided on each job activation. In case of a deadline miss or mode switch, the actual task job can be immediately interrupted. Beside basic task and mode abstractions our API provides semantics for applications consisting of several independent tasks with multiple modes. These apps are integrated with OSGi's module and lifecycle management mechanisms.

However, although RTSJ syntactically provides abstractions for custom thread scheduling, the implementation of such mechanisms is restricted [40]. As a consequence, we emulate EDF on top of RTSJ's fixed priority scheduler with a modified version of the *priority shifting* mechanism in [39, 41]. Thereby, the tasks perform a cooperative scheduling by shifting their own priorities and the priorities of their neighbours between three virtual priorities `preempted`, `active` and `scheduling`. First, an absolute deadline is maintained for each task and updated on each periodic release. For each application the tasks are ordered into a

[6] E.g. in Linux `SCHED_FIFO` [38] supports 100 fixed priorities whereas RTSJ prescribes at least 28 priorities.
[7] Since we did not rely on RTSJ's cost enforcement, the task costs are only informative.

separate run queue relative to their current absolute deadlines. On each suspension and activation a task $\tau_{i,k}$ is allowed to reassign the virtual priorities of all tasks in the same application A_i according to their current absolute deadlines, i.e. to select one active and to preempt all other tasks. For this, all tasks suspend at the end of each period and, thus, release again with a `scheduling` priority, capable to shortly preempt their neighbours and to re-schedule. The virtual application priorities are dynamically mapped and shifted by the system between certain regions within the fixed priority space, according to an application scheduling discipline. The latter is defined by the application level of our scheduler, i.e. the CBS scheduling mechanism.

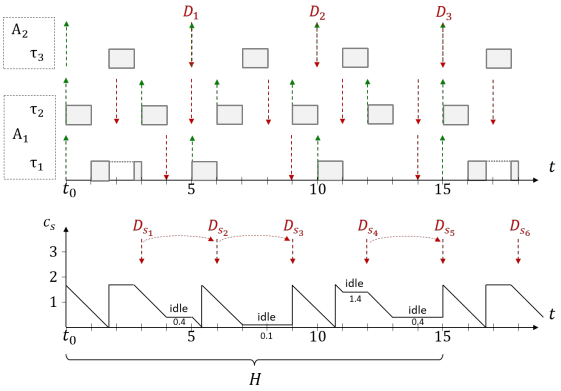
RTSJ provides basic support for custom server mechanisms through (implicit) *processing groups*. Each group is assigned a *budget* and a *reservation period*. The real-time JVM monitors the costs of each group and will suspend all threads if their budget is depleted. However, the budget is automatically replenished at the beginning of the next reservation period and the threads are then resumed. Consequently, this mechanism is not appropriate for a CBS, but for example for a Deferrable Server [37]. We extended the processing groups specification with custom budget replenishment and retrieval techniques for user-defined server mechanisms. The extensions were included in the latest RTSJ 2.0 version. With the help of Aicas, a modified version of the JamaicaVM supporting the suggested mechanisms was developed and used as a prototype for our evaluation. Our particular CBS implementation and scheduling are explained in detail in [28]. For scheduling we used a similar priority shifting mechanism like on the task-level, but now whole applications are moved between two *priority bands* (`ACTIVE` and `PREEMPTED`). As described in Section 5 our CBS scheduling is interlinked with the dispatching on task level where server activation and deactivation are traced per task and notified to the application scheduler for intervention. Budget depletion in turn is automatically tracked by the JVM for each processing group and notified via an asynchronous event. During runtime the CBS scheduling applies according to [1] and the extensions in Section 5 while the tasks perform their application-local cooperative scheduling.

8 Evaluation And Results

For our evaluation scenarios we developed special *artificial apps* which follow predefined load profiles, e.g. emulate certain costs, burst and jitter behaviour. Our experiments involved several tests with focus on (a) correct scheduling, i.e. the interplay between CBS and internal task scheduling, (b) systems adaptational capabilities and reactiveness, (c) suppression of cyclic task bursts and (d) the overall quality benefit of our model. The full spectrum of our experiments can be examined in [26], here we accentuate our latest results for (a) and (d).

8.1 Scheduling

We compared the theoretical schedules of different scenarios with the activation trace generated by our scheduler during runtime. For that we instrumented a `TraceMonitor` app provided by JamaicaVM. An example of its output can be seen in Figure 8. Our experiments involved: a single app with multiple tasks (task-level scheduling), multiple apps with single tasks (CBS scheduling) and multiple apps with multiple tasks (task and CBS scheduling). Here we show an example only for the last case, which involves scheduling on both task and application level. Two apps were involved: A_1 with two tasks $\tau_{i,k} \in \mathcal{T}_i = (T_{i,k}, C_{i,k}, D_{i,k})$:



(a) Theoretical schedule (all values in 10^2 ms)

t	Action
1.7	CBS_1 overrun and postponement of D_{s_1} to D_{s_2}
5.0	CBS_1 is *idle* with capacity rest of $c_s = 0.4$. Release of τ_1 and τ_3. Since $0.4 \leq (6-5) * 0.57$, τ_1 is served with rest capacity and D_{s_2}. For τ_3 the new deadline is D_2.
5.4	CBS_1 overrun and postponement of D_{s_2} to D_{s_3}. Since $D_{s_3} = 9 < D_2 = 10$, $\tau_1$and τ_2 in A_1 are then executed.
7	τ_2 suspends and CBS_1 remains *idle* with $c_s = 0.1$. A_2 and τ_3 are now active.
9	CBS_1 and CBS_2 are *idle* and all tasks suspend. τ_2 is activated and with $0.1 \geq (9-9) * 0.57$ $D_{s_4} = 12$ and $c_s = 1.7$ are set.
10.7	CBS_1 overrun and postponement of D_{s_4} to D_{s_5}. A_1 remains active since $D_{s_5} = D_3$.
11	A_1 suspends with $c_s = 1.4$. A_2 and τ_3 are now active.
12	τ_2 activates CBS_1 and with $1.4 \leq (15-12) * 0.57$ task τ_1 is served with its rest capacity and D_{s_5}.

(b) Comments on the execution

Figure 7 Artificial test scenario.

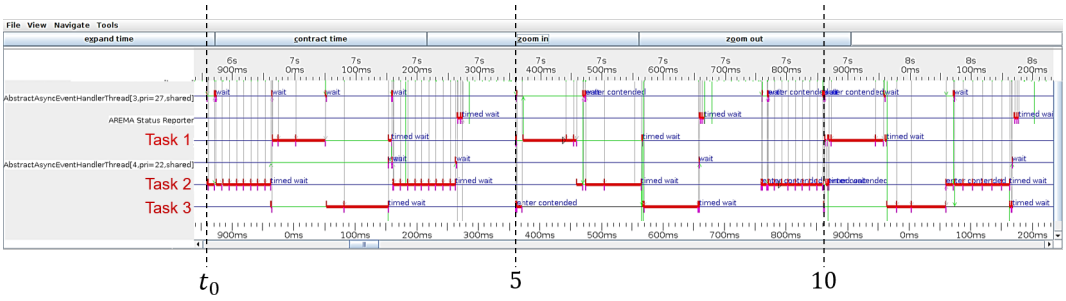


Figure 8 `TraceMonitor` output.

$\tau_1 = (5, 1, 4)$ and $\tau_2 = (3, 1, 2)$, and A_2 with one task $\tau_3 = (5, 1, 5)$[8]. The respective CBS configuration for both apps is shown in Table 1.

Figure 7a shows the theoretical execution plan of the applications and their tasks for a synchronous start, according to the original CBS rules in [1] and our hierarchical extensions in Sect. 5. On the bottom, the CBS state of A_1 is shown, the CBS of A_2 is of less interest, since it encapsulates only one task. Figure 7b shows a legend for certain time events and decisions. Several events occur in that example, e.g. server deadline postponements because of capacity depletion (e.g. $t = 1.7, 5.4, 10.7$) and server deactivations (e.g. $t = 4.0, 7.0, 11.0, 13.0$), as well as server activations with (e.g. $t = 0, 9, 15$) and without (e.g. $t = 5.0, 12.0$) capacity replenishment. The latter occur especially when the CBS wake-up rule applies. Even an application preemption can be seen at $t = 1.7$ because of a server deadline postponement in A_1. However, the schedule is valid and all tasks meet their own deadlines.

Figure 8 shows the real behaviour of our system (after some execution time). The task execution phases are outlined as red lines on the time line, and we also marked indices to the original time line of the theoretical schedule. The results show almost identical theoretical and runtime traces with slight deviations due to RTSJ's event handling procedure. Nevertheless, the dynamic resource budgeting for the apps assured a fault-free execution of their tasks in all experiments.

[8] All values are given in hundreds of milliseconds.

Table 1 CBS configuration for A_1 and A_2.

(in ms)	Period (P_i)	Capacity (R_i)	Bandwidth (B_i)
CBS_1	300	170	57 %
CBS_2	500	100	20 %

8.2 Adaptation

We emulated a sudden bottleneck caused by a burst or arrival of an artificial task. The reactiveness of our system is determined by the burst intensity and the smoothing factor α in Equation 1. A small factor requires a higher burst amplitude and a longer duration for its detection, but induces less reconfigurations. The actual adaptation duration is determined by the costs for 1) feasibility analysis, 2) optimization and 3) mode switches of the affected applications. The costs for feasibility analysis are non-deterministic and depend on the test limit t' (cf. Section 5.1), which in turn depends on the partial utilization U_i of the respective app and the maximal gap between tasks' periods and deadlines. The mode switch costs are application specific and non-deterministic as well.

8.3 Cyclic Bursts

Depending on the size of the history window for the standard deviation σ_n (cf. Equation 2) bursts may be considered as jitter within tasks' cost approximations. In this case the size of the approximation buffer SB^+_- increases. As a consequence, cyclic bursts are automatically approximated and recurring system reconfigurations suppressed [27]. The size of the σ_n window controls the frequency of the burst that can be balanced. Low frequent bursts need a bigger window but possibly lead to a stronger overprovisioning. Small windows in turn approximate more precisely, but lead to more frequent reconfigurations. An automatic adaptation of the window size according to the most prominent burst frequency after spectral analysis of tasks costs is currently in work. However, spare capacities caused by task costs overprovisioning can be instantly shared with other CBS based on a mechanism like CASH [12].

8.4 Overall quality benefit

We developed a real video-on-demand (VoD) streaming application based on our system model [26]. The VoD client was integrated within our framework and supported various modes with different video quality, e.g. different frame rate, resolution and frame quality. During runtime a bottleneck was created via an artificial app forcing the client to request a lower quality from the sender. The latter in turn immediately changes its video encoding format and adapts its sending rate to the requested frame rate. The experiments were done with and without the degradational mechanisms of our system. We supposed that a schedule that causes fewer deadline misses for the cost of lower video quality generally produces a better user experience (i.e. a higher utility), than one with the best video quality in presence of many temporal faults. For such a trade-off we compared frame-wise the peak signal to noise ratio (PSNR) of the received and decoded streams with their originals on the sender side. Our expectation was that, without degradation client's tasks will suffer more frequently from deadline misses leading to a higher signal corruption and thus lower PSNR value. With

Table 2 VoD application mode configurations.

Mode	Resolution	Framerate	CPU Bandwidth
1	480x270	10	10 %
2	1280x720	20	18 %
3	1920x1080	30	42 %

Table 3 PSNR results

Average PSNR	81.28
Maximum PSNR	99.00
Minimum PSNR	4.0
Average of Values < 99 dB	7.6

(a) without degradation

Average PSNR	72.08
Maximum PSNR	99.00
Minimum PSNR	7.06
Average of Values < 99 dB	44.66

(b) with degradation

degradation a lower overall quality is requested but there are no missing or corrupted parts in the received video, leading to a higher PSNR value.

In order to omit transmission errors and delay the experiments were performed within an isolated network. Table 2 shows the configuration of the quality levels of the VoD application and their estimated resource demands during runtime. According to the frame rate of each mode, clients tasks synchronize with periods equal to 100, 50 and 33 milliseconds and deadlines equal to periods ($D_{i,k} = T_{i,k}$). A miss may be caused merely by a deficit of execution time for the tasks. In such cases the client was instrumented to display and store a black frame for the respective video position, i.e. to mark a quality loss. In our experiments the VoD app was first started in its highest mode 3. A bottleneck was then caused with an artificial app which was activated at a certain time and led to a system overbooking of ca. 110 %. While without degradation the rest of the playback caused around 200 deadline misses within the VoD app, with degradation the app was immediately switched to mode 2 without further occurrence of temporal faults. Table II shows the results of the PSNR tests for the transmitted videos.

A bigger PSNR value shows a higher equivalence of the video signals, while a value of 99 dB in our example indicates almost identical parts. The results show that in both scenarios there were identical (max. PSNR) and completely different parts (min. PSNR). Obviously, the occurrence of deadline misses in our scenario without degradation was not frequent enough in order to drastically reduce the overall average video quality (avg. PSNR). Also, a lot of misses came at the expense of the artificial app itself which caused the bottleneck. With respect to that, the requested lower quality level in the degradation scenario reduced the PSNR to an even lower average value. However, for the parts where misses have occurred (avg. $<$ 99 dB) the average quality was improved by ca. 36 %. This value clearly depends on the applications' quality level definitions, the choice of one particular level as also the strength and the length of the bottleneck. Nevertheless, the results confirm that during transient bottlenecks our system preserves a better average quality than without its mode adaptation mechanisms.

9 Related Work

One of the first well-founded approaches for tasks with multiple modes and a quality driven scheduling are the *imprecise computations* (IC) [24]. A base-task quality is guaranteed by assuring mandatory parts of the tasks. This accords to an admission test which requires knowledge about their (worst-case) execution times. However, residual capacity is divided between optional task parts with the use of several heuristic approaches. Some for example tend to minimize the generated divergence from the best possible quality while others ascertain a particular probability for particular optional parts. The IC approach is easily reproducible with our model. However, what IC does not consider is a guideline when a real system must react and spend the effort for decisions and reconfigurations. Here we have a clear definition based on our approximation model. The consequence is that an IC-based system has to re-check its actual service quality periodically leading to a conflict between too frequent reconfigurations and a risk of suboptimality. Several system models base their mode-decision phase on IC. FCS [25] for example implements a feedback control loop, which periodically adjusts the overall system utilization and task quality levels.

Several related works have emerged out of the IRMOS and FRESCOR projects. In [2] and [32] a flat model of one task per CBS is presented with a control based approach for dynamically calibrating tasks' allocated bandwidths. A feedback loop per task is used to control its local scheduling error, i.e. the difference between its virtual finishing time and its job deadline. The objective of the controller is to keep the scheduling error per task with a certain probability within an acceptable range. For this, it uses a predictor to estimate tasks costs evolution and stepwise increases or reduces its allocated bandwidth. This includes also a recovery strategy with a finite error regeneration period for the case of a violation. A second loop with a supervisor is used to restrict an overbooking of systems overall available bandwidth. In case of an overload, the supervisor greedily resets reservations to at least minimal bandwidths for different classes of tasks. In [16] a version of the dual loop scheduler even supporting different task modes is presented. A global QoS controller periodically decides quality levels for the whole task set for different resource dimensions – even zero modes for the tasks are supported. Herefore, a greedy approach based on heuristics with a multipath search through the solution space is proposed.

The previous works ([2, 32, 16]) generally target at similar objectives as our model. However, our basic application abstraction provides a practical solution for tasks that have to be configured differently and synchronized accordingly for each particular application mode. This is not supported by the cited works where modes are effectively decided periodically on a per-task basis. However, consolidating tasks' requirements to common application bandwidths allows for a simple knapsack-based algorithm for the selection of optimal application modes (see Section 6). Since hereby the size of the problem domain is considerably reduced, this promises a lower complexity and a faster solution of the optimization problem. Instead of convergence to an acceptable result using a multipath search through solution space, our optimization directly computes an exact configuration, even in the multi-dimensional case. However, how the presented control mechanisms in [2, 32, 16] would apply for a set of tasks being concurrently served by the same CBS remains open for discussion. Since the tasks now interfere within the same server activation period, their scheduling errors have to be convoluted leading to a common server budget. For that, our approach proposes a simple mechanism based on tasks' processor demand and an application-local feasibility test, while the actual budget calibration steps are minimized. In fact, the algorithm in [16] penalizes applications frequently switching modes in order to minimize re-configuration costs and

attain a steady QoS. This may obstruct the quality of a potential optimal solution. Our model in turn provides an adaptive cost overprovisioning mechanism on task level in order to suppress frequent re-configurations, which is even capable to smooth periodic task bursts, as shown in Section 8. Thus, mode switches occur with the probability of tasks to unexpectedly burst out of their approximations or in case of recurring burst with a variable rate.

Recently, the `SCHED_DEADLINE` [21] scheduler in Linux was extended with a herarchical group scheduling mechanism based on the Linux *CGroups*. The originally flat model of this scheduler, which encapsulates each task in its own CBS, was extended with a second-level scheduler per CBS, i.e. while globally scheduled with EDF the servers allow for an internal fixed-priority–based task scheduling. However, while ongoing works focus on appropriate group scheduling in the presence of multiple processors, `SCHED_DEADLINE` still does not support dynamic reservations for the groups according to actual task demands. Consequently, server bandwidths are configured and allocated statically in advance, guided by respective CGroup definitions, while applications are admitted within the system. Thereby, a simple admission test based on partial task utilizations assures that the system is not overloaded.

In [23] a hierarchical CBS approach (H-CBS) is presented which guarantees temporal isolation between subsets of tasks (i.e. applications). The original CBS mechanism is extended by a virtual time and a scheduling deadline per task, and an excess capacity per task subset. Partial task utilizations are known and used for initial capacity setup, while the virtual times control the observance of task-specific reservations. However, all tasks are scheduled globally by EDF according to their scheduling deadlines, and when a task executes it uses the excess capacity of its subset (application). Meanwhile, H-CBS maintains different execution states for the tasks and according rules for updating their virtual time and deadline, as well as the excess capacity of the whole set. The algorithm ensures temporal isolation on task level as well as between different task subsets. However, the H-CBS algorithm seems still to support a flat task execution model, while the group relationship is encoded within the virtual time and excess capacity update rules for each task. In our approach we have one CBS per app instead with an internal scheduling discipline and run queue. However, it is not clear how the H-CBS mechanism would have to be extended in order to respect arbitrary task deadlines (e.g. $D_{i,k} \leq T_{i,k}$), since now temporal correctness on task level would be compromised. On the other side, no effort has been spent on re-calibrating the excess capacity according to actual task demands (initial task utilizations are still used for all updates). The partial utilization of a subset is assumed to be a static portion of the shared processor resource.

There is a magnificent number of works on optimal CBS dimensioning, for example such minimizing the amount of task preemptions [9] or avoiding the so called *deadline aging* problem [7]. What can be remarked here is that due to the optimal and dynamic adjustment of application capacities we observed a much lesser amount of CBS deadline postponements, which antagonizes the deadline aging problem.

10 Conclusion

In this paper we presented latest improvements of our hierarchical approximation and scheduling model for open soft real-time systems with a dynamic set of applications and tasks supporting multiple execution modes. These enhancements particularly apply to the dynamic server dimensioning and hence resource reservation strategy for applications. Moreover, we exemplified how our optimization mechanism, which is used for selection of an optimal application mode configuration, can be extended for further resource types, e.g. network bandwidth. We also explained the particular configuration and peculiarities

of our current implementation, which neatly integrates as a component-based real-time application framework on various OS and hardware architectures. Besides, latest evaluation results and experiments were discussed in particular proving the correctness of our scheduler and the adaptivity of our system on different load and jitter conditions. A concrete VoD application scenario was used to examine the overall quality benefit of our system especially in overload conditions. We are currently extending our model for distributed compute clusters and real-time HPC applications with multiple parallel tasks based on RT-DLT [22]. Our approach promises to enable a higher overall cluster utilization and throughput while retaining the best possible application quality and performance.

References

1 L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings 19th IEEE Real-Time Systems Symposium*, RTSS'98, pages 4–13. IEEE Computer Society, 1998. doi:10.1109/REAL.1998.739726.

2 L. Abeni, T. Cucinotta, G. Lipari, L. Marzario, and L. Palopoli. Qos management through adaptive reservations. *Real-Time Syst.*, 29(2-3):131–155, March 2005. doi:10.1007/s11241-005-6882-0.

3 K. Albers and F. Slomka. Efficient Feasibility Analysis for Real-Time Systems with EDF Scheduling. In *Proceedings of the Conference on Design, Automation and Test in Europe – Volume 1*, DATE'05, pages 492–497. IEEE Computer Society, 2005. doi:10.1109/DATE.2005.128.

4 AREMA – Adaptive Runtime Environment for Multimode Applications. https://sourceforge.net/projects/arema/, September 2016.

5 ARTOS. https://www.uni-ulm.de/en/in/vs/res/projects/artos/, April 2017.

6 S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings 11th Real-Time Systems Symposium*, pages 182–190, Dec 1990. doi:10.1109/REAL.1990.128746.

7 A. Biondi, A. Melani, and M. Bertogna. Hard Constant Bandwidth Server: Comprehensive formulation and critical scenarios. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, pages 29–37, June 2014. doi:10.1109/SIES.2014.6871182.

8 G. Bolella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000.

9 G. Buttazzo and E. Bini. Optimal dimensioning of a constant bandwidth server. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 169–177, Dec 2006. doi:10.1109/RTSS.2006.31.

10 G. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency*. Springer US, 2005. doi:10.1007/0-387-28147-9.

11 G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Trans. Comput.*, 51(3):289–302, March 2002. doi:10.1109/12.990127.

12 M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proceedings of the 21st IEEE Conference on Real-time Systems Symposium*, RTSS'10, pages 295–304. IEEE Computer Society, 2000.

13 S. Chakraborty, S. Kunzli, and L. Thiele. Approximate schedulability analysis. In *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.*, pages 159–168, 2002. doi:10.1109/REAL.2002.1181571.

14 S. Chakraborty, M. Lukasiewycz, C. Buckl, S. Fahmy, N. Chang, S. Park, Y. Kim, P. Leteinturier, and H. Adlkofer. Embedded systems and software challenges in electric vehicles. In

Proceedings of the Conference on Design, Automation and Test in Europe, DATE'12, pages 424–429. EDA Consortium, 2012.

15 Concierge OSGi. http://concierge.sourceforge.net/, April 2009.

16 T. Cucinotta, L. Palopoli, L. Abeni, D. Faggioli, and G. Lipari. On the integration of application level and resource level qos control for real-time applications. *IEEE Transactions on Industrial Informatics*, 6(4):479–491, Nov 2010. doi:10.1109/TII.2010.2072962.

17 L. George, N. Rivierre, and M. Spuri. Preemptive and Non-Preemptive Real-Time Uni-Processor Scheduling. Research Report RR-2966, INRIA, 1996. Projet REFLECS.

18 N. Henze. *Stochastik für Einsteiger*. Springer Fachmedien, Wiesbaden, 10 edition, 2013. doi:10.1007/978-3-658-03077-3.

19 JamaicaVM. https://www.aicas.com/cms/en/JamaicaVM, February 2017.

20 G. Koren and D. Shasha. Skip-over: algorithms and complexity for overloaded systems that allow skips. In *Proceedings 16th IEEE Real-Time Systems Symposium*, pages 110–117, Dec 1995. doi:10.1109/REAL.1995.495201.

21 J. Lelli, C. Scordino, L. Abeni, and D. Faggioli. Deadline scheduling in the linux kernel. *Software: Practice and Experience*, 46(6):821–839, 2016. doi:10.1002/spe.2335.

22 X. Lin, A. Mamat, Y. Lu, J. Deogun, and S. Goddard. Real-time scheduling of divisible loads in cluster computing environments. *Journal of Parallel and Distributed Computing*, 70(3):296–308, March 2010. doi:10.1016/j.jpdc.2009.11.009.

23 G. Lipari and S. Baruah. A hierarchical extension to the constant bandwidth server framework. In *Proceedings Seventh IEEE Real-Time Technology and Applications Symposium*, RTAS'01, pages 26–35. IEEE Computer Society, 2001. doi:10.1109/RTTAS.2001.929863.

24 J. W. S. Liu, W. K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise computations. *Proceedings of the IEEE*, 82(1):83–94, Jan 1994. doi:10.1109/5.259428.

25 C. Lu, J. A. Stankovic, S. H. Son, and G. Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms*. *Real-Time Syst.*, 23(1/2):85–126, July 2002. doi:10.1023/A:1015398403337.

26 V. Nikolov. *Ein hierarchisches Scheduling Modell für unbekannte Anwendungen mit schwankenden Ressourcenanforderungen*. PhD thesis, Ulm University, 2016. doi:10.18725/OPARU-4099.

27 V. Nikolov, F. J. Hauck, and L. Schubert. Ein hierarchisches Scheduling-Modell für unbekannte Anwendungen mit schwankenden Ressourcenanforderungen. In *Betriebssysteme und Echtzeit*, Informatik aktuell, pages 49–58. Springer Berlin Heidelberg, 2015. doi:10.1007/978-3-662-48611-5_6.

28 V. Nikolov, F. J. Hauck, and S. Wesner. Assembling a framework for unknown real-time applications with rtsj. In *Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES'15, pages 12:1–12:10. ACM, 2015. doi:10.1145/2822304.2822318.

29 V. Nikolov, K. Kempf, F. J. Hauck, and D. Rautenbach. Distributing the Complexity of Schedulability Tests. In *21st IEEE Real-Time and Emb. Techn. and Appl. Symp.*, RTAS 2015, page 2. IEEE, 2015.

30 V. Nikolov, S. Kächele, F. J. Hauck, and D. Rautenbach. Cloudfarm: An elastic cloud platform with flexible and adaptive resource management. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, UCC'14, pages 547–553. IEEE Computer Society, 2014. doi:10.1109/UCC.2014.84.

31 V. Nikolov, M. Matousek, D. Rautenbach, L. D. Penso, and F. J. Hauck. ARTOS: system model and optimization algorithm. Technical Report VS-R08-2012, Inst. of Dist. Sys., University of Ulm, 2012.

32 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.

33 W. B. Powell. What you should know about approximate dynamic programming. *Naval Research Logistics (NRL)*, 56(3):239–249, 2009. doi:10.1002/nav.20347.

34 PREEMPT_RT. https://rt.wiki.kernel.org, January 2016.

35 Wind River. A Smart Way To Drive ECU Consolidation. https://www.windriver.com/whitepapers/automotive/a-smart-way-to-drive-ecu-consolidation/, January 2017.

36 RT-mutex implementation design. https://www.kernel.org/doc/Documentation/locking/rt-mutex-design.txt, January 2017.

37 J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Comp.*, 44(1):73–91, January 1995. doi:10.1109/12.368008.

38 The Open Group and IEEE. IEEE Std 1003.1. The Open Group technical standard base specification, Issue 6, Base Definitions. http://www.opengroup.org/onlinepubs/009695399/mindex.html, 2004.

39 A. Zerzelidis and A. J. Wellings. Getting more flexible scheduling in the RTSJ. In *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*, pages 8 pp.–, April 2006. doi:10.1109/ISORC.2006.38.

40 A. Zerzelidis and A. J. Wellings. Getting more flexible scheduling in the RTSJ. In *Proc. of the 9th IEEE Symp. on Obj.-Oriented Real-Time Distrib. Comp.—ISORC*, pages 3–10, 2006.

41 A. Zerzelidis and A. J. Wellings. A framework for flexible scheduling in the rtsj. *ACM Trans. Embed. Comput. Syst.*, 10(1):3:1–3:44, August 2010. doi:10.1145/1814539.1814542.

Applying Real-Time Scheduling Theory to the Synchronous Data Flow Model of Computation*

Abhishek Singh[1], Pontus Ekberg[2], and Sanjoy K. Baruah[3]

1 **The University of North Carolina, Chapel Hill, NC, USA**
abh@cs.unc.edu
2 **Uppsala University, Uppsala, Sweden**
pontus.ekberg@it.uu.se
3 **The University of North Carolina, Chapel Hill, NC, USA**
baruah@cs.unc.edu

Abstract

Schedulability analysis techniques that are well understood within the real-time scheduling community are applied to the analysis of recurrent real-time workloads that are modeled using the synchronous data-flow graph (SDFG) model. An enhancement to the standard SDFG model is proposed, that permits the specification of a real-time latency constraint between a specified input and a specified output of an SDFG. A technique is derived for transforming such an enhanced SDFG to a collection of traditional 3-parameter sporadic tasks, thereby allowing for the analysis of systems of SDFG tasks using the methods and algorithms that have previously been developed within the real-time scheduling community for the analysis of systems of such sporadic tasks. The applicability of this approach is illustrated by applying prior results from real-time scheduling theory to construct an exact preemptive uniprocessor schedulability test for collections of recurrent processes that are each represented using the enhanced SDFG model.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems, C.3 Signal Processing Systems

Keywords and phrases real-time systems, synchronous dataflow (SDF), hard real-time streaming dataflow applications, algorithms

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.8

1 Introduction

The research discussed in this document is inspired in part by problems that arise in designing base stations for wireless cellular communication systems. A base-station can handle a certain number of connections; for each such handled connection, streams of data packets arrive at the base-station and go through a number of stages of data-flow processing. (The precise nature of such processing depends upon the kind of connection, and may, therefore, be different for different connections.) A minimum duration is assumed between the arrival of successive data packets of the same connection at the base station, and the processing of the packet is required to complete within a specified duration of its arrival.

It is quite natural to model such processing requirements using *sporadic task models* of the kind that have been very widely studied in the real-time scheduling literature (see [26, 24] for

* This research is supported by NSF grants CNS 1115284, CNS 1218693, CNS 1409175, and CPS 1446631, AFOSR grant FA9550-14-1-0161, and ARO grant W911NF-14-1-0499. It was conducted while the second author was visiting the University of North Carolina.

a survey), in which the minimum inter-arrival duration of successive data packets is modeled by the 'period' parameter, and the duration allowed for the processing of each packet by the 'relative deadline' parameter, of sporadic tasks. However, the modeling of the actual processing of the packets, which, for telecom applications, are typically represented using the Synchronous Data Flow Graph (SDFG) [15] modeling abstraction[1], proves more challenging: there does not appear to be a straight-forward means of directly modeling such processing requirements using the concepts and terminology of real-time scheduling theory. Although the SDFG abstraction, which is widely used in the modeling, design, and analysis of real-time streaming applications in telecommunications and other domains, has been studied for decades, the run-time scheduling of computational workloads that are represented in the SDFG model has traditionally been done via static scheduling methods (e.g., [13, 14]; see [23] for a text-book description), in which scheduling tables are determined prior to run-time and these pre-computed tables are used for making run-time scheduling decisions. Techniques for constructing such tables have been developed from first principles, and these techniques display little commonality with the ones that are used in the real-time scheduling theory community. As telecommunications and other streaming applications become increasingly more computation-intensive and efficiency of implementation becomes an increasing concern, however, efforts are being made to apply the more efficient dynamic scheduling approaches, of the kind that is very familiar to the real-time scheduling community, to the scheduling of such systems represented using the SDFG model. Examples of such efforts include the following (this is not an exhaustive list):

1. Bouakaz et al. [5] apply EDF scheduling in order to implement multiple independent applications each specified in the SDFG model upon a shared (uniprocessor or partitioned multiprocessor) computing platform.
2. Bamakhrama and Stefanov [2, 3] propose techniques for transforming SDFG specifications of a particular kind (called *acyclic cyclo-static* data flow graphs) to collections of periodic tasks, which can then be scheduled using the methods and algorithms developed in real-time scheduling theory for periodic task scheduling.
3. Ali et al. [1] have developed techniques for transforming SDFG specifications of a different kind, called *cyclic homogeneous* SDFG, to collections of periodic tasks.
4. Mohaqeqi et al. [18] describe how to represent SDFG specifications in the *digraph* real-time (DRT) task model [25].

To our knowledge, these approaches are all proved correct but none claim optimality; indeed, it is fairly easy to construct example instances in which each such approach will result in implementations that make very inefficient use of platform computing resources. Other data-flow approaches that have been explored in the real-time systems community, such as the Processing Graph Method (PGM) [20] similarly suffer from an absence of optimality results. Some other recent research to have explored the relationship between SDF and real-time scheduling include [11, 12], which describe how periodic tasks with inter-task dependencies may be modeled using SDFGs (and thereby scheduled using approaches that have been developed for the scheduling of SDFGs).

Motivation

The long-term objective of our research is to investigate the applicability of the concepts, techniques, methodologies, and results of real-time scheduling theory to the analysis of real-

[1] The SDFG model is described in Section 2.

time workloads that are represented using the SDFG model. We believe that there are plenty of opportunities here: while real-time scheduling theory has made tremendous progress in recent years, this progress has, by and large, remained focused upon the workload models popular within the community. Meanwhile, data-flow models such as SDFG are finding increasing use in many embedded application domains, but research on these models have thus far primarily concentrated on ensuring correctness of design rather than enhancing efficiency of implementation. It is only now, with embedded software becoming increasingly computationally demanding, that obtaining efficient implementations of such software that are often specified using the SDFG model is becoming a primary consideration on par with correctness; this offers us in the real-time scheduling theory community an opportunity to demonstrate the usefulness and applicability of our research endeavors, simultaneously providing us with a rich source of interesting new problems that are of immediate interest outside the real-time scheduling theory community.

This research

In this paper, we report on our efforts at developing algorithms that allow us to transform, for the purposes of analysis, recurrent tasks that are specified using the SDFG model into a task model that is widely studied in the real-time scheduling theory literature: the 3-parameter sporadic task model [19, 4]. We do so by

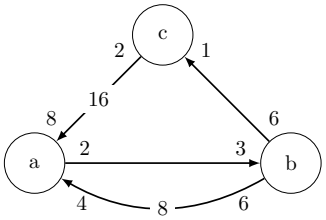
1. detailing extensions to the basic SDFG model that render it suitable for representing real-time requirements; and then
2. deriving an algorithm that allows us to transform any task that is represented using this SDFG model into a collection of 3-parameter sporadic tasks that have worst-case computational requirement (as represented using the *demand bound function abstraction* [4]) exactly equal to the worst-case computational requirement of the SDFG task.

In this manner, determining schedulability for a collection of independent SDFG tasks implemented upon a shared platform is reduced to the problem of determining schedulability of a collection of 3-parameter sporadic tasks implemented upon a shared platform – this is a problem that is well understood in real-time scheduling theory.[2] In particular, this means that we can schedule a collection of independent SDFG tasks *optimally* upon a preemptive uniprocessor platform, by exploiting the well-known result concerning the optimality of the earliest deadline first scheduling algorithm (EDF) upon preemptive uniprocessors [17, 7].

Organization

The remainder of this paper is organized as follows. In Section 2 we describe the SDFG model of computation and detail some enhancements to the basic model that make it better suited for representing real-time constraints. In Section 3 we briefly discuss the 3-parameter sporadic task model, and present without proof some results concerning analysis of systems represented using this model. In Section 4 we present, prove correct, and characterize the effectiveness of, our algorithm for transforming any SDFG task to a collection of equivalent 3-parameter sporadic tasks.

[2] As an added benefit, this transformation allows us to schedule collections of tasks, some of which are represented using the SDFG model and others, using traditional real-time task models, upon a shared platform and analyzed within a common framework.



$$\Gamma = \begin{array}{c} \\ (a,b) \\ (b,a) \\ (b,c) \\ (c,a) \end{array} \begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{pmatrix} 2 & -3 & 0 \\ -4 & 6 & 0 \\ 0 & 6 & -1 \\ -8 & 0 & 2 \end{pmatrix} \end{array} \qquad \mathrm{q} = \begin{array}{c} \\ a \\ b \\ c \end{array} \begin{pmatrix} 3 \\ 2 \\ 12 \end{pmatrix}$$

Figure 1 An example SDFG. From left to right: the SDFG, its topology matrix Γ, and its repetition vector q. (Rows and columns of the topology matrix are labeled by channel and actor respectively, and rows of the repetition vector are labeled by actor.)

A note on the presentation. This paper is aimed at a readership that is familiar with the real-time scheduling literature, but not necessarily the literature on synchronous and other data-flow models. We have therefore chosen to provide a rather detailed explanation of the SDFG model, and have made certain simplifications by essentially ignoring aspects of the model that are orthogonal to our perspective of scheduling to provide real-time guarantees. In contrast, we have been terse with our discussion of the 3-parameter sporadic task model, assuming that the reader is already very familiar with this model.

2 A real-time SDF model

In this section we describe both the basic SDF model [13, 15], and several extensions that have been proposed to the model in order to enhance its capabilities to more accurately depict real-time considerations.

2.1 Synchronous Data Flow Graphs

We now provide a brief introduction to the synchronous data-flow graph model of computation; we refer the interested reader to [16, Ch 6.3.2] for a text-book description and additional references.

A dataflow graph is a directed graph[3] in which the vertices (known as *actors*) represent computation and edges (known as *channels*) represent FIFO queues that direct data (called *tokens*) from the output of one computation to the input of another. Actors *consume* tokens from their input channels, perform computations upon them (this is referred to as a *firing* of the actor) and *produce* tokens on their output channels. Channels may contain *initial tokens* (also known as *delays*) – these represent data that populate the FIFO queues prior to run-time. In a *synchronous dataflow graph (SDFG)*, the number of initial tokens on each channel, as well as the number of tokens produced (consumed, respectively) by each actor on each of its outgoing (incoming, resp.) channels upon a firing of the actor, is a known constant.

▶ **Definition 1** (SDFG). An SDFG G is represented as a 5-tuple $G = (V, E, \mathsf{prod}, \mathsf{cons}, \mathsf{delay})$ where

[3] Most SDFG models allow for *multi*graphs, in which there may be multiple edges between the same pair of vertices. This feature is not particularly relevant to determining how they are scheduled and are we, therefore, ignore them in this paper. For the same reason, we also ignore edges that are self-loops: lead from a vertex back to itself. We point out that our results are easily extended to deal with multiple edges and self-loops.

- V denotes the set of actors.
- $E \subseteq (V \times V)$ is the set of channels. For each channel $e = (u, v)$, we denote u as $\mathsf{tail}(e)$ and v as $\mathsf{head}(e)$. We assume that $u \neq v$; i.e., there are no channels leading from an actor back to itself.
- $\mathsf{prod} : E \to \mathbb{N}_{>0}$. For each $e \in E$, $\mathsf{prod}(e)$ tokens are added to channel e each time the actor $\mathsf{tail}(e)$ fires.
- $\mathsf{cons} : E \to \mathbb{N}_{>0}$. For each $e \in E$, $\mathsf{cons}(e)$ tokens are removed from channel e each time the actor $\mathsf{head}(e)$ fires.
- $\mathsf{delay} : E \to \mathbb{N}_{\geq 0}$. For each $e \in E$, $\mathsf{delay}(e)$ denotes the number of initial tokens (or delays) on channel e.

We shall use the SDFG depicted in Figure 1 as our running example. It has three actors a, b and c, denoted by circles containing the actor name. Edges represent channels and are annotated at their ends with production and consumption rates and at their centers with the number of delays if the number is > 0.

Some additional terminology:

- The channels leading into an actor v from other actors are called *input channels* of v and are collectively denoted as $\mathrm{In}(v)$. *Output channels* of v are defined similarly and denoted as $\mathrm{Out}(v)$.
- If each channel $e \in \mathrm{In}(v)$ contains at least $\mathsf{cons}(e)$ tokens, then actor v is said to be *enabled*.
- An enabled actor may *fire*, i.e., execute. The firing of actor v removes $\mathsf{cons}(e)$ tokens from each $e \in \mathrm{In}(v)$, and adds $\mathsf{prod}(e)$ tokens to each $e \in \mathrm{Out}(v)$.

According to the description above, the initial configuration of tokens on channels (as represented by the delay function) determines all the future firings of actors; external events play no role in the SDFG's behavior. Lee and Messerschmitt [15] state this explicitly: 'Connections to the outside world are not considered [...] a node with only inputs from the outside is considered a node with no inputs, which can be scheduled at any time.' (Equivalently, it may be assumed that external input is always available when needed by an actor in order for it to fire.) While this assumption may have been reasonable for the original intended use of this model of computation for representing streaming computations, it is inconsistent with our efforts to incorporate real-time considerations; in Section 2.2 below, we discuss how we extend the SDFG model to incorporate timing properties of externally-provided data, which we model as external input tokens.

SDFG analysis techniques and algorithms have been developed [13, 15] for determining, for a given SDFG, whether sequences of firings of actors could lead to *deadlock* – a configuration of tokens on channels such that no actor is enabled, or to *buffer overflow* – the number of tokens in a channel growing without bound. We briefly summarize some of these results below.

▶ **Definition 2** (Topology Matrix)**.** For an SDFG $G = (V, E, \mathsf{prod}, \mathsf{cons}, \mathsf{delay})$, its *topology matrix*, denoted $\Gamma(G)$, is an $|E| \times |V|$ matrix in which the entry in the i'th row, j'th column, is as follows:

$$\Gamma(G)[i,j] \stackrel{\text{def}}{=} \begin{cases} \mathsf{prod}(e_i), & \text{if } \mathsf{tail}(e_i) = v_j \\ -\mathsf{cons}(e_i), & \text{if } \mathsf{head}(e_i) = v_j \\ 0, & \text{otherwise} \end{cases}$$

The topology matrix for the SDFG depicted in Figure 1 is shown in the figure. Consider, for instance, its first row corresponding to the channel leading from actor a to actor b. The entry $\Gamma(G)[1,1] = 2$ denotes that each firing of actor a (represented by the first column) adds (produces) two tokens to this channel; the entry $\Gamma(G)[1,2] = -3$ denotes that each firing of actor b (represented by the second column) removes (consumes) three tokens from this channel.

The following results are from [13, 15]:

1. There are methods to determine whether a given SDFG G is deadlock-free.
2. A connected[4] An SDFG G, with n actors, that is deadlock-free will not suffer from buffer overflow if and only if the rank[5] of its topology matrix $\Gamma(G)$ equals $(n-1)$.
   In the remainder of this paper we will assume that the SDFGs we deal with have been *a priori* verified to possess the properties of being deadlock-free and not subject to buffer overflow.
3. For such an SDFG G, we can efficiently find a positive integer vector $\vec{v}$ such that

$$\Gamma(G) \cdot \vec{v} = \mathbf{0}. \tag{1}$$

   (Since the topology matrix $\Gamma(G)$ has n columns it is evident that for Equation 1 to be well-formed, any such $\vec{v}$ must comprise n components – recall that n denotes the number of actors.)

If we interpret the n components of $\vec{v}$ as number of firings of the n actors, the reader may verify that satisfying Equation 1 is equivalent to asserting that upon completing the number of firings of each actor represented in $\vec{v}$, the total number of tokens in each channel is unchanged. This observation motivates the following definitions:

▶ **Definition 3** (Repetitions vector; Iteration)**.** The *repetitions vector* for an SDFG G is the smallest positive integer vector $\vec{v}$ for which Equation 1 holds, and is denoted by $\mathrm{q}(G)$. An *iteration* is a set of actor firings with as many firings as the repetitions vector entry for each actor.

Hence the number of tokens in each channel remains unchanged upon completion of an iteration, during which each actor would fire as many times as indicated by its corresponding entry in the repetitions vector. This is formally stated in the following *balance equation* for each channel $e \in E$:

$$\mathsf{prod}(e) \cdot \mathrm{q}(G)[\mathsf{tail}(e)] = \mathsf{cons}(e) \cdot \mathrm{q}(G)[\mathsf{head}(e)] \tag{2}$$

In the remainder of the text when the SDFG G under consideration is evident, we will often ***simplify our notation and write*** Γ ***(***q***, respectively) for*** $\Gamma(G)$ ***(***$\mathrm{q}(G)$***, resp.).***

In addition to the topology matrix, the repetitions vector for the SDFG depicted in Figure 1 is also shown in the figure. For this example, it is easily verified that Equations 1 and 2 do indeed hold:

$$\Gamma \cdot \mathrm{q} = \begin{pmatrix} 2 & -3 & 0 \\ -4 & 6 & 0 \\ 0 & 6 & -1 \\ -8 & 0 & 2 \end{pmatrix} \cdot \begin{pmatrix} 3 \\ 2 \\ 12 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

[4] An SDFG $G = (V, E, \mathsf{prod}, \mathsf{cons}, \mathsf{delay})$ is said to be *connected* if its set of actors and set of channels, when interpreted as the vertices and edges of a directed graph, represents a weakly connected digraph.

[5] The *rank* of a matrix is the maximum number of linearly independent columns in it. Efficient polynomial-time algorithms are known for computing rank.

An iteration of this example SDFG therefore comprises three firings (i.e., executions) of actor a, two of actor b, and twelve of actor c.

We close this section out defining a restricted kind of SDFG.

▶ **Definition 4** (Homogeneous SDFG (HSDFG))**.** An *homogeneous* SDFG is one in which all the prod and cons rates are equal to one.

For such homogeneous SDFG's, it follows from the balance equation (Equation 2 above) that the repetitions vector, if one exists (i.e., if the SDFG is deadlock-free and is not subject to buffer overflow) will comprise all ones: $\mathrm{q}(a) = 1$ for all $a \in V$.

Bounding buffer sizes

A significant amount of the SDFG research literature deals with the problem of minimizing the size of the channel buffers; i.e., minimizing the maximum number of tokens that need to be stored in individual channels. While this is an important dimension to SDFGs, we have chosen to ignore this aspect of the problem in the current paper, leaving its consideration as future work. (We have considered several ad hoc approaches for dealing with buffer capacity constraints on channels by, e.g., associating a deadline (temporal) with the actor at the head of the channel, but these ad hoc solutions have not yet been proved optimal.)

2.2 Incorporating real time

We now discuss extensions that have been made to the basic SDFG model described above over the years in order to incorporate real-time considerations.

Actor execution times

The SDFG model, as described in Section 2.1 above, does not incorporate consideration of real time; it was subsequently extended [10] to additionally associate an execution time with each actor. That is, along with the parameters V, E, prod, cons, and delay as defined in Definition 1, we specify an additional parameter, a *worst-case execution time* function

$$\mathsf{wcet} : V \to \mathbb{N}_{\geq 0}$$

with the interpretation that for each $v \in V$, $\mathsf{wcet}(v)$ is the worst-case execution time of a (single) firing of actor v.

Input-output Latency

As we had stated earlier, the SDFG model as originally proposed did not explicitly model the interaction of the SDFG with the external world: 'Connections to the outside world are not considered' [15]. But our interest is expressly in this interaction: as stated in the introduction, the kinds of applications we are analyzing typically require that the processing of a packet complete within a specified (real-time) duration of its arrival. The SDFG model may be extended as follows in order to incorporate such real-time considerations. For each SDFG, we additionally specify

- a single *input actor* v_{in} and a single *output actor* v_{out}.[6] External tokens are assumed to arrive at the input actor v_{in}. That is, we can imagine an additional channel e_{in}

[6] The rationale behind the decision to restrict the number of input and output actors to being one is explained in Section 2.3.

with $\mathsf{head}(e_{\text{in}}) = v_{\text{in}}$ and $\mathsf{tail}(e_{\text{in}})$ not specified, but rather representing the external environment within which the SDFG is operating. The consume rate $\mathsf{cons}(e_{\text{in}})$ is one, while no produce rate $\mathsf{prod}(e_{\text{in}})$ is specified; instead, tokens 'appear' on channel e_{in} according to the period parameter (discussed next).

- a *period* parameter, denoting the minimum duration between successive arrivals of external tokens on the input channel e_{in}, and
- a *relative deadline* parameter, denoting the maximum duration that may elapse between the arrival of an external token on the input channel e_{in} and the completion of the 'corresponding' execution of the output actor v_{out} (this notion of correspondence is elaborated upon below – see Definition 5).

In the example SDFG of Figure 1, we could, for instance, specify that actor a is the input actor, actor b the output actor, the period is 5 time units and the relative deadline is 2 time units. This would mean that

- Actor a may only fire when there are at least 8 tokens on its input channel (c, a) and at least 4 tokens on its input channel (b, a), *and* at least one external input token; its firing consumes 8 tokens from (c, a), 4 tokens from (b, a), and one external input token.
- The duration between the arrival of successive external input tokens at a is no smaller than 5 time units.
- The maximum duration that may elapse between the arrival of the external input token at a and the completion of the firing of a corresponding execution of b is 2 time units.

Let us now formalize this notion of correspondence.

Suppose that the first external input token arrives at actor a at some time instant, thereby causing actor a to fire. Observe that since the production rate $\mathsf{prod}(a, b)$ of the channel leading from a to b is two while the consumption rate $\mathsf{cons}(a, b)$ is three, at least two firings of actor a must occur before actor b may fire for the first time. But since the period of the SDFG denotes only a *lower* bound on duration between the arrival of successive external input tokens, we cannot provide an upper bound upon the time instant at which actor b is enabled – this depends upon when the second external input token arrives at actor a. It is therefore not particularly meaningful to discuss the latency of the response to the first external input token since the response will be triggered by not the first, but the second external input token.

This conundrum was resolved by Ghamarian et al. [10] based on the reasoning that an entire iteration (see Definition 3) of an SDFG should be thought of as representing a single logical chunk of computation. Therefore it is not meaningful to consider the arrivals of external input tokens at the input actor, and firings of the output actor, within an iteration; instead, we should only consider the delay between the arrival of an external input token that initiates the first firing of the input actor within an iteration, and the completion of the execution of the last firing of the output actor during that iteration. We do so by preprocessing an SDFG to make the following changes (Figure 2 illustrates the result of applying these changes to the example of Figure 1):

- Add two new actors src and dst, with $\mathsf{wcet}(\mathsf{src}) = \mathsf{wcet}(\mathsf{dst}) = 0$, and ensure (see below) that each will execute exactly once per iteration. These now become the designated input and output actors, while v_{in} and v_{out} are just 'regular' actors.
- Add two new channels: $e_1 = (\mathsf{src}, v_{\text{in}})$, $e_2 = (v_{\text{out}}, \mathsf{dst})$, with
  - $\mathsf{delay}(e_1) = \mathsf{delay}(e_2) = 0$,
  - $\mathsf{prod}(e_1) = \mathrm{q}(v_{\text{in}})$, $\mathsf{cons}(e_1) = 1$, (recall that each actor a executes $\mathrm{q}(a)$ times per iteration), and
  - $\mathsf{prod}(e_2) = 1$, $\mathsf{cons}(e_2) = \mathrm{q}(v_{\text{out}})$.

  These assignments of $\mathsf{delay}, \mathsf{prod}$, and cons values to e_1 and e_2 ensure that src and dst both execute exactly once per iteration (i.e., $\mathrm{q}(\mathsf{src}) = \mathrm{q}(\mathsf{dst}) = 1$).

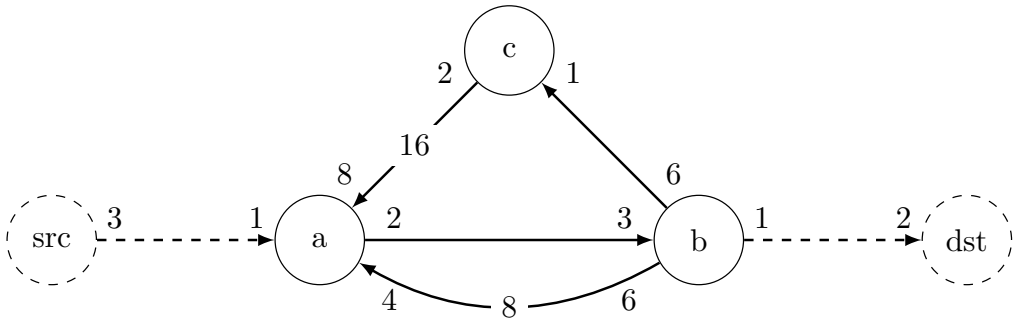


Figure 2 Applying the transformations of Section 2.2 to the example SDFG shown in Figure 1. Suppose that actors a and b are identified as the input and output actors of the example SDFG of Figure 1. The (dashed) actors and channels designated src and dst are added, and become the designated input and output actors. Recall from Figure 1 that $\mathrm{q}(a) = 3$ and $\mathrm{q}(b) = 2$; hence the production and consumption rates assigned to the channel connecting src to a are 3 and 1 respectively, while the production and consumption rates assigned to the channel connecting b to dst are 1 and 2 respectively. This SDFG is further characterized with a relative deadline and a period parameter, both being positive integers.

After the transformation, the arrival of $\mathrm{q}(v_{\text{in}})$ external input tokens at actor v_{in} in the original SDFG is modeled as the arrival of one external input token at src.[7] If the period parameter of the original SDFG had represented the minimum inter-arrival duration of external input tokens at v_{in}, the period parameter of the transformed SDFG should be set equal to this original period multiplied by $\mathrm{q}(v_{\text{in}})$; the relative deadline parameter may also need to be modified suitably. (Indeed, the interpretation of the relative deadline parameter is ambiguous when input and output actors may fire multiple times per iteration; we, therefore, assume that the value is actually assigned to this parameter *after* the modifications outlined above have been carried out.)

Henceforth, we will assume that our ***SDFGs have been pre-processed in this manner***, and that as a consequence, we have SDFGs with designated input and output actors that are guaranteed to execute exactly once per iteration. We will also assume that each actor is 'reachable' via channels from src, and that dst is reachable from each actor; i.e., each actor is involved in processing and relaying data from the input to the output. (Actors not reachable in this manner will not impact the real-time properties of the SDFG, and may be removed from consideration during pre-processing to be executed in the background during run-time.)

An additional point of interest arises from the tokens that populate each channel initially, before the first arrival of an external input token at src. There are $\mathsf{delay}(e)$ such tokens on each $e \in E$; since each $\mathsf{delay}(e)$ is finite and since we require that each actor be reachable from src, any actor can be fired at most a finite number of times prior to src firing for the first time. The *dependency distance* denotes the maximum number of times dst can be fired before exhausting the initially-supplied tokens:

[7] One may choose to think of src as a dummy actor that queues the external input tokens directed at v_{in} until it has accumulated $\mathrm{q}(v_{\text{in}})$ tokens, at which instant it releases them all simultaneously to a; hence, a does not have to deal with the possibility of unbounded durations between the arrivals of the three tokens. (However, an unbounded duration may elapse before the *next* set of three tokens are released to it.) An analogous interpretation may be made for dst.

▶ **Definition 5** (Dependency Distance δ [22]; Correspondence)**.** Due to the initial distribution of tokens on the channels specified by delay, dst can fire some δ times before src fires for the first time. The number δ is called the dependency distance.

For any $k \in \mathbb{N}$, the k-th firing of src is said to *correspond to* the $(k+\delta)$-th firing of dst, where δ is the dependency distance.

Suppose that in our example SDFG of Figure 1, appropriately pre-processed to take the form depicted in Figure 2, $\mathsf{delay}(a,b)$ were equal to 10 rather than zero (i.e, 10 tokens were initially provided in this channel). Since $\mathsf{cons}(a,b) = 3$, it is evident that actor b may fire a total of three times prior to the arrival of any external input tokens at src, thereby placing three tokens on the channel connecting actor b to actor dst. Since actor dst needs two tokens on this channel to fire, it may fire once prior to the first arrival of any external input tokens at src; the dependency distance for this SDFG is therefore 1, and for all $k \in \mathbb{N}$ the k'th firing of the input actor src corresponds to the $(k+1)$'th firing of the output actor dst.

There are a variety of semantic reasons as to why channels of an SDFG may be populated with initial tokens. From the perspective of minimizing the amount of execution that must be performed in response to the arrival of an external input token, it is a good strategy to perform as much '*pre-computation*' on the SDFG as possible, and fire as many actors as one can prior to the arrival of the first external input token. (Continuing the example above of having 10 initial tokens on channel (a,b), we could fire actor b thrice beforehand, thus placing 3 tokens on channel (b, dst), $(3 \times 6 =)$ 18 tokens on channel (b,c), and $(3 \times 6 + 8 =)$ 26 tokens on channel (b,a). The tokens on channel (b, dst) would allow actor dst to fire once, while the tokens on channel (b,c) would allow actor c to fire 18 times, placing $(18 \times 2 + 16) = 52$ tokens on channel (c,a). The final state of the channels is then

$$\mathsf{delay}(a,b) = 1; \quad \mathsf{delay}(b,\mathsf{dst}) = 1; \quad \mathsf{delay}(b,a) = 26;$$
$$\mathsf{delay}(b,c) = 0; \quad \mathsf{delay}(c,a) = 52; \quad \mathsf{delay}(\mathsf{src},a) = 0.$$

In order to keep things simple, in the remainder of this paper we will assume that all enabled actors are repeatedly fired prior to run-time, so that *there are no enabled actors prior to the arrival of the first external input token*. This immediately implies that the dependency distance $\delta = 0$: the k'th firing of the input actor corresponds to the k'th firing of the output actor for all $k \in \mathbb{N}$.

2.3 Summary of, and rationale for, the sporadic real-time SDFG model

We will refer to the recurrent task model obtained by making all the enhancements discussed in Section 2.2 above to the 'traditional' SDFG model as the ***sporadic real-time SDFG*** model. A task in this model is specified as follows:

$$G \stackrel{\text{def}}{=} \Big\langle (V,\, E,\, \mathsf{prod},\, \mathsf{cons},\, \mathsf{delay}), \mathsf{wcet}, \mathsf{src}, \mathsf{dst}, D, T \Big\rangle \tag{3}$$

with

- V, E, prod, cons, and delay as specified for traditional SDFGs;
- $\mathsf{wcet} : V \to \mathbb{N}_{\geq 0}$ specifying the worst-case execution times of the actors;
- Actors $\mathsf{src} \in V$ and $\mathsf{dst} \in V$ being specified as the unique input and output actor, respectively; and
- $D \in \mathbb{N}$ and $T \in \mathbb{N}$ specifying the relative deadline and period parameters of this sporadic real-time SDFG task.

Additionally, we assume that the SDFG has been validated to be deadlock-free and free from buffer overflow, and to have the repetition rates for the input and output actors equal to one: $\mathrm{q}(\mathsf{src}) = \mathrm{q}(\mathsf{dst}) = 1$.

We now briefly discuss the rationale behind some of the design decisions we have made in the specification of the sporadic real-time SDFG task model.

1. **A single input actor.** Tokens are assumed to arrive at an input actor in a sporadic manner, with a minimum inter-arrival duration, but no maximum inter-arrival duration, specified. Latency or response time is measured from the instant that such an input token arrives, to the instant that the corresponding firing of the output actor completes. The following simple example illustrates the problem with allowing multiple independent input actors.

   Suppose that there are two input actors a and b; external tokens arrive sporadically at each. Suppose that there are channels (a, c) and (b, c) leading from a and b to a third actor c, and both a and b must complete firing in order for c to fire. After an external token arrives at a, there is no upper bound on the duration of time before an external token arrives at b; hence, we cannot bound the duration of time between the arrival of the input token at a and the firing of c.

   It is, of course, possible to have the *same* sporadic input stream of tokens arrive at multiple actors, but this is effectively modeled by having a single dummy input actor (with $\mathsf{wcet} = 0$) from which channels lead out to all the original input actors receiving this stream of tokens.

2. **The input (and output) actors execute once per repetition** ($\mathrm{q}(\mathsf{src}) = \mathrm{q}(\mathsf{dst}) = 1$). This was discussed above, when introducing the transformation of adding the single source actor src: since we cannot bound the duration between the arrival of successive external tokens from above, the concept of latency is not meaningful except in considering arrivals of a group of external input tokens for an entire iteration of the SDFG. This concept is abstracted into the new input actor src that is added, and guaranteed to have $\mathrm{q}(\mathsf{src}) = 1$.

3. **A single output actor.** This is not a necessary restriction – it is quite possible to specify multiple output actors, with different latencies ('relative deadlines') specified for each. (Of course, each output actor so specified must satisfy the property that it executes exactly once per iteration: $\mathrm{q}(v) = 1$ for each such output actor v.) In this paper we restrict consideration to a single output actor per task in order to keep things simple; our results are easily extended to deal with multiple output actors.

4. **Each actor is reachable from** src**, and** dst **is reachable from each actor.** It is easily seen that any actor that is either not reachable from src, or from which dst is not reachable, need not fire at all in order to ensure that dst fires in response to a firing of src. Hence, we need not execute such actors during run-time to ensure real-time correctness, and their presence has no impact on schedulability. (In practice, such actors may be executed in the background when there are no real-time actors awaiting execution.)

5. **No actors are enabled before the first external input arrives.** (And as a result, $\delta = 0$.) As we had argued above, performing pre-processing prior to run-time by maximally firing all enabled actors is a reasonable strategy from the perspective of minimizing the run-time computational workload. We, therefore, assume this in the remainder of this paper. However, we point out that this is not necessary – our algorithms are easily extended to deal with the case where such pre-processing is not done for whatever reason, and $\delta > 0$.

3 The three-parameter sporadic task model

We now provide a very brief introduction to the 3-parameter sporadic task model [19], which is widely used in real-time scheduling theory. (This introduction is primarily intended to introduce terminology and notation; we assume that the reader is already very familiar with this model.)

A 3-parameter sporadic task $\tau_i = (C_i, D_i, T_i)$ is characterized by a WCET C_i, a relative deadline parameter D_i, and a period T_i. Such a task generates an unbounded sequence of jobs, with each job having an execution requirement $\leq C_i$ and successive arrivals at least T_i time units apart. Each job is required to complete by a deadline that is D_i time units after its arrival time.

The scheduling of systems of 3-parameter sporadic tasks upon preemptive uniprocessors by the earliest deadline first scheduling algorithm (EDF) has been extensively studied, and algorithms derived for determining whether a given task system is EDF-schedulable or not. These algorithms make use of the concept of the *demand bound function* [4]. For any sporadic task τ_i and any real number $t > 0$, the demand bound function $\mathsf{dbf}(\tau_i, t)$ is the largest cumulative execution requirement of all jobs that can be generated by τ_i to have both their arrival times and their deadlines within a contiguous interval of length t. It is evident that the cumulative execution requirement of jobs of τ_i over an interval $[t_o, t_o + t)$ is maximized if one job arrives at the start of the interval – i.e., at time instant t_o – and subsequent jobs arrive as rapidly as permitted – i.e., at instants $t_o + T_i$, $t_o + 2T_i$, $t_o + 3T_i, \ldots$ (this fact is formally proved in [4]). We, therefore, have [4]:

$$\mathsf{dbf}(\tau_i, t) \stackrel{\text{def}}{=} \max\left(0, \left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) \times C_i\right) .$$

A *load* parameter, based upon the dbf function, may be defined for any sporadic task system τ as follows:

$$\mathsf{load}(\tau) \stackrel{\text{def}}{=} \max_{t>0}\left(\frac{\sum_{\tau_i \in \tau} \mathsf{dbf}(\tau_i, t)}{t}\right) .$$

It has been shown [4] that a necessary and sufficient condition for 3-parameter sporadic task system τ to be EDF-schedulable on a unit-speed preemptive uniprocessor is that $\mathsf{load}(\tau) \leq 1$. Pseudo-polynomial algorithms are known [4, 21, 28] for computing $\mathsf{load}(\tau)$, for task systems τ possessing the additional property that the quantity $(\sum_{\tau_i \in \tau} C_i/T_i)$ is bounded from above by a constant < 1. Polynomial-time approximation schemes (PTAS's) have also been derived that are able to compute an approximation to $\mathsf{load}(\tau)$ in polynomial time, to any desired degree of accuracy [9].

4 Optimal uniprocessor scheduling of sporadic real-time SDFGs

In this section, we will develop an optimal algorithm, and an associated exact schedulability test, for scheduling a collection of independent sporadic real-time SDFGs upon a preemptive uniprocessor. The algorithm is *optimal* in the following sense: if any run-time scheduling algorithm can guarantee to schedule the collection to always meet all deadlines for all permissible arrival sequences of external input tokens, then our algorithm also guarantees to always meet all deadlines for all permissible arrival sequences of external input tokens.

Our run-time scheduling algorithm is EDF-based: individual firings of actors are assigned deadlines, and at each instant in time, the earliest-deadline enabled actor firing that has not

yet completed execution is executed. The manner in which deadlines are assigned to firings of actors is described later.

We start with a high-level overview of our schedulability test. As with 3-parameter sporadic tasks (Section 3), we will characterize the execution requirement of a sporadic real-time SDFG by a demand bound function (dbf): for any sporadic real-time SDFG G (characterized as in Expression 3) and any positive real number t, let $\mathsf{dbf}(G,t)$ denote the maximum cumulative execution requirement that could be generated by SDFG G over a contiguous interval of duration t. Let $k(G,t)$ denote the following function:

$$k(G,t) \stackrel{\text{def}}{=} \max\left(0, \left(\left\lfloor \frac{t-D}{T} \right\rfloor + 1\right)\right) . \tag{4}$$

(In the remainder of the text when the SDFG G under consideration is evident, we will ***simplify our notation and write*** $k(t)$ ***for*** $k(G,t)$.)

It is evident, using an argument analogous to those used in computing dbf for 3-parameter sporadic tasks, that over any contiguous time-interval of duration t there may be at most $k(t)$ external input tokens arriving at src for which the corresponding firings of dst must occur within the interval (this happens when the first external input token arrives at the start of the interval, and successive external input tokens arrive exactly T time units apart). Since each arrival of an external input token at src triggers one iteration (see Definition 3) of G, an upper bound for $\mathsf{dbf}(G,t)$ may be obtained by simply assuming that each actor a fires a total of $\mathrm{q}[a]$ times during each such iteration, thereby obtaining a bound of

$$k(t) \times \sum_{a \in V} \big(\mathrm{q}[a]\mathsf{wcet}(a)\big) . \tag{5}$$

However this bound, while safe, is not necessarily tight – the presence of initial tokens on some of the channels (as represented by the $\mathsf{delay}(c)$ values) means that not all firings of all actors need take place. Consider, for instance, the example of Figure 2 and consider a value of t_o satisfying $D < t_o < T + D$, so that $k(t_o)$ evaluates to 1 by Equation 4. Even though we had previously computed (see Figure 1) that $\mathrm{q}[c] = 12$, the reader may verify that firing actor c just four times suffices to ensure that dst is able to fire. Hence over such a t_o, $\mathsf{dbf}(G,t_o)$ equals

$$\Big(3\,\mathsf{wcet}(a) + 2\,\mathsf{wcet}(b) + \mathbf{4}\,\mathsf{wcet}(c)\Big), \text{ rather than } \Big(3\,\mathsf{wcet}(a) + 2\,\mathsf{wcet}(b) + \mathbf{12}\,\mathsf{wcet}(c)\Big)$$

as suggested by the upper bound in Expression 5 above.

In the remainder of this section, we will describe the computation of a *skip vector* $\mathrm{s}(G)$ of non-negative integers, with $|V|$ components, which will represent the maximum number of firings of each actor that we may 'skip' as a consequence of the presence of initial tokens on the channels.[8] That is, we will show that for each actor a the computed skip-vector value $\mathrm{s}(G)[a]$ is the largest integer possessing the property that actor a will need to complete no more than

$$\max\Big(0, \big(k(t) \times \mathrm{q}[a]\big) - \mathrm{s}(G)[a]\Big)$$

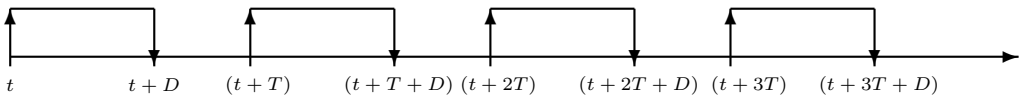
firings over any contiguous interval of duration t.

[8] We will show, in Lemma 6, that this skip vector is uniquely defined for a given G.

(As we did with the topology matrix Γ and the repetitions vector q, when the SDFG G under consideration is evident we will often ***simplify our notation and write*** s ***for*** s(G).)

Before deriving formulae for computing the skip vector, let us briefly illustrate how the skip-vector values, once computed, will be used during run-time for assigning deadlines to the firings of actors. Suppose that we have determined, for a particular actor a in given sporadic real-time SDFG G, that s[a] = 10 and q[a] = 3 (i.e., the actor fires three times per iteration, but a maximum of ten firings may be skipped). Suppose an external input token arrives at src at some time instant t, we would schedule two firings of actor a to complete by a deadline $t + 3T + D$, and a further firing of actor a to complete by a deadline $t + 4T + D$. (This is equivalent, for the purposes of developing a uniprocessor EDF schedulability test, to representing actor a's computational requirements by two 3-parameter sporadic tasks: one with parameters $(2\,\mathsf{wcet}(a), 3T + D, T)$, and one with parameters $(\mathsf{wcet}(a), 4T + D, T)$.)

Let us now seek to understand the rationale behind this strategy. If we were to not schedule any firings of actor a in response to the arrival of the external input token at time instant t, we would have 'used up' three of the ten skips that are permitted; over four iterations, all the skips would thus be completely used up, and future iterations would need to complete three firings by their deadlines. Recall that our objective is to minimize the run-time computational demand of the task, which, as quantified by dbf, is a *worst case* measure; under such a strategy, the dbf for the task is defined by these future iterations during which no firings may be skipped – applied to all the actors, this would be exactly equal to the upper bound of Expression 5. So instead we do schedule the three firings of a associated with the arrival of the external input token at time instant t, but rather than assigning them a deadline at $t + D$, we assign them later deadlines, thereby 'spreading out' their contribution to the dbf. As shown in the following figure, we know that the next three external input tokens cannot arrive before time-instants $t + T$, $t + 2T$, and $t + 3T$:



Since we are allowed to skip 10 firings of the actor, we may skip all three firings for the first three iterations of the SDFG, and one of the firings for the next (i.e., fourth) iteration of the SDFG; however, we cannot skip the other two firings for the fourth iteration, nor any for the fifth (and future) iterations. We, therefore, schedule two of the firings of a associated with the current iteration to complete by the deadline of the fourth iteration, and the third to complete by the deadline of the fifth iteration. As the figure above shows, the deadline of the fourth iteration is $\geq (t + 3T + D)$, while the deadline of the fifth iteration is $\geq (t + 4T + D)$; hence the decision to schedule two firings of actor a to complete by a deadline $t + 3T + D$, and a further firing of actor a to complete by a deadline $t + 4T + D$.

Generalizing the example above from q[a] ← 3 and s[a] ← 10 to arbitrary values for q[a] and s[a], it is straightforward to show that in response to an external input token's arrival at time-instant t, we would schedule each actor a to have

$$\big(\mathrm{q}[a] - (\mathrm{s}[a] \bmod \mathrm{q}[a])\big) \text{ firings with a deadline at } \Big(\big\lfloor \mathrm{s}[a]/\mathrm{q}[a] \big\rfloor \cdot T + D\Big)$$

and the remaining

$$\big(\mathrm{s}[a] \bmod \mathrm{q}[a]\big) \text{ firings with a deadline at } \Big(\big(\big\lfloor \mathrm{s}[a]/\mathrm{q}[a] \big\rfloor + 1\big) \cdot T + D\Big).$$

4.1 Computing the skip vector

Our intent is that the skip vector value $\mathrm{s}[a]$ denote the maximum number of times the execution of actor a may be skipped, due to the presence of initial tokens on the edges. Let us now consider any channel $e \in E$ of the sporadic real-time SDFG under consideration, and let $u = \mathsf{tail}(e)$, $v = \mathsf{head}(e)$. Let n_u and n_v denote the number of times that actors u and v have fired by some point in time; it must be the case that

$$n_u \cdot \mathsf{prod}(e) + \mathsf{delay}(e) \geq n_v \cdot \mathsf{cons}(e). \tag{6}$$

Let us instantiate Equation 6 above to the end of the k'th iteration of the sporadic real-time SDFG under consideration. At that point in time, $n_u \leftarrow (k \cdot \mathrm{q}[u] - \mathrm{s}[u])$ and $n_v \leftarrow (k \cdot \mathrm{q}[v] - \mathrm{s}[v])$; hence we have

$$\begin{aligned}
& \Big(k \cdot \mathrm{q}[u] - \mathrm{s}[u]\Big) \cdot \mathsf{prod}(e) + \mathsf{delay}(e) && \geq && \Big(k \cdot \mathrm{q}[v] - \mathrm{s}[v]\Big) \cdot \mathsf{cons}(e) \\
\Leftrightarrow\ & k \cdot \mathrm{q}[u] \cdot \mathsf{prod}(e) - \mathrm{s}[u] \cdot \mathsf{prod}(e) + \mathsf{delay}(e) && \geq && k \cdot \mathrm{q}[v] \cdot \mathsf{cons}(e) - \mathrm{s}[v] \cdot \mathsf{cons}(e) \\
\Leftrightarrow\ & k \cdot \underbrace{\Big(\mathrm{q}[u] \cdot \mathsf{prod}(e) - \mathrm{q}[v] \cdot \mathsf{cons}(e)\Big)}_{=0 \text{ by the balance equation (Eqn. 2)}} + \mathsf{delay}(e) && \geq && \mathrm{s}[u] \cdot \mathsf{prod}(e) - \mathrm{s}[v] \cdot \mathsf{cons}(e) \\
\Leftrightarrow\ & \mathsf{delay}(e) && \geq && \mathrm{s}[u] \cdot \mathsf{prod}(e) - \mathrm{s}[v] \cdot \mathsf{cons}(e)
\end{aligned}$$

We will use this relationship that we have just derived above (replacing u and v with $\mathsf{tail}(e)$ and $\mathsf{head}(e)$):

$$\mathrm{s}[\mathsf{tail}(e)] \cdot \mathsf{prod}(e) - \mathrm{s}[\mathsf{head}(e)] \cdot \mathsf{cons}(e) \leq \mathsf{delay}(e)$$

$$\Leftrightarrow \quad \mathrm{s}[\mathsf{tail}(e)] \leq \left\lfloor \frac{\mathsf{delay}(e) + \mathrm{s}[\mathsf{head}(e)] \cdot \mathsf{cons}(e)}{\mathsf{prod}(e)} \right\rfloor \tag{7}$$

to help us compute the skip vector: our objective is to determine the largest values for $\mathrm{s}[a]$ for all actors a, such that Equation 7 is satisfied across all channels of the SDFG. Before doing so, we prove in Lemma 6 below, that there cannot be multiple incomparable skip vectors for the same SDFG.

▶ **Lemma 6.** *The skip vector* s *is unique.*

Proof. We prove the lemma by contradiction. Assume for this purpose that there exists two different skip vectors s and s′ for which Equation 7 holds for all channels $e \in E$. Assume also that both s and s′ are maximal, so that Equation 7 would not hold for either of them if we increased some values of s or s′.

Now let s″ be defined so that $\mathrm{s}''[v] = \max(\mathrm{s}[v], \mathrm{s}'[v])$ for all $v \in V$. For any edge $e \in E$ we have

$$\begin{aligned}
\mathrm{s}''[\mathsf{tail}(e)] &= \max(\mathrm{s}[\mathsf{tail}(e)], \mathrm{s}'[\mathsf{tail}(e)]) \\
&\leq \max\left(\left\lfloor \frac{\mathsf{delay}(e) + \mathrm{s}[\mathsf{head}(e)] \cdot \mathsf{cons}(e)}{\mathsf{prod}(e)} \right\rfloor, \left\lfloor \frac{\mathsf{delay}(e) + \mathrm{s}'[\mathsf{head}(e)] \cdot \mathsf{cons}(e)}{\mathsf{prod}(e)} \right\rfloor\right) \\
&= \left\lfloor \frac{\mathsf{delay}(e) + \mathrm{s}''[\mathsf{head}(e)] \cdot \mathsf{cons}(e)}{\mathsf{prod}(e)} \right\rfloor,
\end{aligned}$$

but then Equation 7 holds for all edges $e \in E$ also when using skip vector s″. It follows that s and s′ can not both be maximal. ◀

We now derive our algorithm for determining this skip vector. We start defining some additional terminology and notation. For each actor a, let $\check{s}[a]$ denote an upper bound on the value of $s[a]$; we will refer to these upper bounds as *skip estimates*. Our algorithm for computing the skip vector values will initialize these skip estimates as follows:

$$\check{s}[a] \leftarrow \begin{cases} 0, & \text{if } a = \mathsf{dst} \\ \infty, & \text{otherwise} \end{cases} \tag{8}$$

It is evident that these initial values on $\check{s}$ are indeed upper bounds on the skip vector values: since all skip vector values are necessarily finite, ∞ is an upper bound on the actual skip-vector values, and recall from Section 2.3 that our model assumes that that the dependency distance between the input and output actors equal zero ($\delta = 0$).[9]

Relaxing (along) a channel

For a given assignment of $\check{s}$ values to all the actors, the process of *relaxing* a channel consists of identifying a channel e for which the current skip estimates violate Condition 7:

$$\check{s}[\mathsf{tail}(e)] > \left\lfloor \frac{\mathsf{delay}(e) + \check{s}[\mathsf{head}(e)] \cdot \mathsf{cons}(e)}{\mathsf{prod}(e)} \right\rfloor$$

and updating (by decreasing) the skip estimate of $\mathsf{tail}(e)$ in order to cause it to satisfy Condition 7:

$$\check{s}[\mathsf{tail}(e)] \leftarrow \left\lfloor \frac{\mathsf{delay}(e) + \check{s}[\mathsf{head}(e)] \cdot \mathsf{cons}(e)}{\mathsf{prod}(e)} \right\rfloor \tag{9}$$

If no channel can be relaxed, then the current assignment of values to $\check{s}$ is the desired skip vector. Our algorithm for computing the skip vector can thus be stated as follows:

> ***Procedure*** Compute Skip-Vectors. *Repeatedly relax channels until no further channel relaxations are possible.*

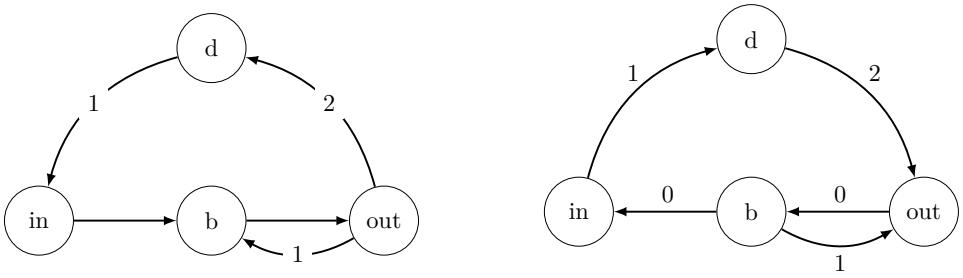
We elaborate upon the details of this algorithm first for the case of homogeneous SDFGs in Section 4.1.1 below; the case of general (i.e., not necessarily homogeneous) SDFGs is then considered in Section 4.1.2.

4.1.1 Homogeneous SDFGs

As stated in Definition 4, all produce and consume rates for homogeneous SDFGs are equal to one; for homogeneous SDFGs, Expression 9 above can therefore be simplified in the following manner:

$$\begin{aligned} &\check{s}[\mathsf{tail}(e)] \leftarrow \left\lfloor \frac{\mathsf{delay}(e) + \check{s}[\mathsf{head}(e)] \cdot \mathsf{cons}(e)}{\mathsf{prod}(e)} \right\rfloor \\ &\Leftrightarrow \check{s}[\mathsf{tail}(e)] \leftarrow \left\lfloor \frac{\mathsf{delay}(e) + \check{s}[\mathsf{head}(e)] \cdot 1}{1} \right\rfloor \\ &\Leftrightarrow \check{s}[\mathsf{tail}(e)] \leftarrow \Big(\mathsf{delay}(e) + \check{s}[\mathsf{head}(e)]\Big) \end{aligned} \tag{10}$$

[9] As we have stated in Section 2.3, in this paper we have assumed $\delta = 0$ in order to keep things simple. However, our algorithm is easily extended to handle non-zero dependency distances: we omit details here.



(a) Example homogeneous SDFG

(b) Shortest-path graph

Figure 3 Illustrating the computation of skip vectors for homogeneous SDFGs.

Observe that across any channel e, Condition 10 specifies that $\check{s}[\mathsf{tail}(e)]$ be no larger than $\check{s}[\mathsf{head}(e)]$ plus $\mathsf{delay}(e)$. This is similar to the constraints on single-source shortest-path problems on directed graphs (see, e.g., [6, Chapter 24] for a textbook discussion): if (u, v) is an edge of cost w in a directed graph, the shortest path from some designated vertex to v is of length no greater than the shortest path to u plus the cost w of the edge (u, v). This observation motivates our transformation of the skip vector computation problem to a single-source shortest path problem.

We will describe an algorithm for transforming the problem of assigning $\check{s}[v]$ values for all actors $v \in V$ in such a manner that no further channels relaxations are possible, to a single-source shortest paths problem in graphs. Let us attempt to obtain some intuition by working through the simple example HSDFG depicted in Figure 3 a. (In an HSDFG, since all repetition vector entries are always equal to one, the correspondence between the firing of input and output actors is well-defined and hence it is not necessary to add additional src and dst vertices.)

Let us consider the computation of the skip vector on the HSDFG of Figure 3 (a).

1. Upon initialization, the $\check{s}$ values are as follows:

$$\check{s}[\text{in}] = \check{s}[\text{b}] = \check{s}[\text{d}] = \infty;\ \check{s}[\text{out}] = 0$$

2. It is evident from visual inspection of Figure 3 (a) that the only channels that can be relaxed immediately after initialization are those for which the actor out is the head, i.e., the channel (b, out). Since $\mathsf{delay}(\text{b}, \text{out}) = 0$, such relaxation, which consists of applying Expression 10 to the channel $e = (\text{b}, \text{out})$, results in $\check{s}[\text{b}] \leftarrow 0$.
3. As a consequence, channels for which the actor b is the head potentially become relaxable. There are two such channels: (in, b) and (out, b). Relaxing the channel (in, b) updates $\check{s}[\text{in}]$ to 0, but it may be verified that channel (out, b) already satisfies Condition 7 and is hence not relaxable.
4. The update to $\check{s}[\text{in}]$ (in the step above) renders channels for which the actor in is the head potentially relaxable. This is the channel (d, in); since $\mathsf{delay}(\text{d}, \text{in}) = 1$, relaxing this channel updates $\check{s}[\text{d}]$ according to Expression 10 to a value of 1.
5. The update to $\check{s}[\text{d}]$ renders channels for which the actor d is the head potentially relaxable. This is the channel (out, d); it may be verified that this channel already satisfies Condition 7 and is hence not relaxable.

No further channels are relaxable, and hence the algorithm terminates with the following $\check{s}$ values:

$$\check{s}[\text{in}] = \check{s}[\text{out}] = \check{s}[\text{b}] = 0;\ \check{s}[\text{d}] = 1$$

(The reader may verify that these values are indeed correct, by observing that the presence of one initial token on the channel from actor d to actor in permits us to skip one firing of actor d, and that no further skips are possible.)[10]

We now describe the algorithm for transforming a homogeneous SDFG into a directed graph, such that solving a single-source shortest paths problem upon this graph will compute the skip vector for all the actors in that HSDFG.

1. The graph is constructed to have one vertex corresponding to each actor in the homogeneous SDFG.
   The graph for our example is depicted in Figure 3 (b); since the homogeneous SDFG of Figure 3 (a) has four actors this graph has four vertices, each labeled with the name of its corresponding actor.
2. For each channel c with $\mathsf{tail}(c) = u$ and $\mathsf{head}(c) = v$; we add an edge from the vertex corresponding to actor v to the vertex corresponding to actor u, and assign this edge a cost equal to $\mathsf{delay}(c)$.
   Observe that the edges of the graph depicted in Figure 3 (b) are *reversed* from the channel they correspond to, and that each is assigned a cost equal to the number of initial tokens (delays) on the channel.
3. As a consequence of the structural similarity of Condition 10 to shortest-path constraints, it follows by a direct application of shortest-paths arguments (see, e.g., [6, Sec 24.4]), that the skip vector value for each actor is equal to the shortest path in the graph to the vertex corresponding to it, from the vertex corresponding to the output vertex (out).
   In Figure 3 (b), it is evident that the shortest paths from the vertex labeled out to the vertices labelled in, out, b, and d are 0, 0, 0, and 1 respectively. Therefore, we conclude that the skip vector values are as follows: s[in] = 0; s[out] = 0; s[b] = 0; and s[d] = 1.

Observe, additionally, that since none of the $\mathsf{delay}(e)$ values are negative, the shortest-paths problem is easily solved using Dijkstra's shortest-path algorithm [8], for which implementations are known that have $O(|V| \log |V| + |E|)$ running time.

4.1.2 General SDFGs

We now turn our attention to general SDFGs; for such SDFGs, produce and consume rates are allowed to be arbitrary non-negative integers, and we cannot, therefore, simplify Expression 9 to a more tractable form (as we did for HSDFGs, in Expression 10). Hence, procedure Compute Skip-Vectors, which had been defined earlier as

> ***Procedure*** Compute Skip-Vectors. *Repeatedly relax channels until no further channel relaxations are possible.*

is run through in its entirety. In this section, we informally argue that this procedure concludes upon performing no more than exponentially many relaxations; since each relaxation takes constant time, this immediately yields an exponential-time upper bound on the running time of procedure Compute Skip-Vectors.

[10] This example also illustrates the advantage of the *pre-processing* we advocate, of maximally firing all actors prior to the first arrival of an external input token. In the example of Figure 3 (a), such pre-processing would fire actor d twice, yielding three tokens on channel (d, in) (and removing the two tokens from channel (out, d). Repeated relaxations on the resulting configuration would increase š[d] to 3, while keeping the other values unchanged. I.e., a further decrease in the run-time computational requirement is possible.

As we saw in our HSDFG example, initially the only channels that can be relaxed are those for which the dst actor is the head: the š values assigned to these actors are updated from ∞ to a value that is polynomially bounded in the values of the prod, cons, and delay parameters of the SDFG. Each such actor with a non-∞ š value, in turn, causes the š values of other actors to be reduced from ∞ to a value that is polynomially bounded in its value and the values of the prod, cons, and delay parameters. We state without proof the following facts:

1. For each actor u that is not dst, the first relaxation that changes $\check{s}[u]$ from ∞ assigns it a value that is polynomially bounded in the values of the prod, cons, and delay parameters of the SDFG, and the current š values of actors that were previously assigned values other than ∞; and
2. Each relaxation decreases $\check{s}[u]$ for some actor u by at least one.

Since the composition of polynomially many polynomial functions is an exponential, the first fact above implies that the maximum value that each $\check{s}[u]$ may have, other than ∞, is exponentially bounded. Hence a total of $(|V| - 1)$ relaxations – one per actor except for dst – reduces all the š values to be exponentially bounded. Henceforth, each relaxation reduces one of the š values by at least one. Since the sum of $(|V| - 1)$ values each of which is exponentially bounded is also exponentially bounded, it follows that the total number of relaxations is exponentially bounded in the values of the parameters characterizing the SDFG.

Experimental evaluation

Above, we are only able to provide an exponential-time upper bound on the running time of the algorithm for determining the skip vector for a general (i.e., not necessarily homogeneous) SDFG. We have implemented and tested our algorithm on randomly-generated SDFGs; these experiments indicate that the convergence occurs rather more rapidly than is implied by the exponential bound. Specifically, we used the SDFG random graph generator that is provided[11] as part of the SDF3 [27] tool-suite to generate 1000 deadlock-free and consistent, weakly-connected, cyclic SDFGs. This tool allows for rates and degrees of actors to be specified using minimum and maximum bounds, average value, and variance. The probability that initial tokens are added to a channel and the sum of the repetition vector can also be specified. We generated all these specifications randomly from uniform distributions; in all 1000 cases, convergence occurred upon performing no more than $|V| \times |E|$ relaxations.

We close this section with an example illustrating, at a high level, the execution of procedure Compute Skip-Vectors, upon the example SDFG of Figure 2.

1. Upon initialization, the skip estimate values are

$$\Big\langle \check{s}[\mathsf{src}] = \infty,\ \check{s}[a] = \infty,\ \check{s}[b] = \infty,\ \check{s}[c] = \infty,\ \check{s}[\mathsf{dst}] = 0 \Big\rangle.$$

2. The only channel that can be relaxed now is $(\mathrm{b}, \mathsf{dst})$; the skip estimate values are updated to

$$\Big\langle \check{s}[\mathsf{src}] = \infty,\ \check{s}[a] = \infty,\ \check{s}[b] = 0,\ \check{s}[c] = \infty,\ \check{s}[\mathsf{dst}] = 0 \Big\rangle.$$

[11] Available for download at `www.es.ele.tue.nl/sdf3/` (accessed January 2017)

3. Let us suppose that the channel (a, b) is relaxed next. The skip estimate values are updated to
$$\Big\langle \check{\mathrm{s}}[\mathsf{src}] = \infty,\ \check{\mathrm{s}}[a] = 0,\ \check{\mathrm{s}}[b] = 0,\ \check{\mathrm{s}}[c] = \infty,\ \check{\mathrm{s}}[\mathsf{dst}] = 0 \Big\rangle.$$
4. Let us suppose that the channel $(\mathsf{src}, \mathrm{a})$ is relaxed next. The skip estimate values are updated to
$$\Big\langle \check{\mathrm{s}}[\mathsf{src}] = 0,\ \check{\mathrm{s}}[a] = 0,\ \check{\mathrm{s}}[b] = 0,\ \check{\mathrm{s}}[c] = \infty,\ \check{\mathrm{s}}[\mathsf{dst}] = 0 \Big\rangle.$$
5. Let us suppose that the channel (c, a) is relaxed next. From Expression 9 we have
$$\check{\mathrm{s}}[c] \leftarrow \left\lfloor \frac{\mathsf{delay}(\mathrm{c}, \mathrm{a}) + \check{\mathrm{s}}[a] \cdot \mathsf{cons}(\mathrm{c}, \mathrm{a})}{\mathsf{prod}(\mathrm{c}, \mathrm{a})} \right\rfloor = \left\lfloor \frac{16 + 0 \cdot 8}{2} \right\rfloor = 8.$$
Hence the skip estimate values are updated to
$$\Big\langle \check{\mathrm{s}}[\mathsf{src}] = 0,\ \check{\mathrm{s}}[a] = 0,\ \check{\mathrm{s}}[b] = 0,\ \check{\mathrm{s}}[c] = 8,\ \check{\mathrm{s}}[\mathsf{dst}] = 0 \Big\rangle.$$

It may be verified that no further relaxations are possible: procedure Compute Skip-Vectors has converged after just four relaxations. The final values for the skip vector that are computed by procedure Compute Skip-Vectors, are therefore the estimates given above. (Observe that this matches with what we had informally argued at the beginning of Section 4, when we had reasoned that although $\mathrm{q}[c] = 12$, it suffices to fire actor c four times (i.e., skip $(12 - 4) = 8$ firings.)

5 Conclusions

The Synchronous Data Flow Graph (SDFG) model is widely used in the modeling of embedded real-time systems. In this research, we have attempted to apply ideas, techniques, and results from real-time scheduling theory to the analysis of systems represented using this model. We have developed what is, to our knowledge, the first optimal algorithm for dynamically scheduling a collection of such tasks upon a preemptive uniprocessor platform. Our algorithm achieves optimality by exploiting the presence of initial tokens to 'skip' (actually, delay) the executions of some actors. Significant improvement in performance over prior approaches depends upon the presence of a relatively large number of initial tokens in the SDFG under consideration; while this may be the case for only a limited class of systems, we hope that the theoretical insights provided by our optimal algorithm will lead to additional results that may be applicable to a wider variety of systems.

In addition to this particular result, we believe that a major contribution of this paper lies in its opening up a plethora of problems concerning real-time data-flow models to scrutiny by the real-time scheduling theory community. There are many aspects of the SDF model that are of interest to the SDF community that we have chosen to ignore in this paper, that merit further attention – of particular note are consideration of bounds on channel buffer sizes, and extension to multiprocessor platforms. We are optimistic that some of these aspects will prove amenable to analysis using recently-developed techniques of real-time scheduling theory.

References

1 H. I. Ali, B. Akesson, and L. M. Pinho. Generalized extraction of real-time parameters for homogeneous synchronous dataflow graphs. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 701–710, March 2015. doi:10.1109/PDP.2015.57.

2 M. Bamakhrama and T. Stefanov. Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT'11, pages 195–204, New York, NY, USA, 2011. ACM. doi:10.1145/2038642.2038672.

3 M. Bamakhrama and T. Stefanov. Managing latency in embedded streaming applications under hard-real-time scheduling. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS'12, pages 83–92, New York, NY, USA, 2012. ACM. doi:10.1145/2380445.2380464.

4 S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 182–190, Orlando, Florida, 1990. IEEE Computer Society Press. doi:10.1109/REAL.1990.128746.

5 A. Bouakaz, T. Gautier, and J. P. Talpin. Earliest-deadline first scheduling of multiple independent dataflow graphs. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 1–6, Oct 2014. doi:10.1109/SiPS.2014.6986102.

6 T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.

7 M. Dertouzos. Control robotics : the procedural control of physical processors. In *Proceedings of the IFIP Congress*, pages 807–813, 1974.

8 E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959. doi:10.1007/BF01386390.

9 N. Fisher, T. Baker, and S. Baruah. Algorithms for determining the demand-based load of a sporadic task system. In *Proceedings of the International Conference on Real-time Computing Systems and Applications*, Sydney, Australia, August 2006. IEEE Computer Society Press. doi:10.1109/RTCSA.2006.12.

10 A. H. Ghamarian, S. Stuijk, T. Basten, M. C. W. Geilen, and B. D. Theelen. Latency minimization for synchronous data flow graphs. In *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, pages 189–196, Aug 2007. doi:10.1109/DSD.2007.4341468.

11 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'16, pages 87–96, New York, NY, USA, 2016. ACM. doi:10.1145/2997465.2997479.

12 Enagnon Cédric Klikpo and Alix Munier-Kordon. Preemptive scheduling of dependent periodic tasks modeled by synchronous dataflow graphs. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, RTNS'16, pages 77–86, New York, NY, USA, 2016. ACM. doi:10.1145/2997465.2997474.

13 E. A. Lee. *A Coupled Hardware and Software Architecture for Programmable Digital Signal Processors*. PhD thesis, EECS Department, University of California, Berkeley, 1986. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1986/715.html.

14 E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, January 1987. doi:10.1109/TC.1987.5009446.

15 E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept 1987. doi:10.1109/PROC.1987.13876.

16 E. A. Lee and S. A. Seshia. *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*. MIT Press, 2011. URL: http://LeeSeshia.org.

17 C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973. doi:10.1145/321738.321743.

18 M. Mohaqeqi, J. Abdullah, and W. Yi. Modeling and analysis of data flow graphs using the digraph real-time task model. In *Proceedings of the 21st Ada-Europe International Conference on Reliable Software Technologies – Ada-Europe 2016 – Volume 9695*, pages 15–29, New York, NY, USA, 2016. Springer-Verlag New York, Inc. doi:10.1007/978-3-319-39083-3_2.

19 A. Mok. *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1983. Available as Technical Report No. MIT/LCS/TR-297. URL: http://hdl.handle.net/1721.1/15670.

20 Naval Research Laboratory. *PGM – Processing Graph Method Specification*, December 1987. Prepared by the Naval Research Laboratory for use by the Navy Standard Signal Processing Program Office (PMS-412). Version 1.0.

21 I. Ripoll, A. Crespo, and A. K. Mok. Improvement in feasibility testing for real-time tasks. *Real-Time Systems: The International Journal of Time-Critical Computing*, 11:19–39, 1996.

22 F. Siyoum. *Worst-case temporal analysis of real-time dynamic streaming applications*. PhD thesis, PhD thesis, Eindhoven University of Technology, 2014. doi:10.6100/IR780952.

23 S. Sriram and S. S. Bhattacharya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2000.

24 M. Stigge. *Real-Time Workload Models: Expressiveness vs. Analysis Efficiency*. PhD thesis, Ph.D. thesis, Uppsala University, 2014.

25 M. Stigge, P. Ekberg, N. Guan, and W. Yi. The digraph real-time task model. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 71–80, April 2011. doi:10.1109/RTAS.2011.15.

26 M. Stigge and W. Yi. Graph-based models for real-time workload: A survey. *Real-Time Syst.*, 51(5):602–636, September 2015. doi:10.1007/s11241-015-9234-z.

27 S. Stuijk, M. C. W. Geilen, and T. Basten. SDF^3: SDF For Free. In *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*, pages 276–278. IEEE Computer Society Press, Los Alamitos, CA, USA, June 2006. URL: http://www.es.ele.tue.nl/sdf3, doi:10.1109/ACSD.2006.23.

28 F. Zhang and A. Burns. Schedulability analysis for real-time systems with edf scheduling. *IEEE Transactions on Computers*, 58(9):1250–1258, Sept 2009. doi:10.1109/TC.2009.58.

On the Pitfalls of Resource Augmentation Factors and Utilization Bounds in Real-Time Scheduling*

Jian-Jia Chen[1], Georg von der Brüggen[2], Wen-Hung Huang[3], and Robert I. Davis[4]

1 **TU Dortmund University, Dortmund, Germany**
`jian-jia.chen@cs.uni-dortmund.de`
2 **TU Dortmund University, , DortmundGermany**
`georg.von-der-brueggen@tu-dortmund.de`
3 **TU Dortmund University, Dortmund, Germany**
`wen-hung.huang@tu-dortmund.de`
4 **University of York, York, UK; and**
INRIA Paris, Paris, France
`rob.davis@york.ac.uk`

Abstract

In this paper, we take a careful look at speedup factors, utilization bounds, and capacity augmentation bounds. These three metrics have been widely adopted in real-time scheduling research as the *de facto* standard theoretical tools for assessing scheduling algorithms and schedulability tests. Despite that, it is not always clear how researchers and designers should interpret or use these metrics. In studying this area, we found a number of surprising results, and related to them, ways in which the metrics may be misinterpreted or misunderstood. In this paper, we provide a perspective on the use of these metrics, guiding researchers on their meaning and interpretation, and helping to avoid pitfalls in their use. Finally, we propose and demonstrate the use of *parametric augmentation functions* as a means of providing nuanced information that may be more relevant in practical settings.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems, I.1.2 Analysis of Algorithms

Keywords and phrases real-time systems, speedup factors, utilization bounds, capacity augmentation bounds

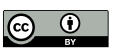
Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.9

1 Introduction

The performance of schedulability tests and scheduling algorithms for real-time systems can be compared in different ways. These can be broadly classified into two categories:

- *Theoretical methods* include deriving dominance relationships, utilization bounds, and various forms of resource augmentation factors, such as speedup factors, capacity augmentation bounds, or approximation ratios. These latter approaches typically give a worst-case comparison against a specific competitor, i.e. against an alternative schedulability test for the same or a different scheduling algorithm.

* This paper is supported by the DFG, as part of the Collaborative Research Center SFB876 (`http://sfb876.tu-dortmund.de/`), and by the ESPRC grant, MCCps (EP/K011626/1), and by the Inria International Chair program. EPSRC Research Data Management: No new primary data was created during this study.



- *Empirical methods* include simulation of the scheduling algorithm, evaluation of the schedulability test on synthetic task sets, case studies, and experiments on real hardware. These approaches typically facilitate an average-case comparison against a number of different scheduling algorithms or schedulability tests. See [36] for a review.

This paper is concerned primarily with theoretical methods. The main approaches are outlined in more detail below. Note, when we discuss comparisons between scheduling algorithms, then we are normally referring to comparisons between exact schedulability tests for those algorithms. Comparisons are also possible using sufficient schedulability tests, thus evaluating the performance of different approximations.

- *Dominance Relationships* are used to indicate if one scheduling algorithm or schedulability test always outperforms another. For example, schedulability test $\mathcal{X}$ is said to dominate test $\mathcal{Y}$ if every task set that is schedulable according to test $\mathcal{Y}$ is also schedulable according to test $\mathcal{X}$, and there are some task sets that are schedulable according to $\mathcal{X}$ but not according to $\mathcal{Y}$. Proving a dominance relationship shows that the dominant method is always better, at least in terms of schedulability; however, no indication is given as to how good the schedulability tests (or algorithms) actually are; a dominant test may still have poor performance, just not quite as poor as that of the test that it dominates.
- *Utilization Bounds* [65, 1, 5, 49] seek to bound the minimum total utilization of any unschedulable task set for a given scheduling algorithm and task model. Thus any task set with a total utilization U_{sum} no greater than the bound is guaranteed to be schedulable. Examples include the Liu and Layland bounds for implicit-deadline sporadic task sets scheduled using earliest deadline first preemptive (EDF-P) ($U_{sum} \leq 1.0$) or fixed priority preemptive (FP-P) scheduling with rate-monotonic priority assignment (also called rate-monotonic (RM) scheduling) ($U_{sum} \leq \ln 2 \approx 0.693$) on a single processor. We note that there are also utilization-based schedulability tests that make use of task utilizations in hyperbolic or quadratic forms [23, 25, 32, 31, 51].
- *Speedup Factors* [56, 68] indicate the factor ρ by which the overall speed of a system would need to be increased so that any task set that was schedulable under a scheduling algorithm $\mathcal{A}$ is guaranteed to be schedulable under scheduling algorithm $\mathcal{B}$. We note that the increase in speed implies that the worst-case execution time (WCET) of each task is reduced by a factor of ρ. Speedup factors illustrate the worst-case performance that one scheduling algorithm can have relative to another. For example, the speedup factor for FP-P scheduling versus EDF-P scheduling is $\frac{1}{\ln 2} \approx 1.44269$ for implicit-deadline task sets on a uniprocessor. Speedup factors can be used to explore sub-optimality with respect to an optimal algorithm, e.g. comparing non-preemptive scheduling algorithms against EDF-P [35] or to make relative comparisons between two non-optimal algorithms, e.g. comparing fixed priority non-preemptive (FP-NP) and EDF non-preemptive (EDF-NP) scheduling [35, 70]. We note that the usefulness of speedup factors is diminished, if no schedulability test is available for the reference algorithm.
- *Capacity Augmentation Bounds* [2, 63, 64] for homogeneous multiprocessor systems quantify scheduling algorithms or schedulability tests via a threshold b, such that the algorithm or test guarantees schedulability of any task set τ provided that $\max_{\tau_i \in \tau} U_i \leq \frac{1}{b}$ and $U_{sum} \leq \frac{M}{b}$, where M is the number of processors, and U_i is the utilization of task τ_i in task set τ with total utilization U_{sum}. The notion of capacity augmentation bounds was formally introduced in 2013 by Li et al. [63] to quantify global-EDF scheduling of task sets where each task can be further characterized using a directed acyclic graph (DAG). In this case, as sub-tasks may execute in parallel, the condition $\max_{\tau_i \in \tau} U_i \leq \frac{1}{b}$

is replaced by $\max_{\tau_i \in \tau} \frac{Critical_i}{T_i} \leq \frac{1}{b}$, where $Critical_i$ is the length of the critical path in the DAG of task τ_i. The capacity augmentation bound of global-EDF was shown to be $\frac{3+\sqrt{5}}{2} \approx 2.6181$ [63, 64] in this case. We note that capacity augmentation bounds differ from speedup factors. For example global-EDF has a speedup factor of 2 in the above case of DAG tasks.

- *Approximation Ratios* for homogeneous multiprocessor systems compare the number of processors needed by (i) scheduling algorithm $\mathcal{A}$ and (ii) an optimal algorithm, to schedule any given task set, as the number of processors required by the optimal algorithm tends to infinity. See [39] for a precise definition. Approximation ratios have been used to characterize partitioned multiprocessor scheduling, with ratios of 1.7 and 1.5 derived for EDF First-Fit [46] and RM Matching [57] respectively. While these approximation ratios can be used to compare different algorithms, their practical use is severely limited. This is because determining the minimum number of processors required by an optimal algorithm is NP-hard, equivalent to the Bin Packing problem.

In this paper, we take a careful look at speedup factors, utilization bounds, and capacity augmentation bounds. Although these three metrics, referred to as *the resource augmentation factors and bounds*, have been widely adopted and accepted by the real-time scheduling research community as the *de facto* standard theoretical tools for assessing scheduling algorithms and schedulability tests, it is not always clear how researchers and designers should view or use such theoretical results. In studying this area, we found a number of surprising results, and related to them, ways in which these metrics can be misinterpreted or misunderstood. The aim of this work is to provide a perspective on the use of these metrics, guide researchers on their meaning and interpretation, and help avoid common pitfalls. Central to this purpose, we seek to answer the following questions:

(Q1) What are the actual meanings of the resource augmentation factors and bounds and how should they be interpreted?

(Q2) Algorithm $\mathcal{A}$ has a better resource augmentation factor or bound than Algorithm $\mathcal{B}$. Does this mean that the performance of $\mathcal{A}$ is *always* better than that of $\mathcal{B}$?

(Q3) Enforcement rules may be employed in algorithm design to achieve good resource augmentation factors or bounds; can they result in design pitfalls that are detrimental to performance?

(Q4) Are resource augmentation factors meaningful when the reference algorithm is not optimal?

(Q5) How can we enhance the information provided by resource augmentation factors and bounds to give a broader perspective on performance?

We answer these questions and present our key observations based on several research results for different scheduling problems and models.

2 Task Model and Recap on Uniprocessor Scheduling

In this section, we introduce the well-known sporadic task model, uniprocessor platform, and EDF and fixed priority scheduling algorithms. These are used in many of the subsequent examples. Other examples introduce more complex models e.g. self-suspending tasks, and multiprocessor platforms by building upon the notation, and models described below.

In the sporadic real-time task model, a task set τ comprises n tasks identified by their indices from 1 to n. Each task τ_i has a WCET of C_i, relative deadline D_i, and period or minimum inter-arrival time T_i. Each task τ_i releases a potentially unbounded number of

task instances (or jobs), separated by at least T_i. If $D_i = T_i$ holds for every task, then τ is an *implicit-deadline* task set. Similarly, if $D_i \leq T_i$ holds for every task, then τ is a *constrained-deadline* task set. Finally, *arbitrary-deadline* task sets are the most general and may also have tasks with deadlines that are longer than their periods. The utilization U_i of task τ_i is defined as C_i/T_i. The total utilization U_{sum} of the task set is the sum of the utilizations of its tasks. The task set executes on a platform which has M identical processors, where $M = 1$ for a uniprocessor system. The tasks are assumed to be independent, i.e., they do not share any resources except for the processor, and they do not suspend themselves. Note that we return to multiprocessor scheduling and self-suspending tasks later. Under fixed priority scheduling, each task is assigned a unique static priority, which is inherited by all of its jobs. Without loss of generality, we assume that the task index reflects this priority; thus task τ_1 has the highest priority and task τ_n the lowest.

For the sporadic task model described above, EDF-P is an optimal uniprocessor scheduling algorithm [45]. Although FP-P scheduling is not optimal in the uniprocessor case, it is widely used in practice. With fixed priority scheduling, priority assignment is an important factor in obtaining a schedulable system [42]. In the preemptive case, rate-monotonic (RM) priority assignment is optimal [65] for implicit-deadline task sets, deadline-monotonic (DM) priority assignment is optimal [62] for constrained-deadline task sets, and for arbitrary-deadline task sets, the Optimal Priority Assignment (OPA) algorithm of Audsley et al. [6] may be used to obtain an optimal priority ordering. For ease of reference, we refer to fixed priority scheduling with RM priority assignment as RM scheduling, and similarly with DM priority assignment as DM scheduling. For FP-P scheduling exact schedulability tests have been developed that have pseudo-polynomial-time complexity [61, 55, 6, 60], as well as sufficient schedulability tests, with polynomial-time complexity [65, 23, 34, 25, 32, 31].

A scheduling algorithm is called work-conserving if it never idles the processor when there is a job ready to be executed. Among uniprocessor work-conserving non-preemptive scheduling algorithms for sporadic task sets, EDF-NP scheduling is optimal [47]. In the case of FP-NP scheduling, RM and DM priority assignments are no longer optimal; however, Audsley's OPA algorithm can be used to obtain optimal priority orderings for all three classes of task sets. Exact schedulability tests have been derived for FP-NP scheduling [47, 41], that have exponential time complexity, as well as sufficient schedulability tests that have pseudo-polynomial-time complexity [41, 72], or polynomial-time complexity [34, 5, 70].

3 The Meaning and Interpretation of Augmentation Factors

Utilization bounds, speedup factors, and capacity augmentation bounds have been widely used in the literature to theoretically quantify the performance of scheduling algorithms and schedulability tests. However, due to the way these quantification metrics are defined, they *focus entirely on the worst-case scenario* and quantify it via a single value. This can make the augmentation bounds poorly suited to distinguishing between the performance of different algorithms or tests. Different algorithms or tests may have identical performance in the worst-case, but very different performance across a broad spectrum of other cases. Further, the worst-case scenario may be a specific corner case that is far removed from practical interest. To illustrate these points, we consider uniprocessor FP-P scheduling of implicit-deadline task sets, and then broaden our view to constrained- and arbitrary-deadline task sets and also to non-preemptive scheduling.

In 1973, Liu and Layland [65] presented the seminal utilization bound $\ln 2 \approx 69.3\%$ for RM scheduling of periodic tasks, which directly leads to a speedup factor of $1/ln(2) \approx 1.44269$

with respect to EDF-P. In 1989 Lehoczky et al. [61] provided a stochastic analysis, showing that the average case is much better than the worst-case behavior, with an average breakdown utilization of 88%. Bini [22] later showed that the optimality degree is even higher (over 90%) when task utilization is uniformly distributed.

A more precise schedulability test that also considers the ratios of task periods was presented by Burchard et al. [26] in 1995. In 1997, Han and Tyan [50] proposed a task transformation technique to convert a set of periodic tasks into a corresponding harmonic task set, such that T_i is an integer multiple of T_j if $T_i \geq T_j$. The utilization bound in [50] analytically dominates those given by Liu and Layland [65] and Burchard et al. [26]. In 1998 Lauzac et al. [59] proposed a utilization bound of $\ln r + 2/r - 1$ based on the ratio r of the minimum task period to the maximum task period if $1 \leq r \leq 2$. When r is 2, this bound is $\ln 2$, the same as the Liu and Layland bound. The harmonic relationship of the task periods was further exploited by Kuo et al. [58] to improve the utilization bound. In 2001, Bini and Buttazzo [21] presented the hyperbolic bound $\prod_{\tau_i \in \tau}(1 + U_i) \leq 2$. More recently, in 2015, Chen et al. [31] developed a utilization-based analysis framework called k2U that can provide hyperbolic bounds almost automatically.

The Liu and Layland utilization bound of $\ln 2$ is independent of the task parameters, while the improvements in the other utilization bounds are based on characteristics of the task set. However, when n is sufficiently large the following worst-case scenario [65] for RM scheduling remains valid for all of the tests mentioned above:

- $T_1 = D_1 = 1$, $C_1 = (2^{\frac{1}{n}} - 1)$.
- $T_i = T_{i-1} + C_{i-1}$, $C_i = (2^{\frac{1}{n}} - 1)T_i, \forall i = 2, 3, \ldots, n-1$.
- $T_n = T_{n-1} + C_{n-1}$, $C_n = (2^{\frac{1}{n}} - 1)T_n + \varepsilon T_n$ where $\varepsilon > 0$ is an arbitrarily small number.

This task set, denoted by τ^{RM}, has a utilization $n(2^{\frac{1}{n}} - 1) + \varepsilon = \ln 2 + \varepsilon$ as $n \to \infty$ and is not schedulable by RM scheduling since task τ_n misses its deadline. All the schedulability tests mentioned above, including [65, 61, 26, 50, 21, 58, 59, 31], indicate that the above task set is not schedulable using RM scheduling. Since all the schedulability tests mentioned above analytically dominate the Liu and Layland bound of $\ln 2$, we conclude that the speedup factor of any of the above schedulability tests is $\frac{1}{\ln 2}$ with respect to EDF-P. Further, since task set τ^{RM} remains schedulable under EDF-P at speed s as long as $\sum_{\tau_i \in \tau^{RM}} \frac{U_i}{s} \leq 1$, it follows that a lower bound on the speedup factor of RM scheduling is also $\frac{1}{\ln 2}$.

Somewhat surprisingly, we can therefore conclude that every one of the schedulability tests in [55, 6, 65, 61, 26, 50, 21, 58, 59, 31] is a *speedup-optimal* schedulability test for RM scheduling of implicit deadline task sets, with respect to an optimal scheduling algorithm, i.e., EDF-P. In other words all of these tests have the minimum possible speedup factor for the class of scheduling algorithms considered, i.e. fixed priority preemptive scheduling. This is the case despite the fact that only [61, 6, 55] are exact tests. Contrary to fact that all of the tests cited above have the same speedup factor, it is well known that they have very different performance in terms of schedulability, as demonstrated by empirical evaluations using synthetic task sets to examine schedulability by measuring acceptance ratios.

The above discussion illustrates the lack of discrimination between these tests when assessed using speedup factors. This is also apparent when utilization bounds (i.e., $\ln 2$) and capacity augmentation bounds (i.e., $\frac{1}{\ln 2}$) are used for assessment. Moreover, Table 1 presents the speedup factors for DM scheduling in both preemptive and non-preemptive cases. The results in Table 1 show that the exact schedulability tests, with pseudo-polynomial or exponential time complexity, have the same speedup factor as some corresponding linear-time sufficient schedulability tests. Further, although DM is not an optimal priority assignment policy for FP-P scheduling of arbitrary-deadline task sets, or for FP-NP scheduling of any of

Table 1 Speedup factors: lower bounds, upper bounds for linear-time schedulability tests, and upper bounds for pseudo-polynomial / exponential-time schedulability tests.

Constraints	Preemptive			Non-Preemptive		
	lower bound	upper bound (DM, linear)	upper bound (DM, expo.)	lower bound	upper bound (DM, linear)	upper bound (DM, expo.)
implicit-deadline	$1/ln(2) \approx 1.44269$ [65]			$1/\Omega \approx 1.76322$ [43]	$1/\Omega \approx 1.76322$ [70]	$1/\Omega \approx 1.76322$ [70]
constrained-deadline	$1/\Omega \approx 1.76322$ [44]	$1/\Omega \approx 1.76322$ [31]	$1/\Omega \approx 1.76322$ [44]	$1/\Omega \approx 1.76322$ [43]	$1/\Omega \approx 1.76322$ [70]	$1/\Omega \approx 1.76322$ [70]
arbitrary-deadline	2 [40]	2 [69]	2 [37]	2 [40]	2 [69]	2 [43]

the three classes of task set, it is an *optimal* fixed-priority scheduling strategy with respect to the speedup factors, since the upper bounds on the speedup factors for DM priority assignment [69, 70] are the same as the lower bounds assuming optimal priority assignment and exact tests [40, 43].

The lack of discrimination between different schedulability tests does not just hold for uniprocessor fixed-priority scheduling, but also for multiprocessor partitioned fixed-priority scheduling with constrained-deadline and arbitrary-deadline sporadic task sets, as recently reported by Chen [29]. The analyses in [29] showed that the achieved speedup factors are identical when using exponential-time exact schedulability tests and polynomial-time sufficient schedulability tests. Similarly, for uniprocessor mixed-criticality scheduling, Baruah showed in a series of papers [11, 12, 14] that EDF-VD and its generalization have the same speedup factor for different task models.

▶ **Observation 1.** *Speedup factors, utilization bounds and capacity augmentation bounds often lack the power to discriminate between the performance of different scheduling algorithms and schedulability tests even though the performance of these algorithms and tests may be very different when viewed from the perspective of empirical evaluation.*

The rationale for Observation 1 is that utilization bounds, capacity augmentation bounds and speedup factors *only* reflect the worst-case corner cases. Having a constant factor or bound, e.g., $\frac{1}{\ln 2}$ or $\ln 2$, does not have any implication with regard to the performance of the algorithm or test in typical cases or in the average cases. Worse, the structure of the corner cases may be easily captured by simple tests, e.g., the hyperbolic bound or the Liu and Layland utilization bound. As other cases, in the broad space of possible task sets, do not contribute to the metric even if the algorithm or the test has relatively poor performance there, then they are simply ignored. Therefore, it is possible that very simple algorithms or very imprecise sufficient schedulability tests may be classified as excellent or even optimal results as long as they can also handle these corner cases well. This explains the results listed in Table 1; even though their performance, in terms of schedulability across a wide range of task sets, is very different.

▶ **Observation 2.** *Speedup factors, utilization bounds and capacity augmentation bounds should only be considered for their negative implications, since these metrics only provide information on performance in the worst case.*

▶ **Observation 3.** *Proving that an algorithm or test has the best possible (or optimal) speedup factor or bound for that class of algorithms does not imply that the algorithm or test cannot be substantially improved upon.*

If an algorithm does not have a constant speedup factor, utilization bound, or capacity augmentation bound, it may still perform reasonably well; however, it may also perform terribly in the worst case. A constant bound or factor only ensures that the performance of the algorithm at least reaches some minimum level in the worst case. Even showing that an

algorithm or test has the best possible (or optimal) speedup factor or bound for the studied problem [13, 15] does *not* imply that its performance will necessarily be good in other cases. As a result, as researchers, we should not be satisfied with just deriving algorithms or tests that have optimal speedup factors or bounds. Rather these can be seen as a step towards developing algorithms and tests that, while retaining performance in the worst-case, provide improved performance in practice.

4 Non-dominance Based on Speedup Factors

In this section, we use uniprocessor FP-P scheduling as an example to examine the relationship between dominance results based on speedup factors and utilization bounds, and schedulability as assessed via empirical evaluation in terms of acceptance ratios. To verify the schedulability of a constrained-deadline task τ_k under uniprocessor fixed-priority scheduling, time-demand analysis (TDA) [61] can be applied. That is, task τ_k is schedulable under FP-P scheduling if and only if

$$\exists t | 0 < t \leq D_k, \quad C_k + \sum_{\tau_i \in hp(\tau_k)} \left\lceil \frac{t}{T_i} \right\rceil C_i \leq t, \tag{1}$$

where $hp(\tau_k)$ is the set of tasks with higher-priority than task τ_k.

Throughout this section, we consider implicit-deadline task sets indexed in the rate-monotonic order i.e., τ_i has higher priority than τ_j if $i \leq j$, and thus $T_i \leq T_{i+1}$. There are many sufficient tests in the literature for testing the schedulability of task τ_k, as explained in Section 3. Here, we consider two sufficient schedulability tests:

- *Hyperbolic bound* (HB) by Bini and Buttazzo [21]: task τ_k is schedulable by RM if $\prod_{i=1}^{k}(U_i + 1) \leq 2$. This test has a utilization bound of $\ln 2 \approx 0.693147$, and hence a speedup factor of $\frac{1}{\ln 2} \approx 1.44269$ compared to EDF-P.
- *Quadratic bound* (QB) by Davis and Burns in Equation (26) in [34], and Bini et al. in Equation (11) in [24]: task τ_k is schedulable by RM scheduling if

$$\sum_{i=1}^{k} U_i + \frac{\sum_{i=1}^{k-1} C_i - \sum_{i=1}^{k-1} U_i C_i}{T_k} \leq 1. \tag{2}$$

  This test has a utilization bound of 0.5, and hence a speedup factor of 2 compared to EDF-P. The utilization bound was not explicitly presented in [34, 24]. A tighter quadratic bound that has a utilization bound of $2 - \sqrt{2}$ can be found in more general task models from [1, 51] or can be derived from the k2Q framework [32].

Consider different settings of $\frac{T_1}{T_2}$ with $T_2 > T_1$. Suppose that $U_1 = 0.4$. By adopting the HB, we know that task τ_2 is schedulable under RM if $U_2 \leq 2/1.4 - 1 \approx 42.8\%$. By adopting the QB in (2), we know that task τ_2 is schedulable under RM if $0.4 + U_2 + \frac{0.4T_1 - 0.4^2 T_1}{T_2} = 0.4 + U_2 + 0.24\frac{T_1}{T_2} \leq 1$. That is, if $\frac{T_1}{T_2} > 0.715$, then the QB is better; otherwise, the HB is better. As a result, HB and QB are incomparable, neither dominates the other.

According to its speedup factor and utilization bound, HB is better than QB; however, QB is better than HB as long as T_1/T_2 is smaller than ≈ 0.715 when $U_1 = 0.4$. To demonstrate the impact of different distributions of T_1/T_2, we conducted the following evaluation for implicit-deadline sporadic task sets with *only two* tasks. We considered four different configurations, in each case T_1 was set to 1 and T_2 was chosen from a different uniform distribution: (a) $[1, 1.5]$, (b) $[1, 2]$, (c) $[1.5, 2]$, and (d)$[1, 10]$, thus $T_2 \geq T_1$. In each configuration, for each utilization level, 10,000 task sets were generated as follows:

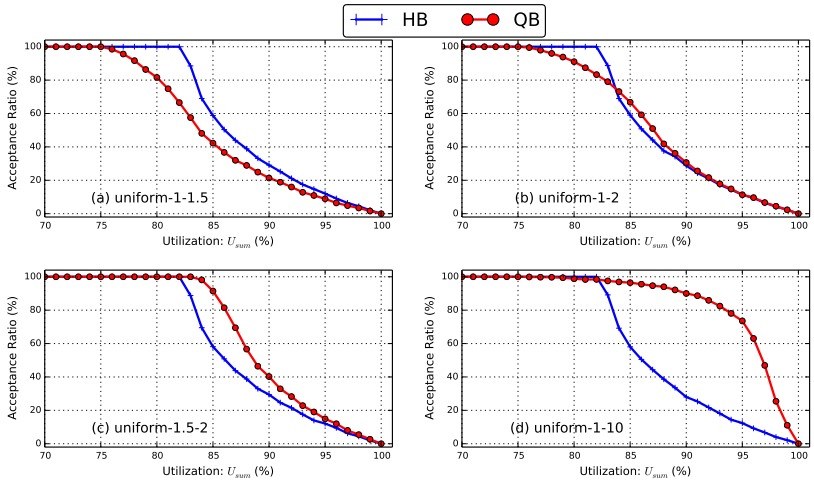


Figure 1 Adopting the hyperbolic bound (HB) and the quadratic bound (QB) for RM uniprocessor scheduling with $k = 2$ for different uniform distributions of T_2/T_1.

- For a given target utilization level U_{sum}, U_1 was chosen from a uniform distribution $[0, U_{sum}]$, with U_2 set to $U_{sum} - U_1$.
- T_1 was set to 1, and C_1 to $U_1 T_1$.
- T_2 was chosen from the uniform distribution specified for the configuration, and then C_2 was set to $U_2 T_2$.

The metric used to compare performance was the *acceptance ratio* of the HB and QB tests with respect to a given task set utilization level U_{sum}. The acceptance ratio is the percentage of generated task sets that are schedulable at that utilization level according to the test.

Figure 1 shows the results for the above configurations. The acceptance ratios of QB are highly dependent on the configuration settings for T_2/T_1, while those for HB are in general independent of these settings. QB is worse than HB if T_2/T_1 is small. Therefore, as shown in Figure 1(a), when T_2 is uniformly distributed in $[1, 1.5] \cdot T_1$, HB is in general better. In contrast, when T_2 is uniformly distributed in $[1.5, 2] \cdot T_1$, QB is always better in the evaluation, as shown in Figure 1(c). The issue with QB is that its worst case happens when T_2 is equal to T_1. The setting $T_2 \in [1.5, 2] \cdot T_1$ avoids such cases and QB benefits from such a setting. When T_2 is uniformly distributed in $[1, 2] \cdot T_1$ there is no clear winning scheme between QB and HB as shown in Figure 1(b). When U_{sum} is below 84%, HB is better, whereas when $0.85 \leq U_{sum} \leq 0.9$, QB is better. When $U_{sum} \geq 0.9$, the two tests perform almost identically. When T_2 is uniformly distributed in $[1, 10] \cdot T_1$ QB still deems most of the task sets schedulable even when U_{sum} is above 95%, as shown in Figure 1(d); however, there are still a few task sets deemed unschedulable when U_{sum} is below 82%. With this configuration, we can conclude that QB is in general much better than HB, since the schedulable utilization level of QB is much higher than that of HB.

It is clear from the evaluation results for the different configurations shown in Figure 1 that although HB has a superior speedup factor and utilization bound to QB, the relative performance of these two schedulability tests is highly dependent on the configurations used, i.e. the task set parameters.

▶ **Observation 4.** *A scheduling algorithm or schedulability test with a worse speedup factor or utilization bound may perform (much) better in practice than another algorithm or test*

with a superior speedup factor or utilization bound, dependent on the task set configurations and parameters used. Conclusions on the relative merits of algorithms or tests drawn from speedup factors or utilization bounds can therefore be in direct contradiction with those drawn from empirical performance evaluation.

This apparent contradiction between dominance in terms of speedup factors or utilization bounds and performance observed from evaluation occurs because of the disconnect between the settings for the task set parameters in the two cases. Speedup factors and utilization bounds are dependent solely on corner cases, which may have parameters that rarely occur or are far removed from practical settings. Further examples can be found in the literature:

- For global-RM scheduling, the schedulability test based on the forced forward method [16] and the test by Bertogna and Cirinei [20] both have a speedup factor of 3, compared to an optimal algorithm. When adopting bounded carry-in [48], the schedulability tests based on the k2U and k2Q frameworks by Chen et al. [31, 32] have worse speedup factors, i.e., 3.62143 for k2U and 3.73 for k2Q; however, the evaluation results in [32, 30] show that k2U and k2Q perform much better than the other two tests.
- For implicit-deadline DAG task sets (explained in Sec. 6.1) on M homogeneous processors, Jiang et al. [54] developed a decomposition algorithm which assigns a relative deadline for each DAG subtask. They proved that the capacity augmentation bound of the algorithm is in the range of $[2 - \frac{1}{M}, 4 - \frac{2}{M})$. The capacity augmentation bound under federated scheduling was proved to be $2 - \frac{1}{M}$ by Li et al. [64]. From the capacity augmentation perspective, one may conclude that federated scheduling dominates the decomposition algorithm; however, the experimental results by Jiang et al. [54] showed that their decomposition algorithm outperforms other algorithms in the experimental settings used.

▶ **Observation 5.** *Identifying regions of dominance in terms of schedulability, between scheduling algorithms and schedulability tests provides valuable information in addition to theoretical analysis in terms of speedup factors or bounds, and empirical evaluations in terms of acceptance ratios.*

5 Speedup Factors Based on Enforced Algorithms

In this section, we consider *enforcements* that are sometimes used when designing scheduling algorithms to simplify the structure of the scheduling problem, and thus allow speedup factors to be derived. If these enforcements are too strong and are applied at an early stage of the algorithm they may make it easy to derive a speedup factor for the scheduling algorithm or schedulability test. However, this may come at the expense of poor performance in practical settings when compared to other algorithms or tests that have worse speedup factors or no speedup factor at all. We illustrate this effect with two examples in the context of scheduling tasks with self-suspensions on uniprocessor and tasks that share resources on multiprocessors.

5.1 One-Segmented Self-Suspension

Equal Deadline Assignment (EDA) by Chen and Liu [33] and Shortest Execution Interval First Deadline Assignment (SEIFDA) by von der Brüggen et al. [71] are two algorithms for scheduling one-segment self-suspending tasks with implicit deadlines on a uniprocessor, using a Fixed-Relative-Deadline (FRD) scheduling strategy [33]. A one-segment self-suspending task τ_i is a periodic task that has two executions segments $C_{i,1}$ and $C_{i,2}$, with total execution time $C_i = C_{i,1} + C_{i,2}$, that are separated by one suspension interval S_i, i.e., τ_i is characterized by $((C_{i,1}, S_i, C_{i,2}), T_i, D_i)$. When a job of task τ_i is released, it has to finish $C_{i,1}$ time units

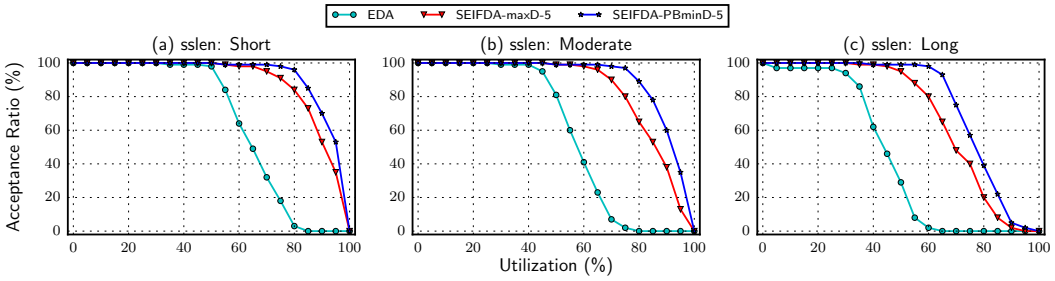


Figure 2 Comparison of the acceptance ratio for EDA [33] and SEIFDA [71], showing that enforcements in the algorithm design can lead to a huge performance loss.

of computation before it suspends itself for at most S_i time units. After that, the job has to finish $C_{i,2}$ time units of computation before its deadline. A Fixed-Relative-Deadline (FRD) scheduling strategy assigns two relative deadlines $D_{i,1}$ and $D_{i,2}$ to the first and second computation segment respectively. For implicit-deadline task sets, we can always assume that $D_{i,1} + D_{i,2} + S_i = T_i$. The interesting question for FRD is how to assign $D_{i,1}$ and $D_{i,2}$.

An intuitive strategy is the proportional deadline assignment presented by Liu et al. [67], i.e., $D_{i,1} = \frac{C_{i,1}}{C_i} \cdot (T_i - S_i)$ and $D_{i,2} = \frac{C_{i,2}}{C_i} \cdot (T_i - S_i)$. While this assignment policy sounds very reasonable, the speedup factor is unbounded as shown by Chen and Liu [33]. They propose to use EDA instead, i.e., $D_{i,1} = D_{i,2} = (T_i - S_i)/2$, and derive a speedup factor of 2 compared to any other FRD strategy and a speedup factor of 3 compared to an optimal scheduling algorithm for one-segmented self-suspension task sets. The speedup factor of 2 compared to FRD strategies easily follows from the enforcement for the relative deadlines. As both segments have half of the execution interval $T_i - S_i$ to execute the necessary workload, any other FRD scheduling strategy could assign at most twice the relative deadline that EDA assigns to any segment. However, this enforcement is strong as it gives the same deadline to two computation segments even if $C_{i,1} = \varepsilon$ and $C_{i,2} = C_i - \varepsilon$, where ε is a very small positive value, jeopardizing schedulability.

The main problem with EDA is that the assignment of deadlines for one task is independent from the deadline assignment for other tasks. This problem was tackled by von der Brüggen et al. [71] when they proposed the SEIFDA algorithm that assigns the relative deadlines of the tasks in decreasing order of the execution intervals $T_i - S_i$, taking the previously assigned deadlines into account when those deadlines are assigned. For the shorter execution segment, which for ease of explanation we assume is $C_{i,1}$, they calculate the possible values of $D_{i,1} \in [C_{i,1}, T_i - S_i]$ and choose one of these values according to one of three strategies: (i) the minima value (minD), (ii) the maxima value (maxD) or (iii) the minima value larger than $\frac{C_{i,1}}{C_i} \cdot (T_i - S_i)$, i.e., proportionally bounded minD (PBminD). While SEIFDA-maxD dominates EDA and SEIFDA-PBminD dominates proportional assignment, they also show that SEIFDA-minD and SEIFDA-maxD are incomparable. All three assignment strategies have the same speedup factors as EDA while clearly outperforming EDA in terms of acceptance ratios, as can be seen in Figure 2 for SEIFDA-maxD-5 and SEIFDA-PBminD-5. As in [71] an approximated schedulability test is used, the suffix 5 indicates that five periods are calculated exactly before the approximation takes place. The reason for the significantly better performance is that SEIFDA does not enforce the deadlines but chooses them dependent on the other tasks. The results shown in Figure 2 are taken directly from [71] and consider randomly generated task sets with 10 tasks. They are shown for short, medium, and long suspension lengths (sslen) respectively.

5.2 Multiprocessor Scheduling with Resource Sharing

A further example of how enforcement can compromise performance comes from scheduling algorithms for tasks that share resources and execute on a platform with M homogeneous processors. The number of shared resources is denoted by R. Here, we assume that $R \leq M$ and consider the simplified execution structure presented by Andersson and Raravi [4], where each task has only one critical section where it may access shared resources guarded by semaphores. This means, each sporadic task τ_i has three execution segments with WCETs $C_{i,1}^N$, C_i^{Crit}, and $C_{i,2}^N$ representing the part before the critical section, the critical section itself, and the part after the critical section. The WCET of task τ_i is $C_i = C_{i,1}^N + C_i^{Crit} + C_{i,2}^N$. Here, we compare two algorithms for implicit-deadline task sets with known speedup factors.

- Andersson and Raravi [4] developed the algorithm *LP-EE-vpr* that has a speedup factor of $4 \cdot (1 + \lceil \frac{R}{M} \rceil) = 8$ in the given setting, since $R \leq M$. Here, we simplify the formulas given in [4] to match the case we analyse. The model in [4] is more general. *LP-EE-vpr* creates M virtual processors with speed $\frac{1}{2}$ to schedule the two non-critical sections of task τ_i and assigns relative deadlines of $\frac{C_{i,1}^N}{C_i} \cdot \frac{T_i}{2}$ and $\frac{C_{i,2}^N}{C_i} \cdot \frac{T_i}{2}$ to them. Non-critical sections are scheduled on virtual processors by using partitioned EDF-P. It also creates M virtual processors with speed $\frac{1}{2}$ to schedule the critical section C_i^{Crit} with a relative deadline $\frac{T_i}{2}$. Critical sections guarded by one semaphore are executed exclusively on one virtual processor using EDF-NP.
- Huang et al. [53] proposed resource-oriented partitioned PCP, called *ROP-PCP*, that has a speedup factor of $11 - \frac{6}{M+1}$. *ROP-PCP* uses two dedicated subsets of the M processors to execute the critical and the non-critical sections individually. Although *ROP-PCP* is applicable to systems in which a task has multiple critical sections, here we limit consideration to one critical section for comparison with *LP-EE-vpr*.

Comparing the speedup factors, one might assume that *LP-EE-vpr* outperforms *ROP-PCP*; however, as *LP-EE-vpr* uses enforcements early in the algorithm to be able to provide the speedup factor while *ROP-PCP* first provides the algorithm and then analyzes the resulting speedup factor, this is not the case. To be precise, the enforcements of *virtualization* at slower speeds in *LP-EE-vpr* and shortened relative deadlines substantially reduce schedulability.

As no schedulability test for *LP-EE-vpr* was provided by Andersson and Raravi [4], we use two necessary conditions for the schedulability of the non-critical execution and the critical section, respectively. This is compared to a sufficient schedulability test for *ROP-PCP* by Huang et al. [53]. For task τ_i, considering the non-critical sections, workload amounting to $C_{i,1}^N + C_{i,2}^N$ must be finished with a relative deadline $\frac{T_i}{2}$ at speed $\frac{1}{2}$. After that, this amount of workload has to be finished every period T_i, leading to the following necessary condition, based on the demand bound function:

$$\forall 0 < t \leq \frac{T_{max}}{2}, \qquad \sum_{\tau_i} \left(\left\lfloor \frac{t + \frac{T_i}{2}}{T_i} \right\rfloor \times 2(C_{i,1}^N + C_{i,2}^N) \right) \leq M \times t, \tag{3}$$

where T_{max} is the maximum among the periods in the task set. The factor 2 in the summation is due to the speed $\frac{1}{2}$ of the virtual processors. We test this necessary condition at each time point $t \in (0, \frac{T_{max}}{2}]$ where the workload changes, i.e., $\forall i \ \frac{T_i}{2} + \ell \cdot T_i \leq \frac{T_{max}}{2}$ where $\ell \in \mathbb{N}$. For the critical sections, each virtual processor has to be analyzed individually. Let the shared resources be numbered from 1 to R, let $r \in \{1, \ldots, R\}$ be one of those resources, and let $\tau^r \subseteq \tau$ be the subset of the tasks that access resource r. We create a necessary condition based on the exact schedulability test for uniprocessor EDP-NP presented by

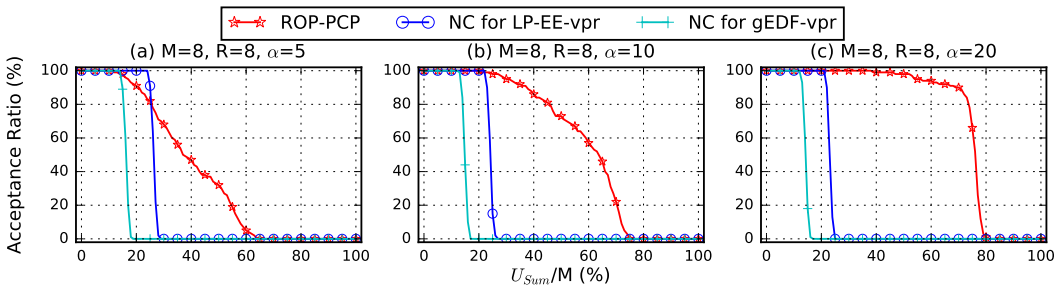


Figure 3 Comparison of the necessary condition for *LP-EE-vpr* [4], the necessary condition for gEDF-vpr [3], and the sufficient schedulability test for *ROP-PCP* [53], showing that enforcements that are made to guarantee a speedup factor can lead to a huge loss in performance.

George et al. [47] for each processor individually. For the virtual processor executing task set τ^r, the demand bound function of the task set is $DBF^r(t) = \sum_{\tau_i \in \tau^r} \left(\left\lfloor \frac{t + \frac{T_i}{2}}{T_i} \right\rfloor \times 2C_i^{crit} \right)$ for any $t > 0$, where the factor 2 is again due to the speed of $\frac{1}{2}$ of the virtual processor. In addition, for each virtual processor the blocking time $B^r(t)$, due to the shared resource r, is $B^r(t) = \max_{\forall \tau_i \in \tau^r, D_i > t}(2 \cdot C_i - \Delta)$, where $\Delta > 0$ but infinitesimally small. The exact schedulability test for uniprocessor EDP-NP by George et al. [47] tests whether (i) $\sum_{\tau_i \in \tau^r} \frac{2C_i^{Crit}}{T_i} \leq 1$ and (ii) $DBF^r(t) + B^r(t) \leq t \; \forall t > 0$. We perform a necessary test by only checking at values of t equal to the deadlines $\frac{T_i}{2}$ of the tasks $\tau_i \in \tau^r$ for each resource r.

A comparison between the sufficient test for *ROP-PCP* and the necessary condition for *LP-EE-vpr* is shown in Figure 3 for a system with 8 processors, 80 tasks, and 1 resource per task that is accessed at most once. We evaluate three ratios $\alpha = 5$, $\alpha = 10$, and $\alpha = 20$ for the length of non-critical sections to the length of critical sections, i.e., $C^{Crit} = \frac{1}{1+\alpha} \cdot C_i$ and $C_{i,1}^N + C_{i,2}^N = \frac{\alpha}{1+\alpha} \cdot C_i$. Detailed configurations can be found in [53]. In addition, we also compared to a necessary condition for *gEDF-vpr* by Anderson and Easwaran [3], which is the predecessor of *LP-EE-vpr* and was the first algorithm with a proven speedup factor, i.e., $12(1 + 3R/4M) = 21$ in our setting when $R = M$. Figure 3 shows that the acceptance ratio for *LP-EE-vpr* drops dramatically from a utilization of roughly $M \times 25\%$ and is zero by $M \times 28\%$ utilization in all cases, while the acceptance ratio for *ROP-PCP* decreases more slowly and is still over 50% when the utilization is at $M \times 76\%$ when $\alpha = 20$, $M \times 61\%$ when $\alpha = 10$ and $M \times 36\%$ when $\alpha = 5$. *gEDF-vpr* even drops before $M \times 20\%$ utilization. These results clearly show that the enforcements, which assign stringent relative deadlines to sub-tasks and enforce slow virtual processors in *LP-EE-vpr*, have significant drawbacks in terms of performance even though the speedup factor obtained is better than that for *ROP-PCP*. As a result, both *gEDF-vpr* and *LP-EE-vpr* have barely any chance to schedule task sets with a total utilization above a certain small threshold value, depending on the exact configuration. We adopt necessary conditions for *LP-EE-vpr* and *gEDF-vpr* to show that the performance loss for those methods is due to the early and restrictive enforcements.

▶ **Observation 6.** *Adding enforcements tailoring the design of a scheduling algorithm or test to facilitate the derivation of a bounded speedup factor can be counterproductive; it may severely compromise performance in practical settings.*

6 Relative Speedup Factors

The speedup factor for a scheduling algorithm or schedulability test is typically given with respect to an optimal algorithm for the same class of problem. For example, we have a speedup factor of $\frac{1}{\ln 2}$ for exact tests for RM scheduling, and also for Liu and Layland's utilization-based test with respect to EDF-P, which is an optimal scheduling algorithm for implicit-deadline sporadic task sets on a uniprocessor. Speedup factors may also be derived relative to another non-optimal algorithm or test. For example the speedup factor of FP-NP scheduling with respect to EDF-NP scheduling on a uniprocessor is 2 for arbitrary-deadline sporadic task sets and ≈ 1.76 for constrained-deadline task sets [43, 40].

It is interesting to consider how bounds on the values of speedup factors can be composed from existing results. Let $S^{\mathcal{A}\to\mathcal{B}}$ denote the speedup factor of some algorithm or test $\mathcal{A}$ with respect to algorithm or test $\mathcal{B}$. When one algorithm $\mathcal{O}$ dominates another algorithm $\mathcal{X}$ in terms of schedulability, then we have $S^{\mathcal{O}\to\mathcal{X}} = 1$ and $S^{\mathcal{X}\to\mathcal{O}} > 1$. Note this is the case when $\mathcal{O}$ is an optimal algorithm for the class of problem. Further, if two algorithms $\mathcal{X}$ and $\mathcal{Y}$ are incomparable, then we have $S^{\mathcal{X}\to\mathcal{Y}} > 1$ and $S^{\mathcal{Y}\to\mathcal{X}} > 1$, i.e. there are non-trivial speedup factors in both directions.

We now consider some simple graphs or chains of speedup factors as examples: Let $\mathcal{O}$ dominate $\mathcal{Z}$ which in turn dominates $\mathcal{Y}$, then the following relationships hold between the speedup factors: $max(S^{\mathcal{Y}\to\mathcal{Z}}, S^{\mathcal{Z}\to\mathcal{O}}) \leq S^{\mathcal{Y}\to\mathcal{O}} \leq S^{\mathcal{Y}\to\mathcal{Z}} \times S^{\mathcal{Z}\to\mathcal{O}}$. Note the first inequality holds only as a result of the dominance relationships, while the second holds regardless.

Let $\mathcal{O}$ dominate both $\mathcal{Z}$ and $\mathcal{X}$, and let $\mathcal{Z}$ and $\mathcal{X}$ be incomparable, then the following holds: $S^{\mathcal{Z}\to\mathcal{X}} \leq S^{\mathcal{Z}\to\mathcal{O}}$ and $S^{\mathcal{X}\to\mathcal{Z}} \leq S^{\mathcal{X}\to\mathcal{O}}$.

Where a speedup factor $S^{\mathcal{X}\to\mathcal{Z}}$ for some algorithm or test $\mathcal{X}$ is determined relative to some other algorithm $\mathcal{Z}$ that dominates it, but is not optimal for the scheduling problem considered, then great care needs to be taken in the interpretation of the result. In particular, if it turns out that the speedup factor of $\mathcal{Z}$ is unbounded with respect to the optimal algorithm $\mathcal{O}$, i.e. $S^{\mathcal{Z}\to\mathcal{O}} = \infty$, then so will be the speedup factor of $\mathcal{X}$ relative to the optimal algorithm, i.e., $S^{\mathcal{X}\to\mathcal{O}} = \infty$ regardless of the value of $S^{\mathcal{X}\to\mathcal{Z}}$. Utilization and capacity augmentation bounds are more robust in this respect, since their reference is the capacity of the processor. Below we give two examples based on recent studies that illustrate the pitfalls in using relative speedup factors, rather than those immediately grounded by optimal algorithms.

6.1 Federated Scheduling

To handle sporadic real-time tasks with intra-task parallelism based on DAG structures on multiprocessor platforms, Li et al. [64] and Baruah [8, 9, 10] proposed to use *federated scheduling*, defined as follows: *Either* a task is restricted to execute sequentially on a single processor while sharing this processor with other tasks; *Or*, the task has exclusive access to the processors assigned to it. This strategy was originally proposed in [64] for implicit-deadline task sets, and later extended to constrained-deadline and arbitrary-deadline task sets in [8, 9, 10]. The existing speedup factor results for federated scheduling on M identical processors can be summarized as follows:

- The capacity augmentation bound of the federated scheduling algorithm in [64] for implicit-deadline task sets is 2. Therefore, its speedup factor with respect to *an optimal scheduling algorithm* is also 2.
- The speedup factor of the federated scheduling algorithms in [8, 10] for constrained-deadline task sets is $3 - 1/M$ with respect to *an optimal federated scheduling algorithm.*

- The speedup factor of the federated scheduling algorithms in [9, 10] for arbitrary-deadline task sets is $4 - 2/M$ with respect to *an optimal federated scheduling algorithm.*

These speedup factor values are effectively the same as those for the EDF-FFID partitioning algorithm [17, 18, 19] for sporadic task sets, hence with these results, Baruah made the following conclusions about the algorithms in [8, 9, 10]:

> Baruah [8, 9, 10]: *... in terms of the speedup metric, there is no loss in going from the three-parameter sporadic tasks model to the more general sporadic DAG tasks model.*

The resulting speedup factors and conclusions in [8, 9, 10] are however only meaningful if an optimal federated scheduling algorithm also has a bounded speedup factor with respect to an optimal scheduling algorithm for this problem. If federated scheduling itself is not a provably good scheduling strategy in this case, then the constant speedup factors of 3 and 4 become less useful. Unfortunately, the following conclusion was recently presented in [28]:

> Chen [28]: *... in terms of the speedup metric with respect to any optimal scheduling algorithm, federated scheduling strategies do not yield any constant speedup factors for constrained-deadline task systems with DAG structures.*

Hence the relative speedup factors derived in [8, 9, 10] cannot be related back to an optimal scheduling algorithm, such a relation is effectively unbounded.

6.2 Uniprocessor Self-Suspension Systems

For the dynamic self-suspension task model, Huang et al. [52] studied how to schedule constrained-deadline self-suspending tasks using fixed-priority scheduling, with a unique priority level assigned to each task. It was shown in [52] that several heuristic priority assignments, including rate-monotonic (RM), deadline-monotonic (DM) and laxity-monotonic (LM), have unbounded speedup factors (Theorem 1 in [52]). Huang et al. [52] proposed using Audsley's optimal priority assignment (OPA) algorithm [7] together with an OPA-compatible schedulability test [38]. They showed that this algorithm has a speedup factor 2 with respect to the *optimal fixed-priority schedule*. Unfortunately, a recent result by Chen [27] shows that the existing scheduling algorithms (that are commonly used) do not yield any constant speedup factors in relation to an optimal algorithm for this problem:

> Chen [27]: *For dynamic self-suspending task systems,..... the speedup factor for any FP preemptive scheduling, compared to the optimal schedules, is not bounded by a constant if the suspension time cannot be reduced by speeding up. Such a statement of unbounded speedup factors can also be proved for earliest-deadline-first (EDF), least-laxity-first (LLF), and earliest-deadline-zero-laxity (EDZL) scheduling algorithms.*

6.3 Remarks on Relative Speedup Factors

Based on the two concrete examples above, we showed that arguments based on relative speedup factors may be undermined and inconclusive if the reference scheduling strategies cannot be related back to optimal algorithms. In both examples, the algorithms proposed (including the reference class of algorithms) actually have unbounded speedup factors with respect to optimal solutions.

▶ **Observation 7.** *Where relative speedup factors are used in relation to a non-optimal algorithm or class of algorithms, then great care needs to be taken in the interpretation of the results. If the reference algorithm has an unbounded speedup factor with respect to optimal solutions, then the speedup factors may not be that meaningful.*

We note that relative speedup factors can still be meaningful if (i) the reference scheduling strategies are well-accepted, defined, and constrained by the system properties, or (ii) the reference strategy facilitates comparison with an optimal algorithm. For example, considering uniprocessor scheduling, in some cases workload-conserving non-preemptive scheduling may be the only implementation option. For such systems, EDF-NP is an optimal scheduling strategy for sporadic real-time tasks. Therefore, the speedup factors between FP-NP and EDF-NP [43, 40] can be useful. In addition, though neither is an optimal algorithm, they can also be related back to EDF-P [35].

7 Parametric Augmentation Functions

As illustrated in the previous sections, using a single factor or bound to represent the theoretical quantification of the performance of scheduling algorithms or schedulability tests can prove inadequate. It would be better to always show the analytical or theoretical dominance of an algorithm; however, this is not always possible. In this section, we propose a more nuanced way to compare algorithms using a *parametric augmentation function* $\mathcal{A}(\vec{x})$, defined as follows:

- $\vec{x}$ is a vector of user-defined parameters that are of interest to classify different troublesome cases. The elements can be, for example, $\max_{\tau_i \in \tau} U_i$, $\max_{\tau_i \in \tau} \frac{Critical_i}{T_i}$, etc. as used in the definition of capacity augmentation bounds.
- The parametric augmentation function $\mathcal{A}(\vec{x})$ represents the augmentation factor (or the utilization bound) respecting all of the parameters described in $\vec{x}$.

In this section, we perform theoretical comparisons of two priority assignment schemes for uniprocessor FP-P scheduling, using parametric augmentation functions.

- rate-monotonic (RM): If $T_i < T_j$, then task τ_i has higher-priority than task τ_j;
- slack-monotonic (SM): If $T_i - C_i < T_j - C_j$, then τ_i has higher-priority than τ_j.

In both cases, ties are broken arbitrarily.

We consider sporadic task sets in which $D_i \geq T_i$ for every task τ_i in the task set. For such a setting, Lehoczky [60] presented utilization bounds for RM scheduling, and the utilization-based scheduling k2U framework by Chen, Huang, and Liu [31] can also be applied. Instead of using the speedup factor (or the utilization bound) for comparing these two algorithms (or the schedulability tests of these two algorithms), we demonstrate why it is more meaningful to compare the algorithms and tests across a broader spectrum.

We assume that the tasks are indexed according to their priority levels, so that τ_1 has the highest priority and τ_n the lowest. We first recall polynomial-time schedulability tests for RM from the literature.

▶ **Theorem 1** (Chen, Huang, and Liu [31]). *Suppose that $D_k = fT_k$ where f is a positive integer. Task τ_k is schedulable under RM scheduling if*

$$\prod_{i=1}^{k}(1 + U_i/f) \leq (f+1)/f. \tag{4}$$

The utilization bound can be further expressed as

$$\sum_{i=1}^{k-1} U_i \leq f \ln\left(\frac{f+1}{f(1+U_k/f)}\right) \qquad \text{and} \qquad U_k \leq 1 \tag{5}$$

Proof. Equation (4) is due to Corollary 3 in [31]. The condition in (5) is obtained with similar techniques to those reported in [31]. ◀

▶ **Theorem 2** (Lehoczky [60]). *Suppose that $D_k = fT_k$ where f is a positive integer. Task τ_k is schedulable under RM scheduling if*

$$\sum_{i=1}^{k} U_i \leq \begin{cases} k\left(2^{\frac{1}{k}} - 1\right) & \text{if } f = 1 \\ f(k-1)\left(\left(\frac{f+1}{f}\right)^{\frac{1}{k-1}} - 1\right) & \text{if } f = 2, 3, \ldots \end{cases} \tag{6}$$

$$\text{When } k \to \infty, \text{ the utilization bound is } \sum_{i=1}^{k} U_i \leq f \ln((f+1)/f) \tag{7}$$

Proof. These two conditions come from Equations (3.1) and (3.2) in [60]. ◀

We next derive sufficient schedulability tests for SM.

▶ **Theorem 3.** *Suppose that $D_k = fT_k$ with $f > 0$. Task τ_k is schedulable under SM scheduling if*

$$\sum_{i=1}^{k} U_i \leq 1 \qquad \text{and} \qquad U_k + (1 - U_k + f) \sum_{i=1}^{k-1} U_i \leq f \tag{8}$$

Proof. For the rest of the proof we implicitly consider that $\sum_{i=1}^{k} U_i \leq 1$. Using the response time upper bounds given by both Bini et al. [25] and Chen et al. [32], for such a task sets, a simple schedulability test for task τ_k is to validate whether

$$\frac{C_k + \sum_{i=1}^{k-1} C_i - \sum_{i=1}^{k-1} U_i C_i}{1 - \sum_{i=1}^{k-1} U_i} \leq D_k \,.$$

By the definition of SM, we know that

$$T_i - C_i \leq T_k - C_k \Rightarrow T_i(1 - U_i) \leq T_k(1 - U_k) \Rightarrow (1 - U_i) \leq \frac{T_k}{T_i}(1 - U_k) \,.$$

Therefore, we have

$$C_i - C_i U_i = C_i(1 - U_i) \leq C_i \frac{T_k}{T_i}(1 - U_k) = T_k U_i (1 - U_k) \,.$$

As a result, the following inequality holds.

$$\frac{C_k + \sum_{i=1}^{k-1} C_i - \sum_{i=1}^{k-1} U_i C_i}{1 - \sum_{i=1}^{k-1} U_i} \leq \frac{C_k + T_k(1 - U_k) \sum_{i=1}^{k-1} U_i}{1 - \sum_{i=1}^{k-1} U_i} \,.$$

When $D_k = fT_k$, a sufficient schedulability condition for SM scheduling is given by:

$$\frac{C_k + T_k(1 - U_k) \sum_{i=1}^{k-1} U_i}{1 - \sum_{i=1}^{k-1} U_i} \leq fT_k \,.$$

By dividing both sides by T_k, this condition can be rewritten as

$$U_k + (1 - U_k) \sum_{i=1}^{k-1} U_i \leq f \left(1 - \sum_{i=1}^{k-1} U_i\right) \Rightarrow U_k + (1 - U_k + f) \sum_{i=1}^{k-1} U_i \leq f \,.$$ ◀

▶ **Theorem 4.** *Suppose that $D_k = fT_k$ with $f \geq 1$. Task τ_k is schedulable under SM scheduling if*

$$\sum_{i=1}^{k} U_i \leq \frac{f}{f+1} \tag{9}$$

Proof. This theorem is proved by finding the infimum $\sum_{i=1}^{k} U_i$ where the condition that $U_k + (1 - U_k + f)\sum_{i=1}^{k-1} U_i > f$ holds using the schedulability condition in Theorem 3. Note that the condition $\sum_{i=1}^{k} U_i \leq 1$ automatically holds. This is equivalent to the following problem:

$$\text{minimum } x + y \quad \text{s. t. } x + (1 - x + f)y = f \text{ and } 0 \leq x \leq 1,$$

where x is U_k and y is $\sum_{i=1}^{k-1} U_i$. By $x+(1-x+f)y = f$, we can express y as $(f-x)/(1-x+f)$. Therefore, $x + y$ is $x + (f - x)/(1 - x + f)$. The first order derivative of $x + y$ is

$$\frac{\partial}{\partial x}\left(x + \frac{f-x}{1-x+f}\right) = 1 - \frac{1}{(1-x+f)^2} \geq 0,$$

since $f \geq 1$ and $0 \leq x \leq 1$ in our assumption. Therefore, $x + y$ is minimized when x is 0 and y is $f/(1 + f)$ and thus the theorem follows. ◀

7.1 Comparing RM/SM Based on Traditional Utilization Bounds

Suppose that $f = \min_{\forall \tau_i}(D_i/T_i)$ for the rest of this section, where $f \geq 1$ due to the problem definition. By (7), we can conclude that the utilization bound of RM is $\left(\lfloor f \rfloor \ln \frac{\lfloor f \rfloor + 1}{\lfloor f \rfloor}\right)$. Similarly, by (9) the utilization bound of SM is $\frac{f}{f+1}$.

▶ **Lemma 5.** $\frac{x+1}{x+2} - x \ln(1 + \frac{1}{x}) \leq 0$ *for any positive integer x.*

Proof. Using Taylor series expansion, $\ln(1 + z)$ can be over-approximated by $z - \frac{z^2}{2} + \frac{z^3}{3}$ when $-1 < z < 1$. Therefore, for any $x \geq 2$, we have

$$\frac{x+1}{x+2} - x \ln(1 + \frac{1}{x}) \leq 1 - \frac{1}{x+2} - x(\frac{1}{x} - \frac{1}{2x^2} + \frac{1}{3x^3}) = \frac{-1}{x+2} + \frac{1}{2x} - \frac{1}{3x^2} < 0,$$

where the last inequality is due to $\frac{-1}{x+2} + \frac{1}{2x} \leq 0$ for any $x \geq 2$. When x is 1, the expression is $\frac{2}{3} - 2 \ln 1.5 \approx -0.144$, hence the lemma is proved. ◀

▶ **Theorem 6.** *Suppose that $\min_{\forall \tau_i}(D_i/T_i) = f$ where $f \geq 1$. The utilization bound $\left(\lfloor f \rfloor \ln \frac{\lfloor f \rfloor + 1}{\lfloor f \rfloor}\right)$ in (7) for RM dominates the utilization bound $\frac{f}{f+1}$ in (9) for SM.*

Proof. Suppose that $\lfloor f \rfloor$ is x. We know that the utilization bound $\frac{f}{f+1}$ in (9) is upper bounded by $\frac{x+1}{x+2}$, hence the theorem follows directly from Lemma 5. ◀

Since the utilization bound for RM is better than the utilization bound for SM for a given $f \geq 1$, the same holds for the speedup factor with respect to EDF-P.

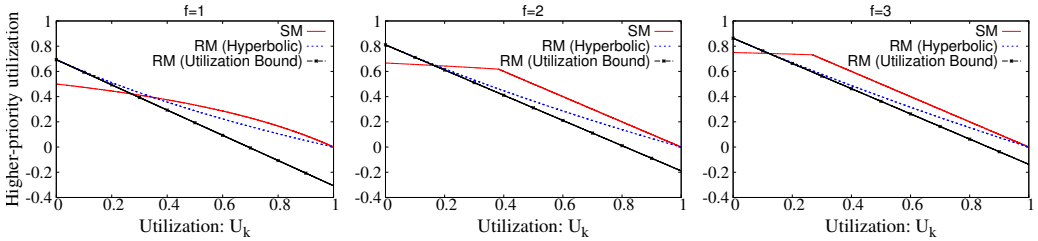


Figure 4 Theoretical comparison of SM and RM. (5) is denoted by RM (Hyperbolic), (7) is denoted by RM (Utilization Bound), and (8) is denoted by SM.

7.2 Comparing RM/SM Based on Parametric-Utilization Factors

The dominance in Theorem 6 is only due to the utilization bounds, or equivalently the augmentation factors under a given $f = \min_{\forall \tau_i}(D_i/T_i)$. If we apply more precise schedulability tests, different conclusions may be drawn.

▶ **Theorem 7.** *Suppose that* $f = \min_{\forall \tau_i}(D_i/T_i)$ *where* $f \geq 1$. *SM is a feasible scheduling algorithm for task* τ_k *if* $U_k \geq \frac{1+f-\sqrt{(1+f)^2-4}}{2}$ *and* $\sum_{i=1}^{k} U_i \leq 1$.

Proof. The test in (8) for SM checks whether both $\sum_{i=1}^{k} U_i \leq 1$ and $U_k + (1 - U_k + f)\sum_{i=1}^{k-1} U_i \leq f$ hold. The first condition constrains the utilization of the higher-priority tasks $\sum_{i=1}^{k-1} U_i$ to be at most $1 - U_k$, and the second condition constrains $\sum_{i=1}^{k-1} U_i$ to be at most $\frac{f-U_k}{1+f-U_k}$. Since $\frac{\partial(1-U_k)}{\partial U_k} = -1$ and $\frac{\partial(\frac{f-U_k}{1+f-U_k})}{\partial U_k} = \frac{-1}{(f-U_k+1)^2} \geq -1$ for $f \geq 1$ and $0 \leq U_k \leq 1$, the intersection given by $1 - U_k = \frac{f-U_k}{1+f-U_k}$ defines which of the two conditions in (8) dominates the schedulability test. Let $U^*(f)$ be such an intersection of U_k for a given f. By solving $1 - U_k = \frac{f-U_k}{1+f-U_k}$, we know that $U^*(f)$ is defined as $\frac{1+f-\sqrt{(1+f)^2-4}}{2}$. This means, $U^*(1)$ is 1, $U^*(2)$ is ≈ 0.382, $U^*(3)$ is ≈ 0.268, $U^*(4)$ is ≈ 0.209, $U^*(5)$ is ≈ 0.172, $U^*(6)$ is ≈ 0.146, $\cdots$, $U^*(10)$ is ≈ 0.092, etc.

According to the above analysis, under SM scheduling, when $U_k \geq U^*(f)$, we know that $1 - U_k \leq \frac{f-U_k}{1+f-U_k}$ and therefore the condition $\sum_{i=1}^{k} U_i \leq 1$ dominates the condition that $U_k + (1 - U_k + f)\sum_{i=1}^{k-1} U_i \leq f$. ◀

The above analysis shows that U_k plays an important role in the schedulability analysis. For ease of comparison, we assume that f is an integer in our discussions for the rest of this section. Figure 4 provides the analytical results by comparing the conditions in (5) denoted by RM (Hyperbolic), (7) denoted by RM (Utilization Bound), and (8) denoted by SM. Figure 4 also shows that $U^*(2)$ is ≈ 0.382, $U^*(3)$ is ≈ 0.268, $U^*(4)$ is ≈ 0.209; these are the values of U_k at the corner points on the line for SM. Further, the utilization bound of SM when $U_k \geq U^*(f)$ is 100%, as shown by the part of the line for SM with a 45 degree slope. (Since the y-axis measures total utilization for higher priority tasks, a line between (0,1) and (1,0) represents 100% utilization).

The parametric utilization bound shows the importance of including U_k when testing the schedulability of task τ_k, i.e., $\vec{x}$ in the parametric augmentation function should include U_k. As shown in Figure 4, when U_k is small, the schedulability tests for RM in (5) and (7) are better than the test for SM in (8). By contrast, for larger values of U_k the above test for SM is better than the above tests for RM. Hence conclusions on the analytical superiority of these tests for RM and SM can only be drawn when U_k is considered.

7.3 Comparing Schedulability Tests Based on Synthetic Workload

To demonstrate the impact of different distributions of U_k, we conducted the following evaluation for arbitrary-deadline sporadic task sets with k tasks, in which we are only interested in validating the schedulability of task τ_k. For target utilization levels $U_{sum} \geq 0.5$, we explored four different uniform distributions for U_k : (a) $[0, U_{sum}]$, (b) $[0, 0.5]$, (c) $[0, 0.3]$, (d) $[0, 0.1]$. In each case $\sum_{i=1}^{k-1} U_i = U_{sum} - U_k$. For each of the above configurations, we tested $f = 1, f = 2, f = 3, f = 4$. We generated 10000 task sets for each utilization level in each configuration. The metric used to compare performance is the *acceptance ratio* for the tests in (5), (7), and (8).

Figure 5 shows the results for the above configurations. The acceptance ratios of the tests are highly dependent on both f and the distribution of U_k, and hence the configuration used. Note that when the range of values that U_k can take is small, i.e. $[0, 0.1]$ then RM (Hyperbolic) and RM (Utilization Bound) have essentially the same performance, hence the line for RM (Hyperbolic) cannot be seen on the graphs in Figure 5(c), as it is under the line for RM (Utilization Bound).

Thanks to the parametric analysis, showing that U_k plays a significant role, the evaluation settings and configurations also consider such a parameter in the experimental setup. This gives a much more comprehensive picture of the performance of the different algorithms and tests, showing how they vary with critical parameters.

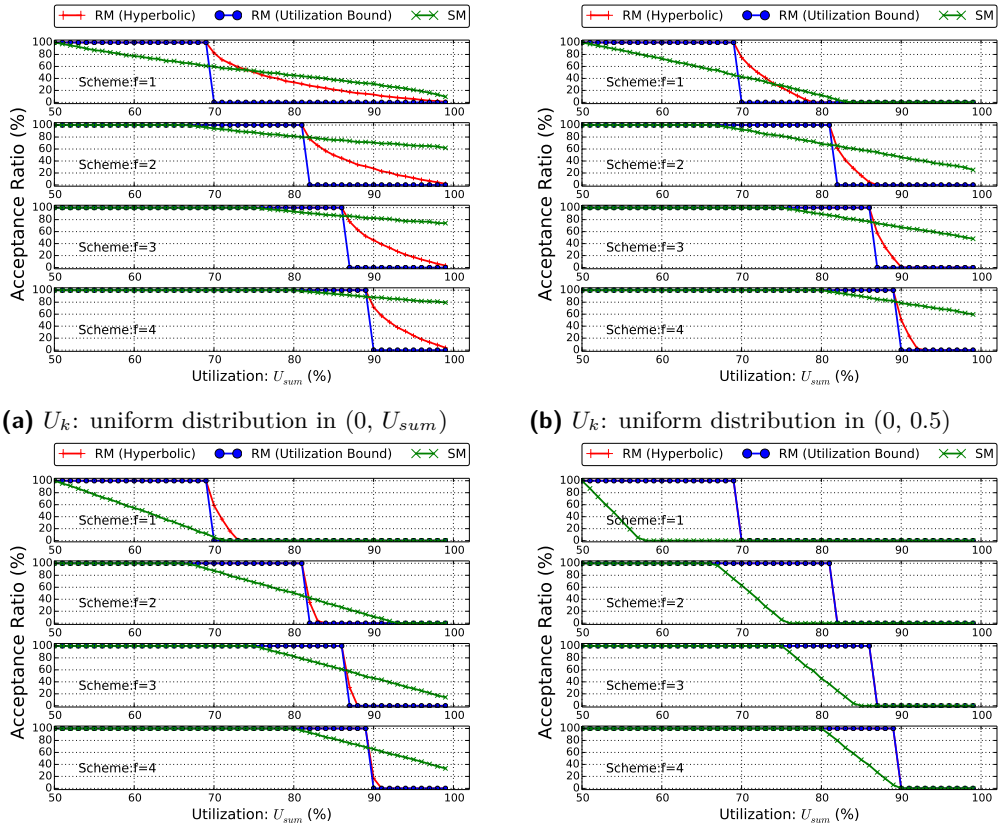
7.4 Remarks on Parametric Augmentation Functions

The concept of parametric augmentation functions can be traced back to Liu and Layland's seminal utilization bound for RM scheduling. This bound is parametric in the number of tasks: $n(2^{\frac{1}{n}} - 1) \geq \ln 2 \approx 0.693$. Similarly, work on speedup factors for DM scheduling of constrained-deadline task sets [44] explored a speedup factor upper bound that is parametric in n. Theorem A.2 in [44] made use of the hyperbolic bound to derive an expression for the speedup factor as a function of n, see also Table 5 and Figures 9 and 10 in Appendix in [44].

In some cases, speedup factors may be unbounded unless some parameter is controlled. For example, Davis et al. [35] used parametric speedup factors to compare non-preemptive uniprocessor scheduling (FP-NP and EDF-NP) against an optimal algorithm (EDF-P). In this case, speedup factors were obtained as a function of C_{max}/D_{min}, where C_{max} is the largest WCET of any task and D_{min} is the smallest deadline. Without using this parameter, the speedup factors $1 + C_{max}/D_{min}$ for EDF-NP and $2 + C_{max}/D_{min}$ for FP-NP would become unbounded, since C_{max}/D_{min} could take an arbitrarily large value. In some practical settings this value can be relatively small, highlighting the utility of such an approach. More recently, parametric augmentation functions have been implicitly used by Liu et al. [66] in the study of EDF-VD scheduling for mixed-criticality systems with degraded quality guarantees. They showed that the augmentation factor depends on two task set dependent constants, denoted by α and λ in [66], with the worst-case speedup factor reducing to 4/3.

▶ **Observation 8.** *Parametric augmentation functions can reveal more detailed and nuanced information about the actual performance of schedulability tests or scheduling algorithms across a wide range of parameter values, including practical settings. In some cases parameterized augmentation functions are essential to avoid singularities and hence unbounded results due to unrealistic combinations of parameter values.*

Finally, as shown in Section 7.3 parametric results can be helpful in the design of empirical evaluations and workload generators aimed at providing a comprehensive comparison between different scheduling algorithms and schedulability tests.



(a) U_k: uniform distribution in $(0, U_{sum})$

(b) U_k: uniform distribution in $(0, 0.5)$

(c) U_k: uniform distribution in $(0, 0.3)$

(d) U_k: uniform distribution in $(0, 0.1)$

Figure 5 Experimental comparison of SM and RM scheduling. (5) is denoted as RM (Hyperbolic), (7) is denoted as RM (Utilization Bound), and (8) is denoted as SM.

8 Summary and Conclusions

In this paper, we studied the use of speedup factors, utilization bounds, and capacity augmentation bounds. Through a series of worked examples and studies, we showed that:

- The metrics often lack the power to discriminate between the performance of different scheduling algorithms and schedulability tests even though their performance may be very different when viewed from the perspective of empirical evaluation.
- The metrics should only be considered for their negative implications, since they only provide information on performance in corner cases.
- Proving that an algorithm or test has an optimal speedup factor or bound for a class of algorithms or problems does not imply that the algorithm or test cannot be substantially improved upon. Further, an algorithm or test with a worse speedup factor or bound may perform much better in practice, and thus conclusions based on speedup factors may directly contradict those drawn from empirical evaluation.
- Identifying regions of dominance, in terms of schedulability, between scheduling algorithms or schedulability tests provides valuable information in addition to speedup factors or bounds, and empirical evaluations in terms of acceptance ratios.
- Adding enforcements tailoring the design of an algorithm or test to facilitate the derivation of a bounded speedup factor can be counterproductive; it may severely compromise performance in practical settings.

- When relative speedup factors are derived in relation to a non-optimal algorithm or class of algorithms, great care needs to be taken in interpreting the results. They can be undermined if the reference algorithm has an unbounded speedup factor with respect to optimal solutions.

We recommend parametric augmentation functions as a theoretical method capable of revealing more detailed and nuanced information about the actual performance of schedulability tests or scheduling algorithms across a wide range of parameter values, including practical settings. We illustrated this technique by deriving such functions for two scheduling algorithms and schedulability tests for uniprocessor scheduling. We also showed that in some cases, parameterized augmentation functions are essential to avoid singularities and hence unbounded results due to unrealistic combinations of parameter values.

Based on our studies of speedup factors, utilization bounds, and capacity augmentation bounds, our considered view is *handle with care*. These theoretical metrics can provide useful information; however, there are also pitfalls in their use. Problems can occur when algorithms are designed with speedup factors in mind, or conclusions are drawn taking a positive perspective solely on the basis of these results. We welcome the extra information that theoretical metrics, particularly parametric augmentation functions, can provide; however, we also recommend that any judgement on the practical utility or otherwise of scheduling algorithms or schedulability tests is backed up by a thorough performance evaluation studying practical settings.

References

1. Tarek F. Abdelzaher, Vivek Sharma, and Chenyang Lu. A utilization bound for aperiodic tasks and priority driven scheduling. *IEEE Trans. Computers*, 53(3):334–350, 2004. doi:10.1109/TC.2004.1261839.
2. Björn Andersson, Sanjoy K. Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. In *Real-Time Systems Symposium (RTSS)*, pages 193–202, 2001. doi:10.1109/REAL.2001.990610.
3. Björn Andersson and Arvind Easwaran. Provably good multiprocessor scheduling with resource sharing. *Real-Time Systems*, 46(2):153–159, 2010. doi:10.1007/s11241-010-9105-6.
4. Björn Andersson and Gurulingesh Raravi. Real-time scheduling with resource sharing on heterogeneous multiprocessors. *Real-Time Systems*, 50(2):270–314, 2014. doi:10.1007/s11241-013-9195-z.
5. Björn Andersson and Eduardo Tovar. The utilization bound of non-preemptive rate-monotonic scheduling in controller area networks is 25%. In *IEEE Fourth International Symposium on Industrial Embedded Systems – SIES*, pages 11–18, 2009. doi:10.1109/SIES.2009.5196186.
6. N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993. doi:10.1049/sej.1993.0034.
7. Neil C. Audsley. On priority assignment in fixed priority scheduling. *Information Processing Letters*, 79(1):39–44, May 2001. doi:10.1016/S0020-0190(00)00165-4.
8. Sanjoy Baruah. The federated scheduling of constrained-deadline sporadic DAG task systems. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition, DATE*, pages 1323–1328, 2015. doi:10.7873/DATE.2015.0200.

9 Sanjoy Baruah. Federated scheduling of sporadic DAG task systems. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS*, pages 179–186, 2015. doi:10.1109/IPDPS.2015.33.

10 Sanjoy Baruah. The federated scheduling of systems of conditional sporadic DAG tasks. In *Proceedings of the 15th International Conference on Embedded Software (EMSOFT)*, 2015. doi:10.1109/EMSOFT.2015.7318254.

11 Sanjoy Baruah. Schedulability analysis for a general model of mixed-criticality recurrent real-time tasks. In *IEEE Real-Time Systems Symposium, RTSS*, pages 25–34, 2016. doi:10.1109/RTSS.2016.012.

12 Sanjoy Baruah. Schedulability analysis of mixed-criticality systems with multiple frequency specifications. In *International Conference on Embedded Software, EMSOFT*, pages 24:1–24:10, 2016. doi:10.1145/2968478.2968488.

13 Sanjoy K. Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Haohan Li, Alberto Marchetti-Spaccamela, Suzanne van der Ster, and Leen Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *24th Euromicro Conference on Real-Time Systems, ECRTS*, pages 145–154, 2012. doi:10.1109/ECRTS.2012.42.

14 Sanjoy K. Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Haohan Li, Alberto Marchetti-Spaccamela, Suzanne van der Ster, and Leen Stougie. Preemptive uniprocessor scheduling of mixed-criticality sporadic task systems. *J. ACM*, 62(2):14:1–14:33, 2015. doi:10.1145/2699435.

15 Sanjoy K. Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Sebastian Stiller. Implementation of a speedup-optimal global EDF schedulability test. In *21st Euromicro Conference on Real-Time Systems, ECRTS*, pages 259–268, 2009. doi:10.1109/ECRTS.2009.31.

16 Sanjoy K. Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Sebastian Stiller. Improved multiprocessor global schedulability analysis. *Real-Time Systems*, 46(1):3–24, 2010. doi:10.1007/s11241-010-9096-3.

17 Sanjoy K. Baruah and Nathan Fisher. The partitioned multiprocessor scheduling of sporadic task systems. In *RTSS*, pages 321–329, 2005. doi:10.1109/RTSS.2005.40.

18 Sanjoy K. Baruah and Nathan Fisher. The partitioned multiprocessor scheduling of deadline-constrained sporadic task systems. *IEEE Trans. Computers*, 55(7):918–923, 2006. doi:10.1109/TC.2006.113.

19 Sanjoy K. Baruah and Nathan Wayne Fisher. The partitioned dynamic-priority scheduling of sporadic task systems. *Real-Time Syst.*, 36(3):199–226, August 2007. doi:10.1007/s11241-007-9022-5.

20 Marko Bertogna and Michele Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Real-Time Systems Symposium (RTSS)*, pages 149–160, 2007. doi:10.1109/RTSS.2007.31.

21 E. Bini, G. C. Buttazzo, and G. M. Buttazzo. A hyperbolic bound for the rate monotonic algorithm. In *Real-Time Systems, 13th Euromicro Conference on, 2001.*, pages 59–66, 2001. doi:10.1109/EMRTS.2001.934000.

22 Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005. doi:10.1007/s11241-005-0507-9.

23 Enrico Bini, Giorgio C. Buttazzo, and Giuseppe M Buttazzo. Rate monotonic analysis: the hyperbolic bound. *Computers, IEEE Transactions on*, 52(7):933–942, 2003. doi:10.1109/TC.2003.1214341.

24 Enrico Bini, Thi Huyen Chau Nguyen, Pascal Richard, and Sanjoy K. Baruah. A response-time bound in fixed-priority scheduling with arbitrary deadlines. *IEEE Trans. Computers*, 58(2):279–286, 2009. doi:10.1109/TC.2008.167.

25 Enrico Bini, Andrea Parri, and Giacomo Dossena. A quadratic-time response time upper bound with a tightness property. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 13–22, 2015. doi:10.1109/RTSS.2015.9.

26 Almut Burchard, Jörg Liebeherr, Yingfeng Oh, and Sang Hyuk Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Trans. Computers*, 44(12):1429–1442, 1995. doi:10.1109/12.477248.

27 Jian-Jia Chen. Computational complexity and speedup factors analyses for self-suspending tasks. In *Real-Time Systems Symposium (RTSS)*, pages 327–338, 2016. doi:10.1109/RTSS.2016.039.

28 Jian-Jia Chen. Federated scheduling admits no constant speedup factors for constrained-deadline dag task systems. *Real-Time Systems*, 52(6):833–838, November 2016. doi:10.1007/s11241-016-9255-2.

29 Jian-Jia Chen. Partitioned multiprocessor fixed-priority scheduling of sporadic real-time tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 251–261, 2016. doi:10.1109/ECRTS.2016.26.

30 Jian-Jia Chen, Wen-Hung Huang, and Cong Liu. Evaluate and compare two utilization-based schedulability-test frameworks for real-time systems. *CoRR*, 2015. URL: https://arxiv.org/abs/1505.02155.

31 Jian-Jia Chen, Wen-Hung Huang, and Cong Liu. k2U: A general framework from k-point effective schedulability analysis to utilization-based tests. In *Real-Time Systems Symposium (RTSS)*, pages 107–118, 2015. doi:10.1109/RTSS.2015.18.

32 Jian-Jia Chen, Wen-Hung Huang, and Cong Liu. k2Q: A quadratic-form response time and schedulability analysis framework for utilization-based analysis. In *2016 IEEE Real-Time Systems Symposium, RTSS*, pages 351–362, 2016. doi:10.1109/RTSS.2016.041.

33 Jian-Jia Chen and Cong Liu. Fixed-relative-deadline scheduling of hard real-time tasks with self-suspensions. In *Real-Time Systems Symposium (RTSS)*, pages 149–160, 2014. doi:10.1109/RTSS.2014.31.

34 R. I. Davis and A. Burns. Response time upper bounds for fixed priority real-time systems. In *Real-Time Systems Symposium, 2008*, pages 407–418, Nov 2008. doi:10.1109/RTSS.2008.18.

35 R. I. Davis, A. Thekkilakattil, O. Gettings, R. Dobrin, and S. Punnekkat. Quantifying the exact sub-optimality of non-preemptive scheduling. In *Real-Time Systems Symposium, 2015 IEEE*, pages 96–106, Dec 2015. doi:10.1109/RTSS.2015.17.

36 R. I. Davis. On the evaluation of schedulability tests for real-time scheduling algorithms. In *WATERS*, July 2016. URL: https://www-users.cs.york.ac.uk/~robdavis/papers/WATERS2016EvalSchedTests.pdf.

37 R.I. Davis, T. Rothvoß, S.K. Baruah, and A. Burns. Quantifying the sub-optimality of uniprocessor fixed priority pre-emptive scheduling for sporadic tasksets with arbitrary deadlines. In *Real-Time and Network Systems (RTNS)*, pages 23–31, 2009. URL: https://hal.inria.fr/inria-00441952.

38 Robert I. Davis and Alan Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Systems*, 47(1):1–40, 2011. doi:10.1007/s11241-010-9106-5.

39 Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35, 2011. doi:10.1145/1978802.1978814.

40 Robert I. Davis, Alan Burns, Sanjoy Baruah, Thomas Rothvoß, Laurent George, and Oliver Gettings. Exact comparison of fixed priority and EDF scheduling based on speedup factors for both pre-emptive and non-pre-emptive paradigms. *Real-Time Systems*, 51(5):566–601, 2015. doi:10.1007/s11241-015-9233-0.

41 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.

42 Robert I. Davis, Liliana Cucu-Grosjean, Marko Bertogna, and Alan Burns. A review of priority assignment in real-time systems. *Journal of systems architecture*, 65:64–82, 2016. doi:10.1016/j.sysarc.2016.04.002.

43 Robert I. Davis, Laurent George, and Pierre Courbin. Quantifying the sub-optimality of uniprocessor fixed priority non-pre-emptive scheduling. In *International Conference on Real-Time and Network Systems (RTNS'10)*, 2010. URL: https://hal.inria.fr/inria-00536363/document.

44 Robert I. Davis, Thomas Rothvoß, Sanjoy K. Baruah, and Alan Burns. Exact quantification of the sub-optimality of uniprocessor fixed priority pre-emptive scheduling. *Real-Time Systems*, 43(3):211–258, 2009. doi:10.1007/s11241-009-9079-4.

45 Michael L. Dertouzos. Control robotics: The procedural control of physical processes. In *IFIP Congress'74*, pages 807–813, 1974.

46 M. R. Garey and D. S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W. H. Freeman and Co., 1979.

47 Laurent George, Nicolas Rivierre, and Marco Spuri. Preemptive and Non-Preemptive Real-Time UniProcessor Scheduling. Research report, INRIA, 1996. URL: https://hal.inria.fr/inria-00073732.

48 Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. New response time bounds for fixed priority multiprocessor scheduling. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 387–397, 2009. doi:10.1109/RTSS.2009.11.

49 Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Fixed-priority multiprocessor scheduling with Liu and Layland's utilization bound. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 165–174, 2010. doi:10.1109/RTAS.2010.39.

50 Ching-Chih Han and Hung-Ying Tyan. A better polynomial-time schedulability test for real-time fixed-priority scheduling algorithms. In *Real-Time Systems Symposium (RTSS)*, pages 36–45, 1997. doi:10.1109/REAL.1997.641267.

51 Wen-Hung Huang and Jian-Jia Chen. Techniques for schedulability analysis in mode change systems under fixed-priority scheduling. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 176–186, 2015. doi:10.1109/RTCSA.2015.36.

52 Wen-Hung Huang, Jian-Jia Chen, Husheng Zhou, and Cong Liu. PASS: Priority assignment of real-time tasks with dynamic suspending behavior under fixed-priority scheduling. In *Proceedings of the 52nd Annual Design Automation Conference (DAC)*, pages 154:1–154:6, 2015. doi:10.1145/2744769.2744891.

53 Wen-Hung Huang, Maolin Yang, and Jian-Jia Chen. Resource-oriented partitioned scheduling in multiprocessor systems: How to partition and how to share? In *Real-Time Systems Symposium (RTSS)*, pages 111–122, 2016. doi:10.1109/RTSS.2016.020.

54 Xu Jiang, Xiang Long, Nan Guan, and Han Wan. On the decomposition-based global EDF scheduling of parallel real-time tasks. In *Real-Time Systems Symposium (RTSS)*, pages 237–246, 2016. doi:10.1109/RTSS.2016.031.

55 M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, May 1986. doi:10.1093/comjnl/29.5.390.

56 Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. *Journal of ACM*, 47(4):617–643, July 2000. doi:10.1145/347476.347479.

57 Andreas Karrenbauer and Thomas Rothvoß. A 3/2-approximation algorithm for rate-monotonic multiprocessor scheduling of implicit-deadline tasks. In *International Workshop on Approximation and Online Algorithms*, pages 166–177. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-18318-8_15.

58 Tei-Wei Kuo, Li-Pin Chang, Yu-Hua Liu, and Kwei-Jay Lin. Efficient online schedulability tests for real-time systems. *IEEE Trans. Software Eng.*, 29(8):734–751, 2003. doi:10.1109/TSE.2003.1223647.

59 Sylvain Lauzac, Rami G. Melhem, and Daniel Mossé. An efficient RMS admission control and its application to multiprocessor scheduling. In *IPPS/SPDP*, pages 511–518, 1998. doi:10.1109/IPPS.1998.669964.

60 John P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *proceedings Real-Time Systems Symposium (RTSS)*, pages 201–209, Dec 1990. doi:10.1109/REAL.1990.128748.

61 John P. Lehoczky, Lui Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium'89*, pages 166–171, 1989. doi:10.1109/REAL.1989.63567.

62 Joseph Y.-T. Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Perform. Eval.*, 2(4):237–250, 1982. doi:10.1016/0166-5316(82)90024-4.

63 Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill. Analysis of global EDF for parallel tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3–13, 2013. doi:10.1109/ECRTS.2013.12.

64 Jing Li, Jian-Jia Chen, Kunal Agrawal, Chenyang Lu, Christopher D. Gill, and Abusayeed Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *26th Euromicro Conference on Real-Time Systems, ECRTS*, pages 85–96, 2014. doi:10.1109/ECRTS.2014.23.

65 C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973. doi:10.1145/321738.321743.

66 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 *IEEE Real-Time Systems Symposium, RTSS*, pages 35–46, 2016. doi:10.1109/RTSS.2016.013.

67 Wei Liu, Jian-Jia Chen, Anas Toma, Tei-Wei Kuo, and Qingxu Deng. Computation offloading by using timing unreliable components in real-time systems. In *Design Automation Conference (DAC)*, pages 39:1–39:6, 2014. doi:10.1145/2593069.2593109.

68 C. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. In *ACM Symposium on Theory of Computing*, pages 140–149, 1997. doi:10.1145/258533.258570.

69 Georg von der Brüggen, Jian-Jia Chen, Robert I. Davis, and Wen-Hung Huang. Exact speedup factors for linear-time schedulability tests for fixed-priority preemptive and non-preemptive scheduling. *Information Processing Letters (IPL)*, 2016. doi:10.1016/j.ipl.2016.08.001.

70 Georg von der Bruggen, Jian-Jia Chen, and Wen-Hung Huang. Schedulability and optimization analysis for non-preemptive static priority scheduling based on task utilization and blocking factors. In *Euromicro Conference on Real-Time Systems, ECRTS*, pages 90–101, 2015. doi:10.1109/ECRTS.2015.16.

71 Georg von der Brüggen, Wen-Hung Huang, Jian-Jia Chen, and Cong Liu. Uniprocessor scheduling strategies for self-suspending task systems. In *International Conference on Real-Time Networks and Systems*, RTNS'16, pages 119–128, 2016. doi:10.1145/2997465.2997497.

72 Gang Yao, Giorgio Buttazzo, and Marko Bertogna. Feasibility analysis under fixed priority scheduling with fixed preemption points. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 71–80, 2010. doi:10.1109/RTCSA.2010.40.

Communication Centric Design in Complex Automotive Embedded Systems

Arne Hamann[1], Dakshina Dasari[2], Simon Kramer[3], Michael Pressler[4], and Falk Wurst[5]

1 **Corporate Research, Robert Bosch GmbH, Germany**
arne.hamann@de.bosch.com
2 **Corporate Research, Robert Bosch GmbH, Germany**
dakshina.dasari@de.bosch.com
3 **Corporate Research, Robert Bosch GmbH, Germany**
simon.kramer2@de.bosch.com
4 **Corporate Research, Robert Bosch GmbH, Germany**
michael.pressler@de.bosch.com
5 **Corporate Research, Robert Bosch GmbH, Germany**
falk.wurst@de.bosch.com

Abstract

Automotive embedded applications like the engine management system are composed of multiple functional components that are tightly coupled via numerous communication dependencies and intensive data sharing, while also having real-time requirements. In order to cope with complexity, especially in multi-core settings, various communication mechanisms are used to ensure data consistency and temporal determinism along functional cause-effect chains. However, existing timing analysis methods generally only support very basic communication models that need to be extended to handle the analysis of industry grade problems which involve more complex communication semantics. In this work, we give an overview of communication semantics used in the automotive industry and the different constraints to be considered in the design process. We also propose a method for model transformation to increase the expressiveness of current timing analysis methods enabling them to work with more complex communication semantics. We demonstrate this transformation approach for concrete implementations of two communication semantics, namely, implicit and LET communication. We discuss the impact on end-to-end latencies and communication overheads based on a full blown engine management system.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems, D.4.4 Communications Management

Keywords and phrases communication semantics, logical execution time, implicit communication, automotive, embedded systems, scheduling simulation, Amalthea

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.10

1 Introduction

Most of the innovation in the automotive area is happening on the electronics front and as a result, the number of Electronic Control Units (ECUs) in the cars has increased, with a high end car now typically having 80 to 100 ECUs. With increased functionality arises the responsibility to address the issue of handling the increased complexity in these systems. On the application front, embedded automotive applications, like the engine management system (EMS), have various application requirements (timing, data consistency, performance). Additionally, parallelizing such applications at task level to leverage the

capabilities of newer multi-core platforms is non-trivial, since all functional and non-functional properties of the software must be preserved after the transition. An EMS is inherently complex due to the existence of multiple tightly coupled functions executed by multiple tasks with various types of activations (periodic, sporadic, angle synchronous) exchanging data with different communication semantics. These communication semantics set rules on how and when data is communicated across functions to ensure data consistency and temporal determinism. While "implicit communication" proposed by AUTOSAR targets data consistency, Logical Execution Time (LET) has been proposed to solve the problem of temporal non-determinism by decoupling computation and communication, especially so when the software is deployed across multiple processors. To our knowledge, it is very uncommon to have uncontrolled/unregulated or direct communication between concurrent tasks in industry grade deployments and more complex communication semantics are employed.

Thus there is a need to factor-in the effects of these semantics in each phase of the system design (including task creation, task distribution, data and code mapping) for effectively using the platform and meeting all application objectives. In this work, we address the often overlooked problem in system design: a communication semantic aware timing analysis.

The problem is imminent since commercial tools that are currently adopted by the industry for timing analysis, like the Timing Architects Simulator [21] or SymTA/S from SymtaVision [9], currently *do not consider the effect of different communication semantics* and assume the most basic communication protocols (direct communication). Therefore the provided analysis may be unreliable and non-applicable in an industry setting where data consistency and temporal determinism must be ensured. Hence there is a need for *Model Transformation* that can convert complex communication semantics into equivalent direct communication mechanisms facilitating the use of these tools more meaningfully.

Contributions: In this paper, we highlight the importance of communication semantics when computing end-to-end latencies of effect chains in an engine management system. We provide a description of different execution models and communication semantics and their role in the entire design process. Next, we propose model transformation methods to transform LET and implicit communication into equivalent direct communication semantics. We also propose a concrete implementation of such a model transformation taking into account platform specific overheads. This transformed model can then be fed into existing scheduling analysis engines without worrying about the underlying communication semantics. Based on experiments conducted with SymTA/S, we then demonstrate the model transformation on a full blown engine management system, showing the impact on end- to-end latencies and communication overheads.

The rest of the document is organized as follows. We first describe the execution model in Section 2 followed by the hardware platform description in Section 3. In Section 4 we describe the communication semantics used in this paper. This is followed by Section 5, where we describe the model transformations along with discussions about the incurred communication overhead, backed up with experiments presented in Section 6.

2 Execution Model

In an earlier paper [14], an engine management system was characterized. Here we add more details, exploring the design space further. In general, automotive applications are organized as software components according to the AUTOSAR specification. These components consist of schedulable entities called runnables, which are grouped by their activation into tasks. We now describe the different elements in detail.

2.1 Runnables and Tasks

Runnables are atomic functions that constitute the smallest executable unit in a software component. They interact with each other over shared variables called labels. Each runnable is characterized by its code footprint, execution time (best case, average case, worst case), activation (as described later) and the set of labels it accesses. Typically runnables with the same activation (period) are grouped into tasks, with each runnable occupying a particular position in a task. Tasks are the units of scheduling by the operating system. Runnables with the same activation scheme can also be grouped into multiple tasks, e.g. for separation or distribution purposes.

2.2 Task Model: Activation Semantics

Different kinds of tasks co-exist in an EMS and they primarily differ in their activation triggers.

2.2.1 Periodic, Sporadic and Single Activation tasks

Periodic tasks are time triggered exactly every P time units and typically an EMS has different tasks with periods in $\{1, 2, 5, 10, 20, 50, 100, 200, 1000\}$ ms. Sporadic tasks are typically defined by a minimum interarrival time $minT$– that is two activations are separated by at least $minT$ time units. Single activation tasks, as the name suggests, occur only once in the system. Typically system setup and initialization tasks fall into this category.

2.2.2 Angle Synchronous tasks

These tasks are asynchronously activated at specific angles of the crankshaft and their periodicity is determined by the speed of the crankshaft, generally represented in rotations per minute, rpm, and the number of cylinders, $nCyl$ in the engine. The period P is then given by $P = 120/(rpm * nCyl)$. As a result, these tasks are also called adaptive rate tasks since their rate changes with the speed of rotation. As seen above, these tasks are activated more frequently with increase in rotation speed.

2.2.3 Chained Tasks

Unlike other tasks that are triggered independently, these tasks are typically triggered by a predecessor task. Here the predecessor task is terminated and the thread of control is handed over to the newly activated task. Tasks can also be chained across cores.

2.2.4 Interrupts Service Routines (ISR)

Interrupt Service Routines are functions that are directly triggered by a hardware event. They usually execute their functionality directly without any task switching overhead, and thus respond faster to events.

2.3 Scheduling Model

Tasks in the EMS are scheduled using fixed priority preemptive scheduling like rate monotonic scheduling (RMS) [15]. With RMS, the priority of the tasks depends on the period (rate) – shorter the period, higher is the priority. In general, only higher priority tasks may preempt lower priority tasks. Furthermore, tasks are scheduled either in a fully preemptive

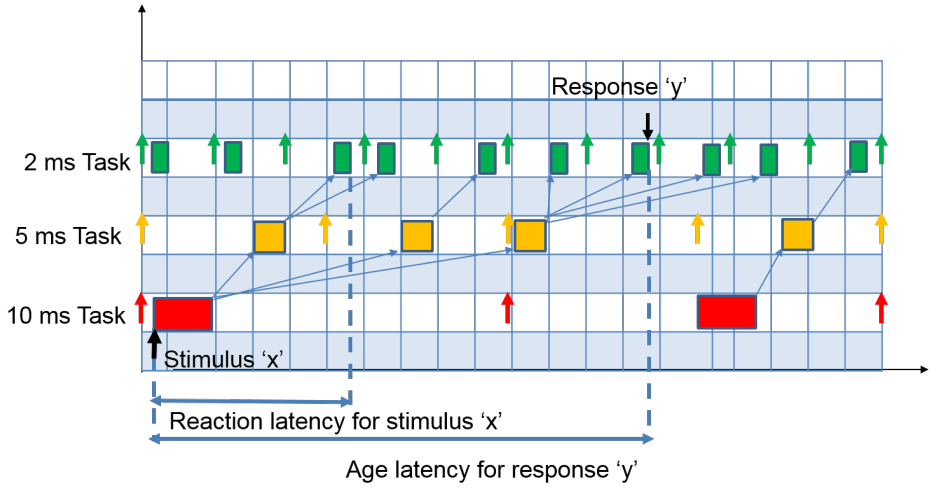


Figure 1 Age and Reaction Latency.

or cooperative manner. Tasks that participate in preemptive scheduling can preempt every other task at any time. However, tasks that are scheduled cooperatively may be preempted by higher priority cooperative tasks only at specified schedule points which correspond to runnable boundaries within a task. This ensures data consistency at the granularity of the runnables. Since preemption can occur at runnable boundaries only, it may lead to cases in which higher priority tasks are blocked by lower priority tasks still executing the current runnable. Note that preemptive tasks may preempt cooperative tasks at any point.

2.4 Event Chains

An event chain, also called effect chain or signal flow, is an abstraction of a data flow through a system. Typically an EMS has multiple event chains wherein data is sensed by the sensor nodes, passed on to control functions which act on this data and finally the output is used to configure the actuation functions to perform the desired action. Thus these event chains are a sequence of successive stimulus-response segments, where the response of one segment is the stimulus of the next segment. Each of these event chains is associated with an end-to-end latency requirement which is specified via one of the two delay semantics: an *Age* or *Reaction* latency constraint (see Figure 1). Event chains may be based on arbitrary events that occur in a system. In this paper we focus on event chains that are based on runnables as well as on start and termination events. With this, runnables could be part of different tasks and hence a stimulus and response segment will be realized by runnables that may belong to two different tasks.

2.4.1 Reaction Latency Constraint

Reaction latency denotes the time between the occurrence of the response to a specific stimulus. In other words, it denotes the time lapsed between a specific (sensor) input value or signal to a corresponding (actuator) output value, specifying how long a value or signal needs from one end to the other. A reaction latency constraint of k time units to a particular stimulus implies that the response should occur no later than k time units after that stimulus. An example from the chassis domain is the time from the brake pedal is pressed until the

brakes are activated [6]. In this work, we will deal with reaction latency constraints.

2.4.2 Age Latency Constraint

For the age latency, the freshness of the data producing the response is important and hence the focus is from the response perspective rather than from the stimulus perspective. In other words, this is the maximum time a specific output (actuator) value is available from a corresponding (sensor) input value or signal. This also equals the validity of a specific value or signal before a new value arrives. A (max) age constraint of "k" time units for a cause-effect chain mandates that for an occurrence of a response event, the corresponding input data is not older than "k" time units [6]. When for example an actuator value is periodically updated, it is of importance that the corresponding input values are not too old.

2.4.3 Multi-Rate Event Chains

The complexity of computing end to end delays (age or reaction) in event chains arises due to the fact that an event chain may consist of tasks executing at different rates or periods. Such multi-rate event chains thus often suffer from effects of i) undersampling in which the producer produces data at a higher rate than the consumer task can process it or ii) oversampling in which the consumer is activated at a higher rate than the producer. With these effects, there is a problem of either produced results being lost as they are not consumed as frequently, or duplicate inputs being acted upon by different consumer task activations. Besides, the effects of jitter during the sensing, scheduling, control and actuation, together with the possible presence of multiple clocks in the system, add to uncertainty in the delay calculations.

3 Platform Model

A cost model of the hardware platform is necessary as it is required to compute the overheads of data accesses while employing different communication semantics. In this work, we consider the AURIX [4] platform as seen in Figure 2, which is widely used for deploying automotive embedded real-time systems. The 32-bit platform consists of three cores, each equipped with a local data and code scratchpad memory. Additionally the platform provides a persistent global flash memory which is used to store code and persistent data, and a global dynamic RAM for storing non-persistent data. The memory is distributed and each core can access all scratchpads and the global memory via a crossbar interconnection.

The AURIX platform has a write buffer to decouple memory write operations from the CPU instruction execution. When the buffer is full, the priority of the buffer is raised, as a consequence of which, the buffer is flushed. Due to this mechanism, additional write access latencies are negligible and of a non-blocking nature for automotive control software categorized in [14]. Hence, we ignore additional write access latencies in the overhead calculation in this work.

Contention arises when multiple cores try to access cross-core/flash data or when high priority DMA blocks the memory. A precise analysis must ideally calculate the exact timing of concurrent accesses but dynamic hardware effects leading to execution jitters make a realistic formal calculation of these overheads nontrivial, and these aspects have been addressed to some extent in [13, 16]. Therefore, contention modelled by α is an additional overhead in the calculations which is highly dependent on the actual software and deployment configuration.

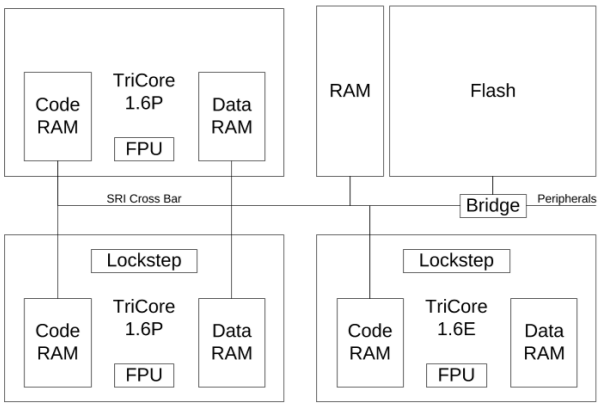


Figure 2 Simplified AURIX Architecture [20].

In this paper, to keep the focus on the semantics, we do not consider the overheads in the experimental section.

To compute the remaining memory access latency, we distinguish between read and write accesses. Data read from local scratchpads is directly available and therefore load instructions need one execution cycle[1]. In the following, we calculate the read access latency A_r, by $A_r(x) = CC_x + 1 + \alpha$, where CC_x represents the CPU access latency to memory x, and the additional cycle accounts for the execution of the load instruction. As stated above the contention α is ignored in our experiments. The CPU access latency CC_x to a local scratchpad costs zero cycles and eight cycles for a remote RAM access. The write access A_w always needs one cycle. We only consider the store instruction execution time and ignore the interconnect latencies due to the write buffer mechanism. Additionally, we assume that all labels in the system under analysis fit into the word size of the AURIX architecture (32 bits), and thus can always be fetched with a single access. This is the case for the EMS considered in this paper.

4 Communication Semantic Centric Design

As mentioned earlier, the importance of data consistency is integral to the engine management system due to the high coupling of combustion control functions [17]. Multiple functions are involved in computing the dynamic process starting from air intake to the end of the exhaust system. Highly dynamic physical values (like engine speed, manifold pressure, air temperature) are constantly exchanged at high rates between different functions. Since many sensed values are required by multiple functions, race conditions are quite probable. This problem is aggravated by the fact that the involved functions are executed by tasks allocated to different cores communicating with each other using shared data that need to be protected by mechanisms that guarantee mutually exclusive access with bounded worst-case blocking time. To cope with this problem, platform mechanisms enforcing data consistency and temporal determinism are employed in industrial systems to constructively ensure functional correctness. The definition of the term data consistency includes two different dimensions. The first dimension is the consistency in value, meaning that the value of a variable/message is not affected by action outside the current execution context. The other dimension is

[1] We ignore micro-architecture effects, e.g. result latencies, here, which could lead to additional delays.

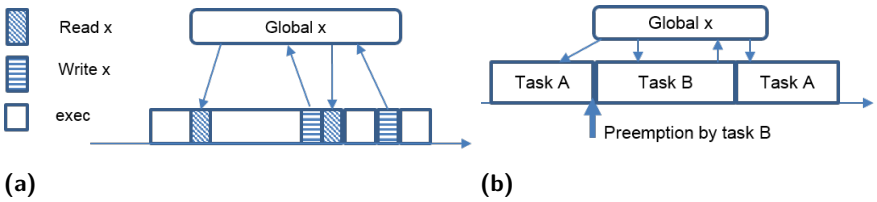


Figure 3 (a) Direct access: task performs read and writes on a global variable during its execution. (b) Example showing how task A uses 2 different values at different points in execution.

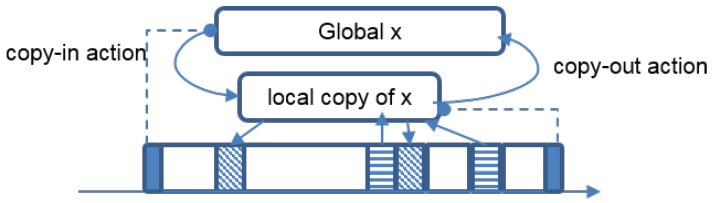


Figure 4 Implicit communication: CopyIn and CopyOut runnables read and write copies at beginning and end of the task.

consistency of a variable with other variables. This means that multiple variables are only valid if all (within the consistency scope) are stemming from the correct (same) point in time. In this work we deal with the first dimension of the data consistency problem by employing inter-task and intra-task communication semantics as described below.

4.1 Inter-Task Communication Semantics

4.1.1 Explicit or Direct Communication

This semantic, often also called direct access, allows unrestricted access to shared variables (labels) across tasks. As seen in Figure 3a, the task works on the global variable of the label. This may work for labels which are strictly read-only, but with labels which may be overwritten, data inconsistency may result. Interleaving of task activations will result in different values of the data. In a single core setting, a simple scenario is one in which a preempting task changes a shared value and so the preempted tasks works on two different values at two different points in time, leading to inconsistencies as seen in Figure 3b.

4.1.2 Implicit Communication

This semantic, proposed by AUTOSAR, is primarily focused towards maintaining data consistency to avoid the pitfalls of explicit communication. It essentially follows a read-execute-write paradigm – Implicit communication mandates that the task always makes local copies of the shared data it needs at the beginning of its *execution*, works on the local copies and writes the data back at the end of its execution (see Figure 4). This ensures that the task works on the same copy throughout its execution, and also preserves consistent state of the data.

From the access latency perspective, all the variables that are read during task execution will have to be pre-fetched into local memory from its source and then the task may execute by referencing readily available local copies (see Figure 5), hence not incurring the cost of remote accesses multiple times.

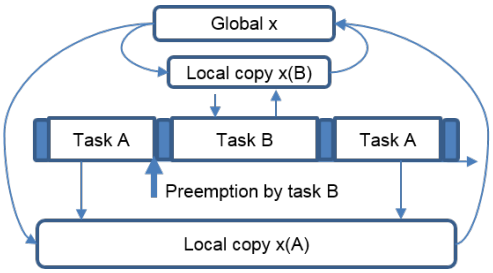


Figure 5 Implicit communication: each task works on its local copies.

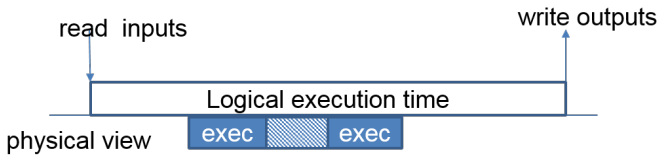


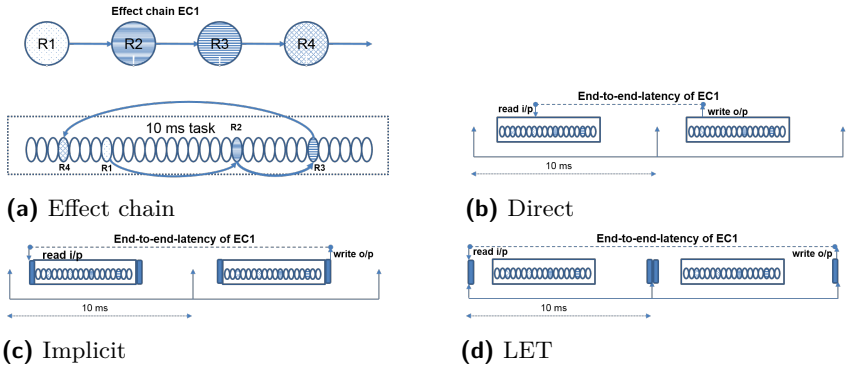
Figure 6 LET: The observed output is independent of the time a task executes in its LET interval.

The access latency in case of implicit communication is therefore dependent on multiple factors including, the cost of access to remote and local memory, number of accesses to the label during one execution to the local memory, and the period/activation rate of the task. However on the memory storage front, more local storage is required, since for every task which accesses the label, an extra local copy is required.

4.1.3 Logical Execution Time Model

Logical Execution Time (LET) was introduced with the time-triggered programming language Giotto [11]. It is a real-time programming concept which ensures temporal determinism by decoupling computation and communication. The problem with an unconstrained communication method, i.e, allowing tasks to read and write arbitrarily is non-determinism due to "execution jitter". The result is highly dependent on possible interferences of other tasks executing within a tasks activation interval (say from its release to the end of its period). The effects of this jitter becomes more prominent in event chains, leading to large variations in end-to-end delays. The LET model is robust against these jitters by enforcing strict communication rules. With the LET model, tasks always read data at the beginning of the activation interval and write data at the end of the activation interval (see Figure 6). As with implicit communication, LET requires that a local copy is available for each variable accessed by a task. Using LET, the observable temporal behavior of a task is independent from its physical execution. That is irrespective of the exact time a task executes within its execution interval, the result will be always available only at the end of its activation interval. LET also ensures portability, i.e, the same behavior of the tasks when migrated to another hardware (core), integrability on addition of newer software and interoperability, which is verified by deterministic communication.

With LET, the end-to-end latencies in case of synchronous stimuli is always equal the sum of the periods of the tasks involved in the chain. However, with asynchronous stimuli it may happen that each task in the effect chain executes as early as possible in its activation window but the data arrives just after it begins execution (meaning it is operating on an older value of the data). Thus the newer data is consumed only one time period later. The



(a) Effect chain **(b)** Direct

(c) Implicit **(d)** LET

Figure 7 The effect chain spans over two task activations due to backward communication.

same scenario could occur with every pair of tasks in the chain. Eventually, the worst case latency in the case of such asynchronous arrivals is *twice the sum of the periods of all the tasks in the chain.*

LET thus leads to longer latencies in event chains. But on the other hand, with LET, there is no need for complex synchronization mechanisms to handle race conditions or priority inversions, given its well-defined semantics.

4.2 Intra-task communication

Runnables within a task may communicate with each other in two different ways. With forward communication, the producer runnable completes before the consumer runnable and hence there is no delay in getting the latest data by the consumer and communication is therefore fast.

With backward communication however, the consumer runnable executes before the producer, and thus there is a delay of one period in this case to receive the latest data.

4.3 Event chains and communication semantics

Figure 7 summarizes how the different communication mechanisms affect the resulting latency of a simple event chain in which all runnables are mapped to one task, but there is a backward communication, since the consumer is positioned before the producer in the task, as shown in Figure 7a, and has a different order in the event chain.

5 Model transformation

Currently existing timing analysis tools like SymTA/S [9] typically do not consider the implementation overheads of the underlying communication semantics or are only equipped to handle direct communication. As a result, these tools cannot be reliably used in actual industry-grade evaluations where typically complex semantics like *implicit* or *LET* communication are deployed. Additionally, since each design phase like task creation, task distribution, data/code mapping and computation of end-to-end latencies of event chains are heavily influenced by the underlying semantics, the implementation-specific details cannot be ignored. We transform the models as shown below – these models can then be analyzed using state-of-the-art timing analysis tools to effectively compute the end-to-end latencies along event chains by considering the implementation specific overheads for copying data.

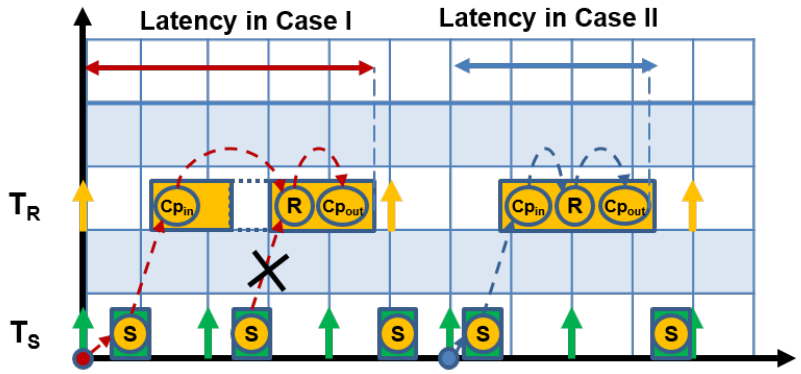


Figure 8 Example event chain exhibiting non deterministic latencies with *implicit communication semantics*. Case I: shows that newer data from the sending runnable might be discarded due to the copy-in mechanism. Case II: shows shorter latency in a different execution scenario.

5.1 Transformation of event chains to model latency effects

In this section we describe how event chains consisting of runnables communicating according to *implicit* and *LET communication* semantics can be transformed to equivalent event chains in terms of end-to-end latencies assuming direct communication between all runnables. The transformations are explained on *event chain segment* basis[2], meaning that they can be mixed along event chains to address systems with heterogeneous communication semantics, which is usually the case in real systems.

5.1.1 End-to-end latency with implicit communication semantic

As mentioned earlier, *implicit communication* mandates that the task makes local copies of the shared data it needs at the beginning of its execution, works on these local copies and writes the data back at the end of its execution. In order to realize this aspect of *implicit communication*, auxiliary Copy-in Cp_{in} and Copy-out Cp_{out} runnables are added at the beginning and the end of each task, respectively. The Cp_{in} runnable prefetches all the labels that are needed by the task during execution, into local memory. Then, during task execution, only these local copies are accessed. Similarly, all the labels that are written by a task are written into local memory and then, after task execution are written back to the original labels by Cp_{out}. The labels that a task needs during execution are extracted by program analysis, which is possible in such embedded programs, where pointers and other complex programming artifacts are not encouraged.

Figure 8 visualizes how event chains need to be transformed to take *implicit communication* semantics into account.

Each event chain segment $\{S \rightarrow R\}$ where the stimulus runnable S and response runnable R, *belong to different tasks*, needs to be replaced by three new consecutive event chain segments with stimulus and responses pairs and the modified event segments are: $\{\{S \rightarrow Cp_{out}(S)\}\{Cp_{out}(S) \rightarrow Cp_{in}(R)\}\{Cp_{in}(R) \rightarrow R\}\}$, where the notation $Cp_{out}(S)$ represents the copy-out runnable of task containing S. For completeness, we similarly model the initial copy-in to the first runnable in the event chain and the copy-out of the last runnable of the event chain.

[2] We assume that event chains are defined on runnable level.

As can be observed in Figure 8, Case I, the copy-in operations potentially increase the end-to-end latency along event chains. More precisely, this is the case when the sending runnable S updates the label after the copy-in procedure of the task containing the receiving runnable R has taken place and before R is executed. In the given example, this leads to the situation where fresher data of T_S is discarded, which would not be the case with direct communication. Obviously, such situations leading to increased latency due to *implicit communication* semantics do not occur systematically as is visualized in Case II. However, similar data race effects occur due to the copy-out mechanisms.

Please note that it is assumed that there is *exactly* one task responsible for writing each label. In case multiple tasks with different priority levels perform write accesses to the same label, there can be data races when all of them try to perform a copy-out to the master copy of the same variable. Lock mechanisms for exclusive update rights must then be provided for maintaining consistency. Also task priorities that prevent data race conditions can be exploited to optimize the number of created label copies.

5.1.2 End-to-end latency with LET communication semantic

Modeling LET communication is similar to modeling *implicit communication*, in that copy in and copy-out tasks must be augmented to the model. However, positioning these auxiliary tasks correctly is important to achieve the expected behaviour. The activation rates of the pair of communicating tasks (relatively harmonic or non-harmonic) and their priorities will have an influence on the resulting end-to-end latencies. Note that with LET, tasks can communicate data *only* at the beginning and the end of their activation interval. Since we consider a system that employs fixed priority preemptive processing, we prioritize the copy operations by elevating their priorities to the highest in the system. Hence the copy-out task is given the highest priority while the next highest priority is assigned to copy-in tasks.

In this paper, *LET communication* semantic is applied at task level, i.e. deterministic communication is ensured at task activation boundaries. For this reason, event chain segments with the stimulus runnable S and the response runnable R need to be transformed if, either, they belong to different tasks, or if they belong to the same task and exchange data with backward communication (see Section 4.2).

Figure 9 visualizes the transformation, where T_S and T_R denote the tasks containing the runnables S and R, respectively. We transform the segment $\{T_S \rightarrow T_R\}$ to $\{T_S \rightarrow Cp_{out}(T_S)\}, \{Cp_{out}(T_S) \rightarrow Cp_{in}(T_R)\}, \{Cp_{in}(T_R) \rightarrow T_R\}\}$. For completeness, we similarly model event chain segments modelling the initial copy-in to the first runnable and the final copy-out from the last runnable. Please note that these tasks are only needed logically to mimic *LET communication* semantics for analysis engines that are based on direct communication. Therefore, the execution time attributed to those copy tasks is equal to zero. How the overhead induced by these copy operations can be taken into account is discussed later.

5.2 Calculating the communication overhead

In this section we explain how to calculate the communication overhead of the system with respect to the different data synchronization mechanisms. We use the cost model of the widely used AURIX [4] platform explained in Section 3. Obviously, there are different implementation alternatives on software level for the mentioned communication mechanisms. Depending on the chosen alternative, runnables for modeling the overhead need to be added

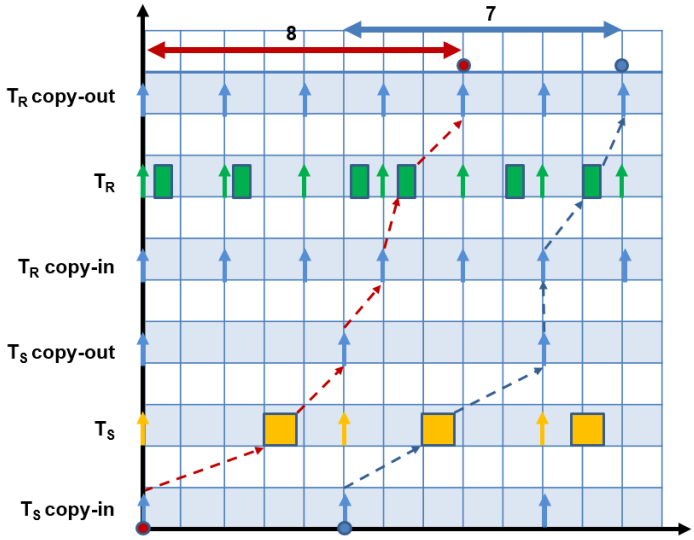


Figure 9 *LET communication* semantic is achieved by 1) adding highest priority copy-out and second highest priority copy-in tasks with 0 execution time for each task, and 2) placing these in event chains between communicating tasks.

to different tasks. We assume that the execution time for each runnable contains code-fetching overhead, but excludes any load/store execution times.

We assume that constants are mapped to the global RAM and that accesses are never cached. Variables have exactly one writer but can have multiple readers and are mapped to the local scratchpad of the core hosting the writer (task or runnable). We also assume that labels are not persisted in registers when multiple accesses to labels in local scratchpads are made, and therefore a load instruction is necessary.

The communication cost per task is calculated as described in Section 3 for each read label access and considers one cycle for each write accesses as described earlier. Let $\eta_i(l)$ be the number of accesses of the label l from task τ_i during one execution. Let π_l denote the memory where label l is mapped. Then,

$$\begin{aligned} C_{com} &= \sum_{l \in R_i} \eta_i(l) * A_r(\pi_l) + \sum_{k \in W_i} \eta_i(k) * A_w(\pi_k) \\ &= \sum_{l \in R_i} \eta_i(l) * A_r(\pi_l) + \sum_{k \in W_i} \eta_i(k) \end{aligned} \tag{1}$$

where R_i and W_i denotes the set of labels read and written by task τ_i and $A_r(\pi_l)$ describes the time for a read access from memory π_l and correspondingly $A_w(\pi_l) = 1$ for any π_l. Eventually C_{com} is added to the effective execution time of the task. Note that for the experiments, we similarly apply the above formula at runnable level to compute their gross execution time. For direct communication, the overall communication cost is given by Equation 1 and so $C_{dc} = C_{com}$. However, additional costs are incurred for the implicit case as seen below.

5.2.1 Communication cost for implicit communication

As explained in Section 5.1.1, two runnables are added to each task for realizing *implicit communication*: Cp_{in} for copying all labels read inside the task into local copies, and Cp_{out} for writing back all local copies of labels written inside the task.

The cost for Cp_{out} depends on the number of written labels, since the access latency overhead is minimal due to the write buffers of the considered AURIX platform (refer Section 3).

To create copies of labels, a read operation followed by a write operation is executed to load a label into the local scratchpad. The read operation loads the label into the CPU register and the write operation stores it into the local scratchpad. The runtime C_{in} for the Cp_{in} runnables is calculated by

$$C_{in} = \sum_{l \in R_i} (A_r(\pi_l) + A_w(x)) = \sum_{l \in R_i} A_r(\pi_l) + |R_i| \tag{2}$$

where again π_l describes the memory location of the label l, which could be mapped locally or remotely. $A_w(x)$ denotes the cost of writing a label to the local scratchpad memory x, which like any other write access always costs 1 cycle. Note that we do not include the frequency of accesses here since a copy is done once for each task that accesses the label.

The cost C_{out} for the write-back of the label copies in Cp_{out} is calculated by the below equation where y denotes the target memory location.

$$C_{out} = \sum_{l \in W_i} (A_r(\pi_l) + A_w(y)) = |W_i| * 2\,. \tag{3}$$

Due to the fact, that the copies are located in the local memory and so $A_r(\pi_l) = 1$, the overhead for the copy is always two cycles per written label.

In addition to the copy runnables, we need to consider the access latencies within the task to compute the overall communication cost $C_{implicit}$ for implicit communication. The formula to calculate the communication cost is given by Equation 1, but considering that the labels to be accessed are now *located in the local scratchpad incurring a single cycle access latency*. Only the constants remain in the global RAM. In the following equation, C_{com} includes the accesses from the runnables of the task whereas C_{in} and C_{out} contain the costs of the auxiliary copy runnables, leading to:

$$C_{implicit} = C_{com} + C_{in} + C_{out}\,. \tag{4}$$

5.2.2 Communication cost for LET communication

In contrast to implicit communication, with LET, communication happens at period boundaries. Thus we add additional runnables for copying the required data for each task into the corresponding local scratchpad memories as with implicit communication, but we trigger these copy operations at different time points. In our proposed implementation, this copying is facilitated via highest priority interrupts.

The first step of the model transformation is to add highest priority interrupts for each periodic task in the system. These interrupts are activated with the same period as the tasks they correspond to.

In the second step of the model transformation, runnables performing the necessary copy operations to realize *LET communication* have to be added to the system *for each pair of communicating tasks*. In the following, the two communicating tasks are denoted by T_A and T_B, whereas their periods are denoted by P_A and P_B.

In order to describe the timing of the copy operations, the notions of *prescale* and *offset* that can be associated to runnables are required. The *prescale* describes the activation frequency of the runnable in relation to the activation frequency of the task it is mapped to. For instance, a *prescale* of 2 denotes that the runnable is executed only every second

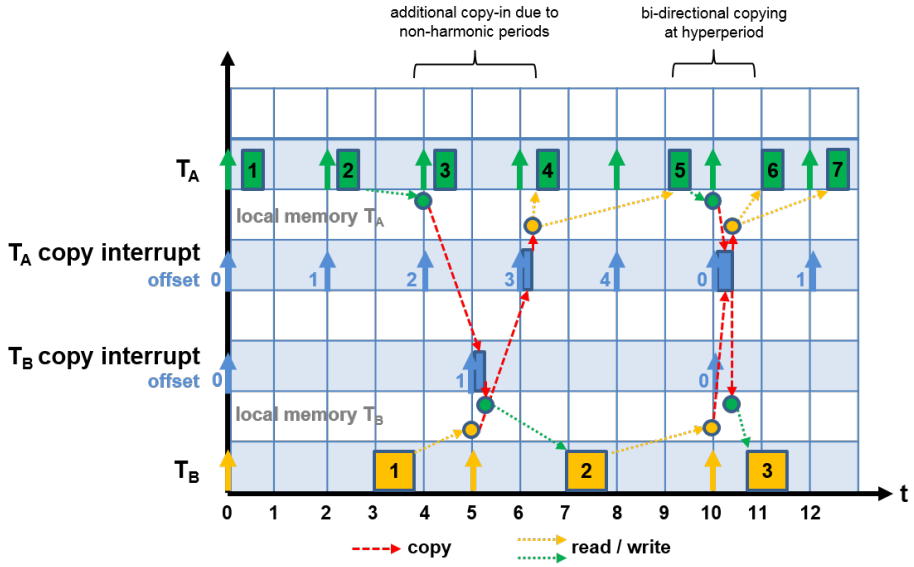


Figure 10 Copy operations (realized by high-priority interrupts) that need to be performed between two tasks with non-harmonic periods in order to realize *LET communication*.

time the parent task is activated. In relation to that, the *offset* describes the initial shift of the first execution starting to count from 0. For instance, a *prescale* of 3 and an *offset* of 1 denote that the runnable is executed at the activations $2, 5, 8, etc.$ of the parent task.

Figure 10 shows the necessary copy operation for the more general case where T_A and T_B have non-harmonic periods. In this case $P_A = 2$ and $P_B = 5$.

5.2.2.1 Costs for label accesses

Firstly, as in implicit communication, the execution times of both T_A and T_B are increased by C_{com}, as in Equation 1, to factor-in the label access costs. Note that for the calculation of C_{com}, it has to be taken into account *that all tasks operate on local copies*.

5.2.2.2 Bi-directional copy at hyperperiods

Then, a bi-directional update of the local label copies needs to be performed at the end of the hyperperiod of the two communicating tasks. This is necessary since according to the *LET* semantic, new written data becomes visible for all reading tasks at that point. These bi-directional copy operations are performed by a single runnable that is mapped to the copy interrupt corresponding to the faster task. Given that $P_A < P_B$, the following copy operations need to be performed:

- The local copies of labels written by T_A and read by T_B need to be updated in the local scratchpad of the core that T_B is mapped to. The cost for these operations is denoted by C_{AB}.
- The local copies of labels written by T_B and read by T_A need to be updated in the local scratchpad of the core that T_A is mapped to. The cost for these operations is denoted by C_{BA}.

An important point in the following cost computations is that labels are mapped to the local memory of the core hosting the task that writes the label. Let S_{AB} be the set of labels

written by task T_A and read by T_B. Likewise S_{BA} denotes the set of labels written by task T_B and read by task T_A. Then given that $A_w(x) = 1$ for any destination memory x, and π_l is the local memory of the core where T_A is mapped to, we have:

$$C_{AB} = \sum_{l \in S_{AB}} (A_r(\pi_l) + A_w(x)) = 2 * |S_{AB}| . \tag{5}$$

Similarly given π_k is the local memory of the core where T_B is mapped to and y is the local memory of the core where T_A is mapped to, we have:

$$C_{BA} = \sum_{k \in S_{BA}} (A_r(\pi_k) + A_w(y)) = \sum_{k \in S_{BA}} A_r(\pi_k) + |S_{BA}| . \tag{6}$$

5.2.2.3 Handling non-harmonic tasks

In the case of harmonic task periods, the copy operations at the hyperperiod are sufficient to realize *LET communication* semantics. However, in case of non-harmonic task periods additional copy operations need to be performed. Figure 10 visualizes the extra operations necessary for *LET communication* between the tasks T_A and T_B with periods $P_A = 2$ and $P_B = 5$, respectively.

As observed, the results of T_A's second execution become visible at time instant 4, and, thus, before the second activation of T_B at time instant 5. For this reason, the local copies of the labels written by T_A that are read by T_B must be updated in the local memory of the core T_B is mapped to *before* its second activation is executed. In the implementation discussed in this paper, the necessary copy operations are performed by a runnable that is mapped to the copy interrupt corresponding to T_B with *prescale* $= 2$ and *offset* $= 1$. In the reverse communication direction, the results of T_B's first execution that become visible at time instant 5 need to be made available for T_A's fourth activation at time instant 6. This is achieved by adding a runnable to the copy interrupt corresponding to T_A with *prescale* $= 5$ and *offset* $= 3$.

Algorithm 1 Calculate additional copy points for tasks with non-harmonic periods

Input: Periods P_A and P_B of two tasks T_A and T_B involved in *LET communication*
Output: List of copy points (prescale p and offset o) at which T_A needs to copy-in new available data from T_B

1: $CopyPoints \leftarrow \{\}$
2: $bCur \leftarrow P_B$
3: **while** $bCur < lcm(P_A, P_B)$ **do**
4: $\quad aCur \leftarrow P_A \times \left\lceil \frac{bCur}{P_A} \right\rceil$
5: $\quad bCur \leftarrow P_B \times \left\lceil \frac{aCur}{P_B} \right\rceil$
6: $\quad$ **if** $aCur \neq lcm(P_A, P_B)$ **then**
7: $\quad\quad CopyPoints \leftarrow \left(\mathrm{p} = \frac{lcm(P_A,P_B)}{P_A}, \mathrm{o} = \frac{aCur}{P_A}\right)$
8: $\quad$ **end if**
9: **end while**
10: **return** $CopyPoints$

Algorithm 1 describes how these uni-directional additional copy points can be calculated in the general case. Given the periods of two communicating tasks, the algorithm returns a list of copy points (tuples of prescale and offset) at which the first specified task needs to fetch the results of the second task in the above explained manner.

The communication costs for each of these copy points is the calculated by Eq. 5 and Eq. 6. The calculations must be done for every pair of periodic tasks that exchange data. The communication costs for one execution of the task differ on whether no updates are needed, a bi-directional hyperperiod data exchange is performed, or non-harmonic task pairs perform intermediate copy updates.

6 Experimental Results

6.1 Experimental Setup

We base our experiments on the model of an engine management system provided in the context of the industrial challenge of the WATERS 2017 workshop [2], an extension of that provided in [14, 1]. The earlier model is augmented to specify the frequency of label accesses from each runnable. The platform consists of 4 cores, running at 200 MHz and executing a set of periodic tasks, an angle synchronous task as well as interrupt service routines (ISR). Please note that the model transformations proposed in this paper are only applied to the periodic tasks. The application consists of 1250 runnables grouped into 21 tasks/ISRs which communicate via 10000 labels. Since the focus is on modeling the communication semantics, we assume that the mapping of labels and tasks is already provided. Constant calibration data, i.e. labels that are only read but never written, is mapped to the global RAM. Variables, i.e. labels that are written by a single task and potentially read by multiple tasks, are mapped to the local memory of the core hosting the writer task. We further assume that the underlying platform does not support data caching for the data mapped into the global RAM. We assume synchronous releases of all periodic tasks for these experiments, whereas the angle synchronous task and all ISRs are asynchronously released. The calculated end-to-end latency does not contain extra delays for sampling effects with external stimuli. In case of implicit communication, the beginning of the event chain is the point in time when the copy-in runnable of the task containing the first runnable of the effect chain is executed.

We use the AMALTHEA [3] meta-model for describing the engine management system. AMALTHEA provides model elements to express event chains, different tasks models, different constraints and the hardware platform. We implemented the above proposed transformations to realize the direct, implicit, and LET communication semantics. This transformed AMALTHEA model is then fed into the SymTA/S tool, wherein the execution behavior is simulated over multiple runs to generate the access latency distributions we present.

6.2 Comparison of the end-to-end latencies

In this experiment, we compare the end-to-end latencies of effect chains across the three communication semantics. For the experimental results we highlight the applicability on two important types of chains for evaluation.

6.2.1 Single-rate effect chain with backward communication

We analyze effect chain $EC1$, composed of four runnables, all positioned in a single task with activation of 10ms similar to the example in Figure 7a. There is backward communication in this scenario, since the relative order of the producer and consumer in the effect chain is different from the positions in the task.

The different latencies observed for around 3500 runs are shown in Figure 11. As seen, the latency for direct communication is the least, since the available input is read immediately

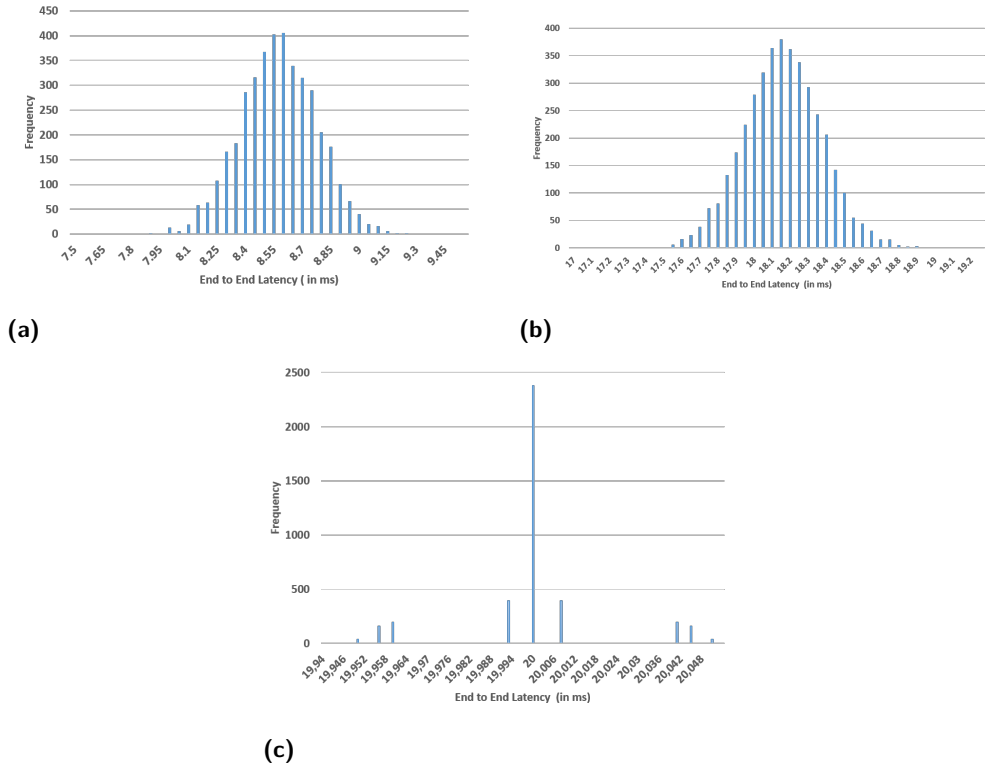


Figure 11 Comparison of end-to-end latency distribution between direct, implicit and LET communication.

by the first runnable of the effect chain and the produced output is immediately reflected across the event chain segments, therefore reducing the effective end-to-end latency. However with implicit and LET semantics, as seen in Fig. 7c and Fig 7d, there is, as expected, an increase in the end-to-end latencies in the chain. As described earlier the output using implicit communication is globally available only at the end of the task execution, while with LET at the end of the activation interval. Note that with LET irrespective of where it executes, the end-to-end latency is always centered around 20ms with a negligible execution jitter in the order of microseconds. This jitter is due to the execution times of the copy-in and copy-out runnables which are executed with highest priorities at the points in time the LET communication takes place. Obviously, this leads to deviations compared to the idealized LET semantics.

6.2.2 Multi-rate chains

We next analyze effect chain $EC2$ composed of 3 runnables, with task periods 100, 10 and 2 ms. The resulting end-to-end latency distributions are presented in Figure 12. As expected, direct communication results in the least latency (say x), while implicit communication and LET have higher (almost 4x and 14x) latencies. This increased latency for implicit communication is attributed to the fact that results are not directly available after runnable completion, but rather after task completion. Obviously, this leads to situations where several instances of the receiving task are "missed" before the receiving runnable can read the data. The negative effect on the end-to-end latency is aggravated with increasing response time of

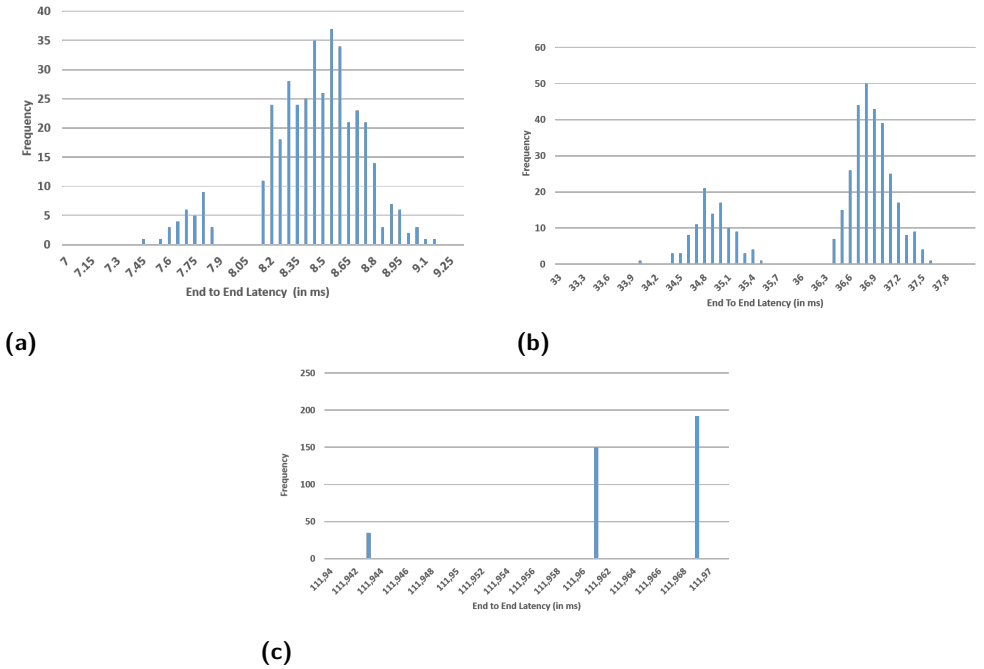


Figure 12 Comparison of latency overheads and variations across direct, implicit and LET communication.

the sending task. The increased latency for LET is purely attributed to the involved task periods since results are available only at the end of the task interval. As seen, the average latency is almost equal to the sum of the periods of the involved tasks. The reason for the jitter is the same as for event chain $EC1$.

6.3 Comparison of the communication overhead

In this experiment, we compute the overhead incurred by applying different mechanisms (see Figure 13). In order to compute it, we consider the cost of accesses of all labels in the application as described earlier, without factoring-in contention. The higher communication overhead for direct access is because remotely stored labels are fetched each time they are accessed. In contrast, references to local copies in implicit and LET communication result in slightly lower overhead. However, each time a new task accesses a label, new copies are to be made – and so, while direct communication is influenced by the frequency of label accesses, implicit and LET communication are influenced by the number of tasks accessing the label, since copies need to be maintained for every task.

6.4 Summary of experiments

We observe that data consistency via implicit communication or temporal determinism via LET comes at a cost of higher end-to-end event chain latencies and reduced communication overheads – but this trade-off is reasonable considering the intangible gains towards functional correctness of the systems and development, validation and deployment efforts.

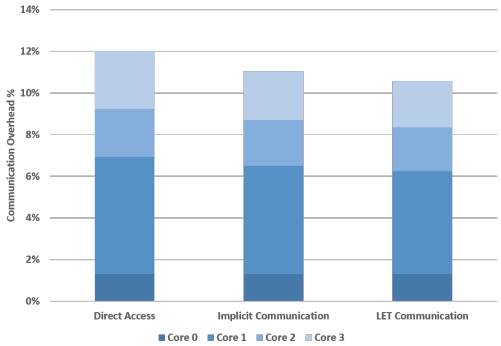


Figure 13 Comparison of communication overheads.

7 Related Work

For automotive embedded systems, in addition to meeting deadlines of individual tasks, meeting end to end latency requirements of event chains is crucial. These end to end timing requirements are described in standards such as AUTOSAR [5] and EAST-ADL [7]. In this regard, [18] compute end-to-end latencies considering the AUTOSAR implicit communication model and describe age and reaction constraints while introducing the first-to-first, first-to-last, last-to-first, last-to-last semantics. Kluge et al. [12] show how an LET based approach can facilitate compositionality by extending the MOSSCA multicore OS for LET. Henzinger et al. [10] proposed a methodology that supports distributed realtime code generation for distributed real-time systems considering LET. Farcas et al. [8] further demonstrate how a transparent task distribution is facilitated via LET and hence irrespective of where a task is mapped on a distributed system, the logical timing behavior is undisturbed. Pellizoni et al. [19] also proposed and analyze tasks under the "Predictable Execution Model" which have semantics similar to implicit communication. Our work is different in that we emphasise on the end-to-end latency implications of event chains in a real-world system. We also focus on how an existing tool can be extended to express these semantics.

8 Conclusion

In this paper we presented communication semantics used in the industry and their role in ensuring data consistency and temporal determinism in real-time multi-core embedded systems, and in particular an engine management system. We proposed model transformations to increase the expressiveness of existing tools and demonstrated the resulting impact on the system behavior. The consideration of communication semantics in widespread modeling approaches allowed us to evaluate their significant impact using a well-known timing analysis tool. These results can guide system designers to evaluate different optimization goals like minimizing communication overheads or reducing event-chain latencies, as well as ensuring deterministic system behavior.

References

1 S. Kramer A. Hamann, D. Ziegenbein and M. Lukasiewycz. Demonstration of the FMTV 2016 timing verification challenge. *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 00:1, 2016.

2 A. Hamann, S. Kramer, M. Pressler, D. Dasari, F. Wurst, and D. Ziegenbein. The industrial challenge of WATERS 2017 provided by Robert Bosch GmbH. URL: `http://waters2017.inria.fr/challenge/`.

3 AMALTHEA. An open platform project for embedded multicore systems. URL: http://www.amalthea-project.org.

4 AURIX. Aurix – safety joins performance. URL: http://www.infineon.com.

5 AUTOSAR – Spec. of Timing Extensions, 2014.

6 M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. 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, Aug 2016. doi:10.1109/RTCSA.2016.41.

7 EAST-ADL – Domain Model Specification, 2014.

8 E. Farcas, C. Farcas, W. Pree, and J. Templ. Transparent distribution of real-time components based on logical execution time. *SIGPLAN Not.*, 40(7):31–39, June 2005. doi:10.1145/1070891.1065915.

9 R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the SymTA/S approach. *Computers and Digital Techniques, IEEE Proceedings* -, 152(2):148–166, March 2005. doi:10.1049/ip-cdt:20045088.

10 T. A. Henzinger, C. M. Kirsch, and S. Matic. Composable code generation for distributed giotto. *SIGPLAN Not.*, 40(7):21–30, June 2005. doi:10.1145/1070891.1065914.

11 C. M. Kirsch and A. Sokolova. *The Logical Execution Time Paradigm*, pages 103–120. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

12 F. Kluge, M. Schoeberl, and T. Ungerer. Support for the logical execution time model on a time-predictable multicore processor. *SIGBED Rev.*, 13(4):61–66, November 2016.

13 L. Kosmidis, D. Compagnin, D. Morales, E. Mezzetti, E. Quinones, J. Abella, T. Vardanega, and F. J. Cazorla. Measurement-Based Timing Analysis of the AURIX Caches. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–11, 2016. doi:10.4230/OASIcs.WCET.2016.9.

14 S. Kramer, D. Ziegenbein, and A. Hamann. Real world automotive benchmarks for free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2015.

15 C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973. doi:10.1145/321738.321743.

16 E. Mezzetti M. Ziccardi, A. Cornaglia and T. Vardanega. Software-enforced Interconnect Arbitration for COTS Multicores. In *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*, volume 47 of *OpenAccess Series in Informatics (OASIcs)*, pages 11–20, 2015. doi:10.4230/OASIcs.WCET.2015.11.

17 L. Michel, T. Flaemig, D. Claraz, and R. Mader. Shared SW development in multicore automotive context. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France, January 2016. URL: https://hal.archives-ouvertes.fr/hal-01284591.

18 F. Nico, R. Kai, N. Johan, and J. Jan. A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. In *Int. Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, 2008.

19 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. doi:10.1109/RTAS.2011.33.

20 D. Reinhardt and G. Morgan. An embedded hypervisor for safety-relevant automotive e/e-systems. In *SIES*, 2014.

21 Timing Architect. Model-based development tools for embedded multi-core systems. URL: https://www.timing-architects.com.

Refinement of Workload Models for Engine Controllers by State Space Partitioning

Morteza Mohaqeqi[1], Jakaria Abdullah[2], Pontus Ekberg[3], and Wang Yi[4]

1 **Uppsala University, Uppsala, Sweden**
`morteza.mohaqeqi@it.uu.se`
2 **Uppsala University, Uppsala, Sweden**
`jakaria.abdullah@it.uu.se`
3 **Uppsala University, Uppsala, Sweden**
`pontus.ekberg@it.uu.se`
4 **Uppsala University, Uppsala, Sweden**
`yi@it.uu.se`

Abstract

We study an engine control application where the behavior of engine controllers depends on the engine's rotational speed. For efficient and precise timing analysis, we use the Digraph Real-Time (DRT) task model to specify the workload of control tasks where we employ optimal control theory to faithfully calculate the respective minimum inter-release times. We show how DRT models can be refined by finer grained partitioning of the state space of the engine up to a model which enables an exact timing analysis. Compared to previously proposed methods which are either unsafe or pessimistic, our work provides both abstract and tight characterizations of the corresponding workload.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

Keywords and phrases engine control tasks, schedulability analysis, minimum-time problem, DRT task model

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.11

1 Introduction

Cyber-physical systems are recognized by a tight integration of a physical process, which is subject to dynamic changes, and a computing system, which is responsible for handling the events initiated by the physical environment. From a real-time scheduling perspective, an event triggers a task to release a new job which must be completed within a specified time. Since the events are released according to the state of physical variables, and as the physical system is subject to dynamic variations, there may exist no regular or completely predictable release pattern for the jobs. In order to precisely specify the real-time workload in the mentioned systems, which is necessary for an accurate timing (and schedulability) analysis, one needs to (i) properly characterize the dynamic behavior of the physical system; and, (ii) construct reliable models (i.e., task systems) to capture the respective workload.

A key parameter in real-time task models is the minimum inter-release time between the jobs that are released by a task. In cyber-physical systems, such as an engine control application [11], this parameter can change at run time based on the physical system dynamics (i.e., engine speed in this case). Therefore, in order to accurately specify minimum inter-release times for each task, the relation between system dynamics and the release pattern of the corresponding events must be exploited in a rigorous way.

In this work, we employ a mathematical model of the physical system which is described through a set of differential equations. We show how the parameters of the corresponding real-time tasks, particularly minimum inter-release times, can be extracted in a faithful manner with a tunable accuracy. For this purpose, we formulate the problem of finding minimum inter-release times as an optimization problem. Then, we use the *calculus of variations* [13] from optimal control theory to calculate the minimum time required by the physical system to transfer from an initial state to a target state. The solution is used to derive the minimum inter-release time of the respective jobs (as the release of a job depends on the physical system's state). Based on the obtained values, a digraph real-time (DRT) [17] task is constructed as a representation of the engine control task. An important benefit of using the DRT task model is that, while it is expressive enough to capture the varying behavior of engine controllers, there exist efficient timing analysis methods for this task model for both static and dynamic priority scheduling policies [20, 22]. Also, using the theoretical framework of calculus of variations provides a "generic" solution which can be adopted for systems with more complicated dynamical behavior.

As it will be shown, construction of a DRT task for an engine controller requires partitioning of the problem state space. In this work, we also propose a method for partitioning the continuous range of the engine speed such that the resultant DRT task contains no pessimism with respect to schedulability analysis. As a result, our method provides a tight workload characterization of engine control tasks. To be consistent with the literature [5, 10, 4, 2], hereafter engine control tasks are referred to as AVR (adaptive variable-rate) tasks.

The rest of this paper is organized as follows. Previous work on timing analysis of AVR tasks is reviewed in the rest of this section. Section 2 introduces the system model considered in the current work, as well as a number of basic relations. Our method for constructing a DRT model for an AVR task is described in Section 3. Section 4 gives an illustrative example of how the proposed method works on a sample AVR task with realistic engine parameters. The faithfulness of the proposed method is formally established in Section 5. We present a method to construct the DRT task such that it provides an exact schedulability analysis of the AVR task in Section 6. The proposed method is evaluated in Section 7. A summary of the paper as well as some remarks for extending the work is presented in Section 8.

Related work: Real-time tasks with continually variable parameters were studied by Kim et al. [12] by introducing the *rhythmic* task model and proposing a respective response-time analysis method for rate monotonic scheduling. The method, however, is restricted to task sets with only one rhythmic task, which must be of the highest priority. Pollex et al. [16, 15] relaxed this constraint by assuming a task system containing an arbitrary number of so-called engine-triggered tasks. They proposed a sufficient schedulability test for such task sets in [16] under the assumption of an arbitrary but constant angular speed. Later, in [15], they removed the limitation to constant speeds, presenting a pessimistic analysis for the worst-case response time of engine-triggered tasks. The accuracy of the analysis was improved in [9] by taking the dependency of the tasks into account. The method also permits the execution time of the jobs to be specified as a *continuous* function of the rotational speed. In contrast to the analyses in [16, 15, 9], which present sufficient/pessimistic real-time analysis, our method provides both necessary and sufficient conditions for schedulability.

A major problem with characterizing the workload of an AVR task is infiniteness of the problem state space. This is because physical variables, such as the rotational speed, can vary continuously, and the fact that task parameters depend on such variables. Davis et al. [8] used state space quantization to address this issue, and then, presented a sufficient

schedulability test using Integer Linear Programming (ILP). However, the method only provides a sufficient test, containing additional pessimism due to the quantization.

Focusing on fixed priority schedulability of AVR tasks, Biondi et al. [5] overcame the infinite state space problem by recognizing a set of finite cases (for the speed at the instant of a job release) which *dominate* the whole state space. Based on this, schedulability analysis is reduced to a search problem in the space of all possible dominant cases. The same approach was used in [6] to derive the worst-case response time of a set of mixed periodic and AVR tasks under fixed priority scheduling. The approach was further employed to develop an EDF schedulability test for the respective tasks in [4]. In this case, dominant speeds are used to calculate the demand bound function (dbf) based on which feasibility analysis is accomplished. The analysis is proposed for those AVR tasks that depend on a single rotation source, and have a zero angular phase and the same period. In [4], it is also proposed to use a digraph structure, i.e., DRT graphs, for modeling and analysis of AVR tasks. A crucial restriction of the analyses presented in [5, 6, 4] is that they are valid only when the following assumption holds: between the release of two consecutive jobs of a task, the angular acceleration is constant. However, in the general case where the acceleration may vary between the release of two jobs, the proposed methods are not necessarily valid. We illustrate this issue in more details in Sections 4 and 7 by providing concrete counter examples.

A model for describing AVR tasks that are implemented with a hysteresis during mode switching has been proposed in [2]. Also, the dependency of the tasks on a common rotation source is taken into account. The work is more general than our method in terms of considering these realistic aspects. However, this generality comes at the cost of inaccuracy as they present only an upper bound on the system utilization, which is used for schedulability tests. The analysis is also restricted to the EDF scheduling algorithm. In a recent work, Biondi and Buttazzo [3] considered the speed estimation problem and its impact on schedulability analysis of AVR tasks. They argue that in practice, there is not a precise knowledge about the instantaneous speed of the engine. Based on this observation, they study the error that is introduced by state estimation into the analysis. Meanwhile, they do not focus on providing a new timing analysis method.

An initial idea of using the DRT task model for feasibility analysis of AVR tasks was also proposed by Guo and Baruah [10]. Compared to our work, the method in [10] considers a pessimistic assumption for extracting the inter-release times. More precisely, when switching from a mode to another one that corresponds to a higher speed range, the method assumes that the engine rotates with the maximum acceleration irrespective of whether this acceleration will lead to the designated speed range or not. Hence, the method provides a pessimistic lower bound on the minimum inter-release time which, as will be shown in Section 4, can be tightened. A similar assumption was used by Feld and Slomka [9] for response-time analysis of AVR tasks under fixed priority scheduling, where the maximum acceleration is used to compute the interfering workload of higher priority tasks in a given interval. As a result, the approach suffers from the same pessimism in timing analysis.

2 System model and preliminaries

This section presents the model used in the current study to describe the engine's dynamics. AVR and DRT task models are also briefly reviewed.

2.1 Dynamical system

In order to specify the state of the dynamical system and its evolution, three variables are defined: $\theta(t) \doteq$ angular position of the crankshaft, $\omega(t) \doteq$ angular velocity (rotational speed), and $\alpha(t) \doteq$ angular acceleration, where t denotes the physical time. These variables are related to each other through the following equations [9]:

$$\omega(t) = \dot{\theta} = \frac{d\theta}{dt}, \tag{1}$$

$$\alpha(t) = \dot{\omega} = \frac{d\omega}{dt}. \tag{2}$$

The minimum and maximum possible accelerations are denoted by α^- and α^+, respectively. Further, the rotational speed can vary in the range $[\omega_{min}, \omega_{max})$.

Under a fixed acceleration, θ and ω can be expressed as explicit functions of time. Towards this, assume a constant acceleration α during an interval of $[0, t_e]$. Then, from (1) and (2), the angular position and speed at any time instant $t \in [0, t_e]$ can be computed by

$$\theta(t) = \frac{1}{2}\alpha t^2 + \omega_0 t + \theta_0, \tag{3}$$

$$\omega(t) = \omega_0 + \alpha t, \tag{4}$$

where θ_0 and ω_0 denote the initial angle and the initial speed, respectively. From these equations, the relation between θ and ω under a constant acceleration α can be specified as

$$\theta(\omega) = \frac{1}{2\alpha}\left(\omega^2 - \omega_0^2\right) + \theta_0. \tag{5}$$

Assuming an initial speed ω_0 and a constant acceleration of α, the time it takes to have an angular change of $\Delta\theta$ can be calculated, as shown in [7], by

$$T(\omega_0, \alpha, \Delta\theta) = \frac{\sqrt{\omega_0^2 + 2\alpha\Delta\theta} - \omega_0}{\alpha}. \tag{6}$$

Also, the speed after an angular change of $\Delta\theta$, denoted by $\Omega(\omega_0, \alpha, \Delta\theta)$, is derived as, [5]:

$$\Omega(\omega_0, \alpha, \Delta\theta) = \sqrt{\omega_0^2 + 2\alpha\Delta\theta}. \tag{7}$$

According to the described relations, we point out two properties assuming the domain of non-negative values for ω_0 and $\Delta\theta$.

▶ **Property 1.** *Function $T(\omega_0, \alpha, \Delta\theta)$ defined in (6) is (strictly) decreasing in ω_0 and α.*

▶ **Property 2.** *Function $\Omega(\omega_0, \alpha, \Delta\theta)$ defined in (7) is (strictly) increasing in ω_0 and α.*

Throughout this work, we assume that $\Delta\theta$ specifies the crankshaft revolution in terms of "number of rotations". As a result, a full rotation is recognized by $\Delta\theta = 1$. Hence, according to (7), and assuming an initial speed of ω_0 and a constant acceleration of α, the speed after one complete revolution, denoted by $\Omega_1(\omega_0, \alpha)$, would be $\Omega_1(\omega_0, \alpha) = \sqrt{\omega_0^2 + 2\alpha}$. As a generalization, by $\Omega_i(\omega_0, \alpha)$ we denote the speed after exactly i rotations, for $i \in \mathbb{N}$, which can be calculated (under the same assumptions) as

$$\Omega_i(\omega_0, \alpha) = \sqrt{\omega_0^2 + 2i\alpha}. \tag{8}$$

2.2 AVR tasks

We consider the AVR task model that is commonly used in the literature (e.g., [5, 2, 4, 10]). Based on this model, an AVR task releases a new job whenever the crankshaft of the engine reaches a specific angle. Without loss of generality, we assume that the jobs are released whenever one full rotation is taken (i.e., $\Delta\theta = 1$).[1] The functionality of an AVR task is determined according to the instantaneous rotational speed. More specifically, a task comprises a set of M *modes*, each of which is related to a certain speed range. At run time, according to the instantaneous speed, a mode is selected and the corresponding functionality is executed. Based on this, each mode is specified by a pair $(C_m, [\omega_m, \omega_{m+1}))$ which associates a worst-case execution time (WCET) C_m and a speed range $[\omega_m, \omega_{m+1})$ to the mode. Consequently, the set of modes associated to a task is described as $\mathcal{M} = \{(C_m, [\omega_m, \omega_{m+1})) \mid m = 1, 2, \ldots, M\}$, where $\omega_1 = \omega_{min}$ and $\omega_{M+1} = \omega_{max}$.

2.3 The DRT task model

Our goal is to abstract the behavior of engine control applications such that, while having an accurate characterization, one can achieve efficient timing analysis. For this purpose, we use the DRT task model, which provides the facility of modeling real-time tasks with variable (mode switching) behavior, as described below.

A DRT task T is specified by a directed graph $G(T) = (V, E)$, where V and E denote the set of vertices and edges, respectively. Each vertex $v \in V$ corresponds to a specific *job type*, labeled by a pair of non-negative integers $\langle e(v), d(v)\rangle$, where $e(v)$ and $d(v)$ represent the WCET and relative deadline of the respective jobs. Each edge $(v, u) \in E$ is also labeled by a non-negative integer $p(v, u)$ which shows the minimum inter-release time between any job of v and any job of u. In such a graph, the existence of multiple outgoing edges from one vertex represents a notion of non-determinism in the type of the next job. Hence, each path in the graph is related to a possible sequence of jobs which can be released by the task. While this real-time task model provides a high degree of expressiveness, there exists quite efficient methods for feasibility [17], schedulability [19], and response-time [18] analysis of DRT task sets. In subsequent sections, we use the terms DRT task and DRT graph interchangeably.

3 AVR to DRT transformation

For modeling an AVR task as a DRT graph, two problems need to be addressed: (i) partitioning the speed range (i.e., $[\omega_{min}, \omega_{max})$) into an appropriate set of sub-intervals (each of which will be mapped to a DRT vertex); and (ii) constructing a DRT graph (and deriving its parameter values) based on these sub-intervals. For a better presentation, we first tackle the second one in this section, where we show how one can obtain a DRT graph for an AVR task assuming a *given* speed range partitioning. In Section 6, we deal with the first issue, and present a method to partition the speed range such that schedulability analysis of the obtained DRT graph provides an exact schedulability test for the original AVR task. We will refer to such a DRT graph as a tight model for the AVR task.

[1] To see how the general case can be considered in the system model, see, for example, [4].

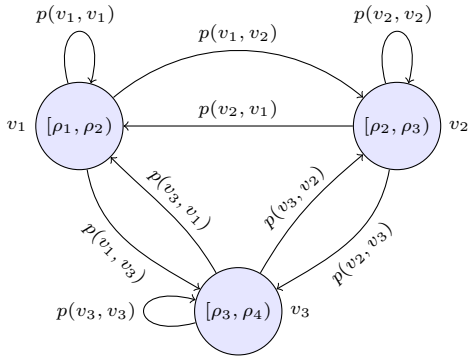


Figure 1 A sample DRT task with three vertices.

3.1 Transformation method

The idea is to capture the continuous state space of the engine speed, which determines the AVR task functionality, by a number of K vertices (job types) of a DRT graph. For this purpose, we assume a partition of the speed range $[\omega_{min}, \omega_{max})$ into a set of K sub-intervals $\{[\rho_1, \rho_2), \ldots, [\rho_K, \rho_{K+1})\}$, where $\rho_1 = \omega_{min}$ and $\rho_{K+1} = \omega_{max}$. Each sub-interval will be mapped a vertex of the DRT task. A trivial partitioning can be obtained by the speed ranges linked to the modes of the considered AVR task. However, any other partitioning can be used as well, leading to different levels of precision of the obtained DRT task, and thus, the respective timing analyses.

Figure 1 demonstrates a sample DRT task where each vertex is associated with a speed range. Based on this model, when the speed at the job release time is a value in the range of $[\rho_i, \rho_{i+1})$, for $i \in \{1, 2, 3\}$, a job of type v_i is released.

After the set of vertices are obtained, the label of each edge $e = (v, u)$, namely the minimum inter-release time, is determined based on the following observation. A job of u can be released whenever an entire rotation is taken by the crankshaft after the release of a job of type v. As a result, the minimum inter-release time is equal to the minimum time that it takes for the crankshaft to have a full rotation. The initial speed may be any value in the range associated to vertex v. In turn, the final speed (i.e., the speed exactly at the moment when the rotation completes) can be any value in the speed range of vertex u.

Based on these descriptions, we construct a DRT graph for a given AVR task. The procedure is shown in Algorithm 1. In Line 6 of Algorithm 1, the WCET of each vertex is determined by the maximum execution time of those modes whose speed range has an overlap with the speed range associated to that vertex. Next, the edge labels are specified (Lines 10 to 16). As seen, the procedure depends on calculating the minimum inter-release times by calling MinTime(.). The procedure MinTime(.) gets two speed ranges and uses a method from the optimal control theory to calculate the minimum possible time required for one rotation starting from a speed in the first range and ending in a speed in the second range. A formal description of the problem of finding the minimum time under these constraints and its solution are presented in the next subsection.

In order to determine the relative deadline of job types, we assume that each job needs to be finished before the release of the next job. As a result, for each job type, the minimum inter-release time among all outgoing edges from the respective vertex is considered as its relative deadline (Line 19 in Algorithm 1).

Algorithm 1 Deriving a DRT graph for an AVR task

input: A set $\mathcal{M} = \{(C_m, [\omega_m, \omega_{m+1})) \mid m = 1, \ldots, M\}$ of AVR task modes; Speed intervals $\{[\rho_1, \rho_2), \ldots, [\rho_K, \rho_{K+1})\}$.
output: A DRT graph.

procedure ConstructDRT
  $V \leftarrow \{\}, E \leftarrow \{\}$
  ▷ Creating the vertices:
  for $i \leftarrow 1$ **to** K **do**
    $v_i \leftarrow$ A new vertex
    $e(v_i) \leftarrow \max_{1 \le j \le M} \{C_j \mid [\rho_i, \rho_{i+1}) \cap [\omega_j, \omega_{j+1}) \neq \emptyset\}$
    $V \leftarrow V \cup \{v_i\}$
  end for
  ▷ Creating the edges:
  for all $v_i, v_j \in V$ **do**
    $p \leftarrow$ MinTime($[\rho_i, \rho_{i+1}), [\rho_j, \rho_{j+1})$)
    if $p \neq \infty$ **then**
      $p(v_i, v_j) \leftarrow p$
      $E \leftarrow E \cup \{(v_i, v_j)\}$
    end if
  end for
  ▷ Setting relative deadlines:
  for $i \leftarrow 1$ **to** K **do**
    $d(v_i) \leftarrow \min \{p(v_i, v_j) \mid (v_i, v_j) \in E\}$
  end for
  $G \leftarrow (V, E)$
  return G
end procedure

3.2 Computing minimum inter-release times

The goal is to calculate minimum inter-release times in the obtained DRT graph. For this, we first express the problem of finding minimum inter-release times as a minimization problem.

▶ **Problem 1** (Minimum-time problem). *Consider a dynamical system described by Eqs.* (1) *and* (2). *Also, assume that* $[\rho_i, \rho_{i+1})$ *and* $[\rho_f, \rho_{f+1})$ *are two given speed intervals. The problem is to find the minimum time* $t^* > 0$ *for which* $\theta(t^*) = 1$ *(which specifies one complete rotation), subject to* $\theta(0) = 0$, $\omega(0) \in [\rho_i, \rho_{i+1})$, $\omega(t^*) \in [\rho_f, \rho_{f+1})$, *and*

$$\omega_{min} \le \omega(t) \le \omega_{max}, \qquad \forall t \in [0, t^*], \tag{9a}$$

$$\alpha^- \le \alpha(t) \le \alpha^+, \qquad \forall t \in [0, t^*]. \tag{9b}$$

In words, the constraints express that we start at the angular position 0 *with a speed in interval* $[\rho_i, \rho_{i+1})$ *and want to end up in speed interval* $[\rho_f, \rho_{f+1})$ *after one full rotation.*

In the context of optimal control of dynamical systems, this problem is regarded as a *minimum-time* control problem [13]. Stating Problem 1 in that context, we can think of the acceleration function $\alpha(t)$ as the control input. Then, the problem is to find the optimal control (namely, the function $\alpha(t)$) which reveals the minimum time, while the initial and terminal states are constrained. The so-called *Pontryagin's minimum principle* [13] provides necessary conditions for an optimal solution to this problem. The principle is based on the notion of Hamiltonian function, reviewed below.

▶ **Definition 1** (Hamiltonian, [13]). Consider the dynamical system and the optimization problem specified in Problem 1. Further, let ψ_1 and ψ_2 denote two functions (of t). Then, the Hamiltonian is defined as[2]

$$H(\psi_1, \psi_2, \theta, \omega, \alpha) = \psi_1 \omega + \psi_2 \alpha + 1. \tag{10}$$

▶ **Theorem 2** (Pontryagin's minimum principle, [13]). *Let $\alpha^*(.)$ be an acceleration function which reveals the optimal solution of Problem 1. Then, functions ψ_1 and ψ_2 exist which satisfy*

$$\frac{d\psi_1}{dt} = \frac{\partial H}{\partial \theta}, \text{ and } \frac{d\psi_2}{dt} = \frac{\partial H}{\partial \omega}, \tag{11}$$

and

$$H(\psi_1, \psi_2, \theta, \omega, \alpha^*) \leq H(\psi_1, \psi_2, \theta, \omega, \alpha) \tag{12}$$

for all admissible controls $\alpha(.)$ and all times $t \in [t_0, t_f]$.

The theorem expresses that the optimal control α^* for the original problem must minimize the Hamiltonian as well. Thus, instead of solving Problem 1 directly, we focus on finding the control function that minimizes the Hamiltonian, which is typically an easier problem.

The necessary condition stated by the theorem provides a way to characterize the optimal control. Specifically, from (11) in Theorem 2, and by taking the derivative of the Hamiltonian, i.e., $H()$ defined in (10), with respect to θ and ω, we reach $\dot{\psi_1}(t) = 0$ and $\dot{\psi_2}(t) = -\psi_1$, which yield

$$\begin{aligned} \psi_1(t) &= c_1, \\ \psi_2(t) &= -c_1 t + c_2, \end{aligned} \tag{13}$$

for two (unknown) constants c_1 and c_2. By replacing $\psi_1(t)$ and $\psi_2(t)$ in (10) with their definition in (13), the Hamiltonian is obtained as

$$H(\psi_1, \psi_2, \theta, \omega, \alpha) = c_1 x_2 + (-c_1 t + c_2)\, \alpha + 1. \tag{14}$$

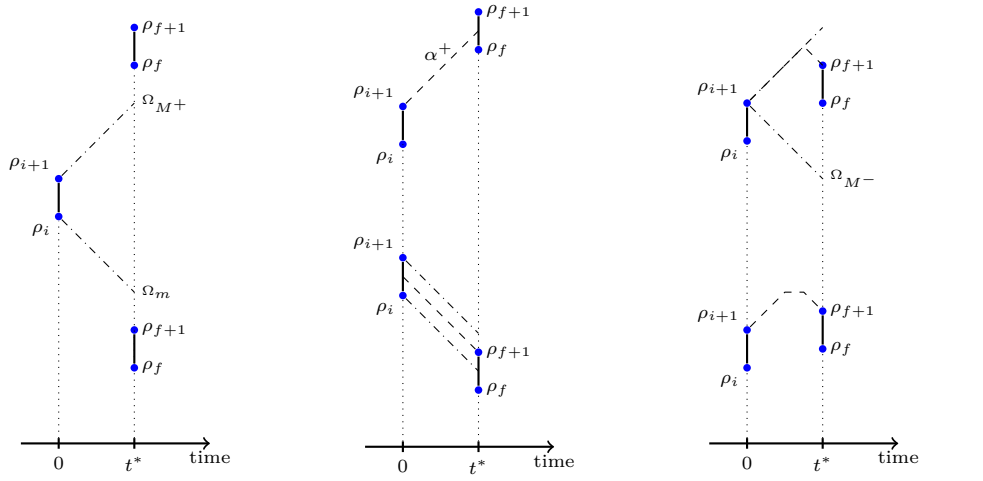
According to the Pontryagin's minimum principle (Theorem 2), a necessary condition for a control variable α to be an optimal solution to Problem 1 is that it minimizes the Hamiltonian. Based on (14), the acceleration function which minimizes the Hamiltonian is determined by

$$\alpha^*(t) = \begin{cases} \alpha^+, & \text{if } -c_1 t + c_2 < 0, \\ \alpha^-, & \text{if } -c_1 t + c_2 > 0. \end{cases} \tag{15}$$

Consequently, as also shown in [13, 21] for a similar setting, the optimal control function, i.e., the engine acceleration which minimizes the time, should be

$$\alpha^*(t) = \begin{cases} \alpha^+, & \text{for all } t \in [0, t^*], \text{ or} \\ \alpha^-, & \text{for all } t \in [0, t^*], \text{ or} \\ \alpha^+, & \text{for } t \in [0, t_1) \text{ and } \alpha^-, \text{ for } t \in [t_1, t^*], \text{ or} \\ \alpha^-, & \text{for } t \in [0, t_1) \text{ and } \alpha^+, \text{ for } t \in [t_1, t^*]. \end{cases} \tag{16}$$

[2] The Hamiltonian does not directly depend on θ in our problem. However, to follow the general form, we include θ in the arguments list. For a generic definition of the Hamiltonian see [13].



(a) Case 1 (infeasible) **(b)** Case 2,3 (fixed accel.) **(c)** Case 4 (variable accel.)

Figure 2 Possible cases for minimum time revolution. Dashed lines indicate speed variation in the optimal solution; dot-dashed lines show speed evolution under an extreme (minimum or maximum) acceleration; t^* stands for the instant of one full rotation.

(as long as the system constraints, i.e., lower and upper speed and acceleration bounds, are not violated.) In order to determine which case in (16) provides the optimal solution, we need a secondary condition. For this, we notice that, in the beginning, the system must start with its maximum possible speed that can lead to a feasible solution. The reason is as follows. Assume an optimal solution where the system does not start from the maximum speed in the permitted range. This means that there exists a higher speed at which the system can start and yet provide a feasible solution. Based on Property 1, this solution provides a smaller time, or equivalently, a better solution compared to the presumed optimal one, which is a contradiction. This observation, which is referred to as the *transversality condition* [21], is used to obtain the actual optimal solutions, as described in the following.

The optimal solution of Problem 1 is determined according to the value of the problem parameters, i.e., α^-, α^+, ρ_i, ρ_{i+1}, ρ_f, and ρ_{f+1}. More particularly, with respect to the existence of a solution, and also the criteria specified by (16) for computing the optimal result, if it exists, different cases can be identified. To introduce and deal with these cases, we first define the following notations (using Eq. (8))

$$\Omega_m \doteq \Omega_1(\rho_i, \alpha^-) = \sqrt{\rho_i^2 + 2\alpha^-} \tag{17a}$$

$$\Omega_{M^-} \doteq \Omega_1(\rho_{i+1}, \alpha^-) = \sqrt{\rho_{i+1}^2 + 2\alpha^-} \tag{17b}$$

$$\Omega_{M^+} \doteq \Omega_1(\rho_{i+1}, \alpha^+) = \sqrt{\rho_{i+1}^2 + 2\alpha^+} \tag{17c}$$

Figure 2 depicts all possible situations regarding an optimal solution. We formally specify and treat each case as follows.

Case 1: $\Omega_{M^+} \leq \rho_f$ or $\Omega_m \geq \rho_{f+1}$ (Fig. 2a). In this case, the problem has no feasible solution (this is implied by Property 2 and the continuity of $\Omega(.,.,.)$ defined in (7)).

Case 2: $\rho_f < \Omega_{M^+} \leq \rho_{f+1}$ (depicted in the upper part of Fig. 2b). Under this condition, selecting an initial speed of ρ_{i+1} and assigning $\alpha = \alpha^+$, $\forall t \in [0, t^*]$ reveal the minimum time. Therefore, the duration of one rotation can be calculated through Eq. (6).

Case 3: $\Omega_m < \rho_{f+1} \leq \Omega_{M^-}$ (seen in the lower part of Fig 2b). In this case, the optimal time is achieved when the acceleration is selected to be α^- and the initial speed is selected such that the final speed is ρ_{f+1}. In other words, from (8), the initial speed (denoted by ρ_0) must satisfy $\rho_{f+1} = \sqrt{\rho_0^2 + 2\alpha^-}$. This yields

$$\rho_0 = \sqrt{\rho_{f+1}^2 - 2\alpha^-}. \tag{18}$$

Again, the duration of one rotation can be calculated by Eq. (6) with assigning $\omega_0 = \rho_0$, $\alpha = \alpha^-$, and $\Delta\theta = 1$.

Case 4: $\Omega_{M^-} \leq \rho_{f+1} \leq \Omega_{M^+}$ (seen in Fig 2c). In this situation, in the optimal solution, the initial and final speed will be the maximum possible values, namely ρ_{i+1} and ρ_{f+1}, respectively. Also, the acceleration will be chosen to be α^+ until a certain instant, denoted by t_1, and then it will be switched to α^- (due to the transversality condition [21]). To obtain the minimum time, we use the result of the following lemma.

▶ **Lemma 3.** *Let ρ_1 denote the rotational speed at time t_1 defined in case 4. Then, ρ_1 can be computed as*

$$\rho_1 = \sqrt{\frac{2\alpha^-\alpha^+ + \alpha^-\rho_{i+1}^2 - \alpha^+\rho_{f+1}^2}{\alpha^- - \alpha^+}}. \tag{19}$$

Proof. Let θ_1 represent the position at t_1. From (5) it follows that $\theta_1 = \frac{1}{2\alpha^+}(\rho_1^2 - \rho_{i+1}^2)$. Also, as the final position is assumed to be 1, we can write $1 = \frac{1}{2\alpha^-}(\rho_{f+1}^2 - \rho_1^2) + \theta_1$. Substituting θ_1 from the former equation to the latter one reveals $1 = \frac{\rho_{f+1}^2}{2\alpha^-} - \frac{\rho_{i+1}^2}{2\alpha^+} + \rho_1^2(\frac{1}{2\alpha^+} - \frac{1}{2\alpha^-})$. Solving this equation for ρ_1 gives (19). ◀

Two different cases may arise according to the value of ρ_1 calculated in Lemma 3. If the speed limit is not violated, i.e., if $\rho_1 \leq \omega_{max}$, then the minimum time for one rotation, denoted by t^*, can be computed as (based on (4))

$$t^* = \frac{\rho_1 - \rho_{i+1}}{\alpha^+} + \frac{\rho_{f+1} - \rho_1}{\alpha^-}. \tag{20}$$

On the other hand, if $\rho_1 > \omega_{max}$, then after reaching the maximum speed, the acceleration is forced to be 0 for a while and then, turning to α^-. Under this scenario, we have $t^* = t_1^* + t_2^* + t_3^*$, where t_1^*, t_2^*, and t_3^* denote the time durations in which the acceleration is α^+, 0, and α^-, respectively, and are calculated as

$$\begin{aligned} t_1^* &= (\omega_{max} - \rho_{i+1})/\alpha^+, \\ t_2^* &= \frac{1}{\omega_{max}}\left(1 - \frac{\omega_{max}^2 - \rho_{i+1}^2}{2\alpha^+} - \frac{\rho_{f+1}^2 - {\omega_{max}}^2}{2\alpha^-}\right), \\ t_3^* &= (\rho_{f+1} - \omega_{max})/\alpha^-. \end{aligned} \tag{21}$$

Algorithm 2 summarizes the described procedure for solving Problem 1. As seen, according to each case, a respective value is calculated, and finally returned.

▶ **Lemma 4.** *Consider two vertices v_i and v_f in the constructed DRT graph associated with speed intervals $[\rho_i, \rho_{i+1})$ and $[\rho_f, \rho_{f+1})$, respectively. Additionally, assume two jobs J_1 and J_2 released by the AVR task such that the rotational speed at the release time of J_1 lies in $[\rho_i, \rho_{i+1})$, and at the release time of J_2 lies in $[\rho_f, \rho_{f+1})$. Then, the inter-release time of J_1 and J_2 is not smaller than $p(v_i, v_f)$.*

Proof. The Lemma follows from the optimality (minimality) of the solution provided by Algorithm 2 for Problem 1. ◀

Algorithm 2 Computing minimum time

input: $[\rho_i, \rho_{i+1})$: Initial speed range; $[\rho_f, \rho_{f+1})$: Final speed range.
output: p: A lower bound on the duration of one rotation.

1: **procedure** MinTime($[\rho_i, \rho_{i+1})$, $[\rho_f, \rho_{f+1})$)
2: Take Ω_m, Ω_{M^-}, and Ω_{M^+} as defined in (17).
3: **if** $\rho_f \geq \Omega_{M^+}$ **or** $\rho_{f+1} \leq \Omega_m$ **then**
4: $p \leftarrow \infty$
5: **else if** $\Omega_{M^+} \in (\rho_f, \rho_{f+1}]$ **then**
6: $p \leftarrow T(\rho_{i+1}, \alpha^+)$ ▷ Using (6)
7: **else if** $\rho_{f+1} \in (\Omega_m, \Omega_{M^-}]$ **then**
8: $p \leftarrow T(\rho_0, \alpha^-)$ ▷ ρ_0 is computed in (18)
9: **else**
10: Take ρ_1 as in (19).
11: **if** $\rho_1 \leq \omega_{max}$ **then**
12: $p \leftarrow t^*$ ▷ t^* is calculated by (20)
13: **else**
14: $p \leftarrow t_1^* + t_2^* + t_3^*$ ▷ See (21)
15: **end if**
16: **end if**
17: **return** p
18: **end procedure**

4 An illustrative example

In this section, we show how the proposed method delivers minimum inter-release times in comparison with existing methods using a sample AVR task. Also, it is shown how the accuracy of the analysis can be improved by using more vertices in the DRT task. For our purpose, we consider an AVR task with the parameters used in [6] and [8]. Accordingly, the minimum and maximum speeds are assumed as $\omega_{min} = 500$ rpm and $\omega_{max} = 6500$ rpm. Further, the acceleration range is set as $[\alpha^-, \alpha^+] = [-600000, 600000]$.[3] This setting is regarded as a reasonable representative for typical production car engines [8].

4.1 Calculating minimum inter-release time

This section demonstrates how the proposed method can provide a safe and also accurate value for minimum inter-release times. We assume two modes with speed ranges $[500, 600)$ and $[700, 800)$. The goal is to calculate the minimum time of one full rotation, while the initial and final speeds are restricted to be in the former and latter speed range, respectively. The diagram in the bottom of Fig. 3 shows the acceleration during one rotation for three scenarios. The first scenario, represented with solid lines, is related to the proposed method, where case 4 applies (Lines 11-12 in Algorithm 2). As seen, the acceleration is selected to get its maximum value, namely 600×10^3, up to a certain point, and then, its minimum value, i.e., -600×10^3, during the rest of the interval. In contrast, the dashed line indicates the acceleration which leads to the minimum time, when the system is supposed to select a fixed acceleration during one rotation. In addition, the dot-dashed line corresponds to a pessimistic approach where the acceleration constantly is assigned the maximum value.

The diagram in the middle of Fig. 3 shows the corresponding speeds for the mentioned three approaches as a function of time. Also, the topmost diagram shows the respective

[3] We use the *revolution per minute (rpm)* unit for rotational speed and *rpm per minute* for acceleration.

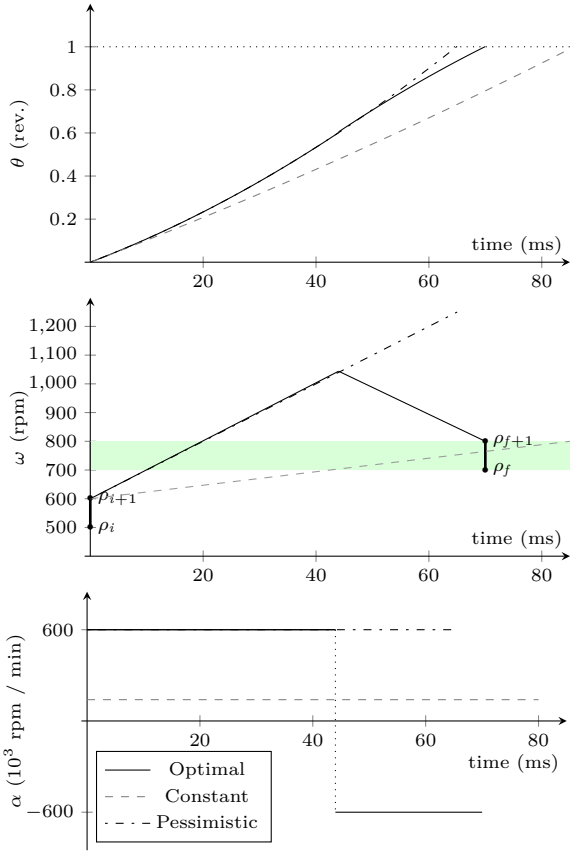


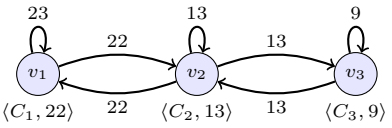
Figure 3 Illustration of a rotation that leads to minimum time using three approaches. The diagram in the bottom shows the acceleration, the middle one shows the respective speeds, and the upper one depicts the corresponding angular positions.

angular variations. In this diagram, the time for one rotation is the point at which the associated curve hits the horizontal (dotted) line. As seen, the minimum time (related to the solid line) is around 70 ms, while the respective time for the case of a fixed acceleration will be nearly 85 ms. This comparison shows that the assumption of fixed acceleration between the release of two successive jobs, as is made in [5], [6], and [4], may lead to an optimistic result. On the other hand, one can obtain a pessimistic result by relaxing the constraint on the final speed, and then, selecting a constant maximum acceleration (as used in [10]). This situation is shown by the dot-dashed line in the diagram, revealing a minimum time of around 65 ms. In contrast, our proposed method exhibits a tighter, yet safe, result.

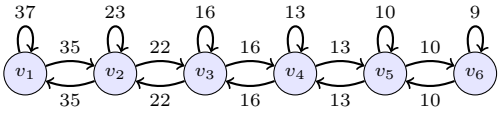
4.2 Deriving a DRT task

For extracting a DRT graph, we consider an AVR task with the same speed and acceleration ranges as before. Further, we assume three modes recognized by the following speed ranges: $[500, 2500)$, $[2500, 4500)$, and $[4500, 6500)$. WCETs of the respective jobs for these modes are assumed to be 5 ms, 3 ms, and 1 ms, respectively.

Figures 4a and 4b show the corresponding DRT tasks obtained through the proposed method when using three and six vertices, respectively. The values on edges are in milliseconds and have been rounded down to get a safe approximation of the minimum inter-release times.



(a) An extracted DRT with three vertices.



(b) An extracted DRT with six vertices.

Figure 4 DRT graphs for a sample AVR task.

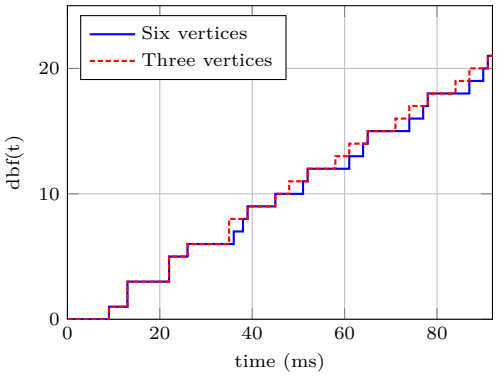


Figure 5 Demand-bound functions.

In the DRT graph shown in Fig. 4a, one vertex is considered per mode. In contrast, the graph shown in Fig. 4b has two vertices for each mode of the AVR task, providing a model with finer granularity. For this model, we considered six speed intervals with equal lengths.

To observe how the number of vertices influences timing analysis, we have calculated the respective *demand-bound function* (dbf) [20] for each DRT task. Informally, a dbf is a function of time; for any time instant $t \geq 0$ its value denotes the maximum accumulated workload that can arrive within any time interval of length t and have a deadline in the same interval. This function provides a direct way to express a necessary and sufficient condition for feasibility of a DRT task set [17]. As seen in Fig. 5, the task model with six vertices always delivers an equal or smaller (i.e., tighter) workload than that of the model with three vertices. This means that an AVR task which is actually schedulable, may be (pessimistically) deemed as an unschedulable task when analyzed with a coarse grain model, i.e., the DRT graph with three vertices. But when the graph is *refined* to a DRT task with more vertices, it may be (correctly) recognized as a schedulable task.

5 Faithfulness of the method

Intuitively, a DRT graph is said to be a faithful (or sound) model for an AVR task if its schedulability implies schedulability of the AVR task. In the following, we formalize this definition, and then, validate that our method constructs a faithful model. For this purpose, we first provide a number of required definitions.

5.1 Definitions

For an AVR task with a given set of modes, let $C(\omega)$ denote a function which takes a speed value ω and returns the WCET of the mode to which ω belongs. Further, by a triple (R_i, C_i, ω_i) we denote a job released by the AVR task with a WCET of C_i and an instantaneous rotational speed of ω_i at time instant R_i. Also, we define a notion of *valid trajectory* as follows.

▶ **Definition 5** (Valid trajectory)**.** Given two speeds ω_0 and ω_f, and a subset of real numbers $I = [r_1, r_2]$ with $r_1, r_2 \in \mathbb{R}_{\geq 0}$, a function $\alpha(t) : I \mapsto \mathbb{R}$ is said to be a valid acceleration

trajectory, or simply a valid trajectory, from ω_0 to ω_f in the domain of I if it satisfies

$$\begin{aligned} &\alpha^- \leq \alpha(t) \leq \alpha^+, && \forall t \in [r_1, r_2] \quad \wedge \\ &\omega_{min} \leq \tilde{\omega}(t) \leq \omega_{max}, && \forall t \in [r_1, r_2] \quad \wedge \\ &\tilde{\omega}(r_2) = \omega_f, && \wedge \\ &\int_{r_1}^{r_2} \tilde{\omega}(t)dt = 1, \end{aligned} \tag{22}$$

where $\tilde{\omega}(x)$ is defined as $\tilde{\omega}(x) = \omega_0 + \int_{r_1}^{x} \alpha(t)dt$.

In order to capture the workload generated by AVR tasks, we define the notion of $\mathcal{A}$-*trace*.

▶ **Definition 6** ($\mathcal{A}$-trace)**.** Take an AVR task as specified in Section 2.2. Then, any sequence of triples $[(R_1, C_1, \omega_1), \ldots, (R_n, C_n, \omega_n)]$ is called an "$\mathcal{A}$-trace" generated by the AVR task if

- $C_i = C(\omega_i)$ for all $i \in \{1, \ldots, n\}$, and
- for each $i \in \{1, \ldots, n-1\}$ there exists a valid trajectory from ω_i to ω_{i+1}, defined over the domain of $[R_i, R_{i+1}]$.

Similarly, a notion of $\mathcal{D}$-*trace* is defined for a DRT task.

▶ **Definition 7** ($\mathcal{D}$-trace)**.** Consider a DRT task $G = (V, E)$. Let $v_1 \to v_2 \to \ldots \to v_n$ be an arbitrary path in G. Then, any sequence of triples $[(R_1, C_1, v_1), \ldots, (R_n, C_n, v_n)]$ is called a $\mathcal{D}$-trace generated by the task if

- $C_i \leq e(v_i)$ for all $i \in \{1, \ldots, n\}$, and
- $R_{i+1} - R_i \geq p(v_i, v_{i+1})$ for all $i \in \{1, \ldots, n-1\}$.

For specifying the workload associated to the AVR and DRT task models in an abstract and also comparable way, we further define a notion of job sequence.

▶ **Definition 8** (Job sequence)**.** Any sequence of tuples $[(R_1, C_1), \ldots, (R_n, C_n)]$ with $R_i, C_i \in \mathbb{R}_{\geq 0}$, for $1 \leq i \leq n$, and $R_i < R_{i+1}$, for $1 \leq i < n$, is called a job sequence.

A job sequence $\sigma = [(R_1, C_1), \ldots, (R_n, C_n)]$ is said to be producible by an AVR task if and only if there exist values $\omega_1, \omega_2, \ldots, \omega_n$ such that $[(R_1, C_1, \omega_1), \ldots, (R_n, C_n, \omega_n)]$ is an $\mathcal{A}$-trace generated by the task. Analogously, σ can be generated by a DRT task G if and only if there exists a path $v_1 \to v_2 \to \ldots \to v_n$ in G such that $[(R_1, C_1, v_1), \ldots, (R_n, C_n, v_n)]$ is a $\mathcal{D}$-trace generated by G. Such workload characterization conforms to the semantics of DRT [20], which enables us to employ the respective timing analysis methods [19, 20]. According to these definitions, the notion of behavioral inclusion is defined as follows.

▶ **Definition 9** (Behavior inclusion)**.** Consider a DRT task, denoted by T_D, and an AVR task, denoted by T_A. Let σ_D and σ_A denote the set of all job sequences generated by T_D and T_A, respectively. Then, the behavior of T_A is said to be included by T_D if $\sigma_A \subseteq \sigma_D$.

Finally, we define faithfulness of a model as follows.

▶ **Definition 10** (Faithfulness)**.** A DRT graph T_D provides a faithful model for an AVR task T_A if schedulability of T_D entails schedulability of T_A.

5.2 Faithfulness validation

The goal is to demonstrate that Algorithm 1 reveals a faithful model for a given AVR task. For this purpose, we first review a result with respect to behavior inclusion.

▶ **Lemma 11.** *Consider a set of AVR tasks τ_A and a set of corresponding DRT tasks τ_D where the behavior of each AVR task is included by the respective DRT task. Then, schedulability of τ_D implies schedulability of τ_A.*

Proof. We prove the contrapositive. Let τ_A be unschedulable. Then, there exists a set of job sequences, each produced by one task, in which a job misses its deadline. As it is supposed that the behavior of every AVR task is included by its corresponding DRT task, the same set of job sequences can be generated by τ_D as well. As a result, there exists a scenario in which τ_D is unschedulable, too, by which the lemma is followed. ◀

Now, we will show that the DRT graph created by Algorithm 1 includes the behavior of the given AVR task. For this goal, we review the following property regarding the algorithm.

▶ **Property 3.** *Consider an AVR task T_A and the corresponding DRT task T_D. Let $[(R_1, C_1, \omega_1), \ldots, (R_n, C_n, \omega_n)]$ denote an arbitrary $\mathcal{A}$-trace generated by T_A. Further, let $v(\omega_i)$ denote the vertex in the graph of T_D whose corresponded speed range includes ω_i. Then, based on Line 6 in Algorithm 1, we have*

$$C(\omega_i) \leq e(v(\omega_i)), \quad \text{for } 1 \leq i \leq n. \tag{23}$$

Moreover, according to Lemma 4, it holds that

$$p\left(v(\omega_i), v(\omega_{i+1})\right) \leq R_{i+1} - R_i, \quad \text{for } 1 \leq i < n. \tag{24}$$

Using this property, we establish a relation between an AVR task and its associated DRT graph in terms of behavioral inclusion.

▶ **Lemma 12.** *The DRT task constructed by Algorithm 1 includes the behavior of the given AVR task.*

Proof. Consider any arbitrary AVR task T_A and the associated DRT task T_D. Based on Definition 9, we need to show that the set of job sequences generated by T_A is a subset of that of T_D. For this purpose, we show that for any $\mathcal{A}$-trace generated by the AVR task, there exists (at least) one $\mathcal{D}$-trace generated by T_D which gets the same job sequence. Take an arbitrary $\mathcal{A}$-trace $\sigma = [(R_1, C_1, \omega_1), \ldots, (R_n, C_n, \omega_n)]$ generated by T_A. Correspondingly, consider a $\mathcal{D}$-trace specified as $\sigma' = [(R_1, C_1, v(\omega_1)), \ldots, (R_n, C_n, v(\omega_n))]$. According to Property 3, relations (23) and (24) hold for σ'. As a result, by the definition of $\mathcal{D}$-trace in Definition 7, σ' constitutes a valid $\mathcal{D}$-trace for T_D. Then, since the job sequence associated to σ and that of σ' are the same, the proof completes. ◀

Putting these together, we show that Algorithm 1 provides a faithful model.

▶ **Lemma 13.** *The DRT graph generated for an AVR task by Algorithm 1 is a faithful model.*

Proof. Based on Lemma 12, the AVR task behavior is included by the obtained DRT graph. According to Lemma 11, this is a sufficient condition for faithfulness defined in Definition 10, which implies faithfulness of the DRT graph. ◀

6 Deriving a tight model

Up to now, we have shown that the constructed DRT graph gives a safe characterization in terms of schedulability analysis. However, it may lead to a pessimistic result. This section provides the criteria under which a DRT task provides a tight workload characterization, i.e., neither pessimistic nor optimistic, for an AVR task.

6.1 Definitions and assumptions

Before tackling tightness of the method, we present two definitions.

▶ **Definition 14** (Point and interval reachability)**.** A speed ω_f is said to be reachable from a speed ω_0, denoted as $\omega_0 \rightsquigarrow \omega_f$, if there exists a valid trajectory from ω_0 to ω_f. Further, a speed interval F is reachable from another speed interval I, denoted as $I \overset{\Delta}{\rightsquigarrow} F$, if $\exists \omega_0 \in I, \exists \omega_f \in F : \omega_0 \rightsquigarrow \omega_f$.

Intuitively, a speed ω_f is reachable from ω_0 if, starting with the speed ω_0, the engine speed can reach ω_f after exactly one rotation, while speed and acceleration constraints are preserved.

▶ **Definition 15** (Tight model)**.** A DRT task T_D is a tight model for an AVR task T_A with respect to a scheduling policy Sch if, when the system is scheduled by Sch,

1. schedulability of T_D entails schedulability of T_A, and
2. unschedulability of T_D entails unschedulability of T_A.

In the following, we assume that the absolute values of the maximum and minimum accelerations are equal, i.e., $\alpha^+ = -\alpha^-$.

6.2 Sufficient conditions for tightness

In this section, we present sufficient conditions under which a DRT task is a tight model for a given AVR task.

▶ **Lemma 16.** *Consider an AVR task, denoted by T_A, and a DRT task, denoted by T_D, which is obtained by Algorithm 1 when applied on T_A with a given speed partitioning. Suppose that*

$$\hat{\sigma}_D \subseteq \sigma_A, \tag{25}$$

where $\hat{\sigma}_D$ denotes the set of those job sequences generated by T_D in which the inter-release separation time between every two successive jobs is exactly equal to the corresponding minimum inter-release time specified by T_D. Then, T_D is a tight model for T_A with respect to EDF and also any fixed priority scheduling algorithm.

Proof. For the proof, we need to verify the two tightness conditions specified in Definition 15. The first one is already shown by Lemma 13. To validate the second property, we notice that $\hat{\sigma}_D$ always provides the critical scheduling instant for both EDF and fixed priority scheduling policies [20]. In other words, if the DRT task is unschedulable, then there exists a job sequence in $\hat{\sigma}_D$ which leads to a deadline miss. In turn, according to (25), such a job sequence is also included by σ_A. As a result, the considered AVR task contains a scenario leading to a deadline miss, which means unschedulability of the task. Thus, the second condition for the tightness also holds. ◀

Using the above lemma, we present conditions under which Algorithm 1 provides a tight DRT task. Take an arbitrary AVR task with a speed range $[\omega_{min}, \omega_{max})$ and a set of M modes. Assume a partitioning of $[\omega_{min}, \omega_{max})$ into K sub-intervals $P = \{[\rho_1, \rho_2), \ldots, [\rho_K, \rho_{K+1})\}$ (with $\rho_1 = \omega_{min}$ and $\rho_{K+1} = \omega_{max}$) for which the following properties hold.

$$\forall i, j \in \{1, \ldots K\} : [\rho_i, \rho_{i+1}) \overset{\Delta}{\rightsquigarrow} [\rho_j, \rho_{j+1}) \implies \rho_{i+1} \rightsquigarrow \rho_{j+1}, \tag{26}$$

$$\forall m \in \{1, \ldots, M\} : \exists i \in \{1, \ldots, K+1\} : \omega_m = \rho_i. \tag{27}$$

The first property states that if there is any speed in the second range which is reachable from one speed in the other range, then the maximum speed of the second range should be

reachable from the maximum speed of the first range. As a result, our method for calculating minimum inter-release times for a vertex in the corresponding DRT graphs, which assumes the maximum feasible speeds (see Fig. 2), does not introduce a pessimism. Property (27) requires that every speed in the boundary of a mode must be a boundary speed in one of the sub-intervals in P.

▶ **Lemma 17.** *The DRT graph obtained by Algorithm 1 for a given AVR task under a given speed partitioning with the properties specified by* (26) *and* (27) *provides a tight model.*

Proof. Let T_D and T_A denote the DRT task and AVR task, respectively, and $\hat{\sigma}_D$ to be as defined as in Lemma 16. According to Lemma 16, it suffices to show that (25) holds. Consider an arbitrary job sequence in $\hat{\sigma}_D$ which corresponds to a $\mathcal{D}$-trace specified as $\sigma' = [(R_1, C_1, v_1), \ldots, (R_n, C_n, v_n)]$. We show that there exists an $\mathcal{A}$-trace as well which produces the same job sequence. Let $[\rho_l(v_i), \rho_u(v_i))$ denote the speed interval associated to the vertex v_i, for $i \in \{1, \ldots, n\}$. Then, consider the $\mathcal{A}$-trace $\sigma = [(R_1, C_1, \rho_u^-(v_1)), \ldots, (R_n, C_n, \rho_u^-(v_n))]$, where $\rho_u^-(v_i)$, for $1 \leq i \leq n$, denotes a value smaller than, but sufficiently close to $\rho_u(v_i)$. Due to the second condition (i.e., (27)), each vertex of the DRT task is associated to only one mode of the AVR task. As a result, the WCET of each vertex is exactly the WCET of that mode. Furthermore, according to the assumed $\mathcal{D}$-trace, there exists at least one speed value in $[\rho_l(v_{i+1}), \rho_u(v_{i+1}))$ which is reachable from $[\rho_l(v_i), \rho_u(v_i))$, for $1 \leq i < n$ (because otherwise, there is no edge from v_i to v_{i+1}, and then, v_i and v_{i+1} cannot be successive jobs). Due to the first assumption above, i.e., (26), this implies that the speed $\rho_u^-(v_{i+1})$ is reachable from $\rho_u^-(v_i)$. The corresponding edge label, that is, the specified minimum inter-release time, in the DRT task is also equal to the time it takes to make one full rotation when the initial speed is $\rho_u^-(v_i)$ and the final speed is $\rho_u^-(v_{i+1})$ (based on Algorithm 1 and Fig. 2). As a result, σ denotes a valid $\mathcal{A}$-trace which can be generated by the AVR task. Hence, $\hat{\sigma}_D \subseteq \sigma_A$. ◀

6.3 An exact DRT model

In this section we provide a method to obtain a speed partitioning satisfying Properties (26) and (27), and as a result, providing a tight DRT task.

Intuitively, in order for a speed partitioning to satisfy (26), all speeds reachable from a boundary speed must be also a boundary speed. Based on this, the set of speeds used to partition the speed range $[\omega_{min}, \omega_{max})$ of an AVR task which contains M modes is defined as:

$$\begin{aligned}\rho = \{\omega_i | 1 \leq i \leq M+1\} \cup \{\Omega_n(\omega_i, \alpha^+) < \omega_{max} | 1 \leq n, 1 \leq i \leq M\} \\ \cup \{\Omega_n(\omega_i, \alpha^-) > \omega_{min} | 1 \leq n, 2 \leq i \leq M+1\}\end{aligned} \tag{28}$$

▶ **Corollary 18.** *Consider the set ρ defined in* (28)*. For any $s \in \rho$, either $\Omega_1(s, \alpha^+) \in \rho$, or $\Omega_1(s, \alpha^+) > \omega_{max}$.*

We then obtain a speed partitioning using the speeds in ρ as interval *boundaries*.

▶ **Lemma 19.** *Consider the set ρ defined in* (28) *sorted in ascending order. Let the interval between any two consecutive speeds of ρ to be a distinct speed range, leading to a set of $|\rho| - 1$ intervals. The speed partitioning obtained in this way satisfies Properties* (26) *and* (27)*.*

Proof. The second property (namely (27)) holds immediately by the definition of ρ. Suppose that the first property does not hold. This means that, $\exists i, j \in \{1, \ldots, |\rho| - 1\}$ such that

Table 1 Mode parameters of the considered AVR task.

i (mode)	1	2	3	4	5	6
ω_i (rpm)	500	1500	2500	3500	4500	5500
C_i (μs)	965	576	424	343	277	246

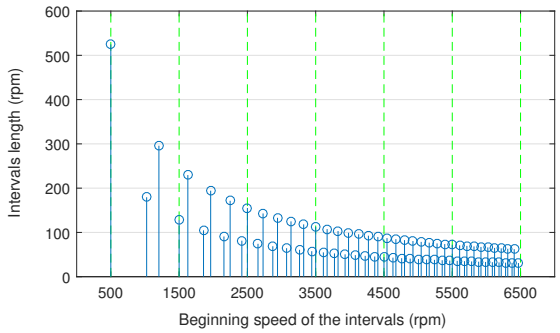


Figure 6 A partitioning of the speed range which leads to a tight characterization.

$[\rho_i, \rho_{i+1}) \overset{\Delta}{\rightsquigarrow} [\rho_j, \rho_{j+1})$ but ρ_{j+1} is not reachable from ρ_{i+1}. Without loss of generality, we assume $\rho_{i+1} \leq \rho_j$. These assumptions imply that

$$\rho_j < \Omega_1(\rho_{i+1}, \alpha^+) < \rho_{j+1}. \tag{29}$$

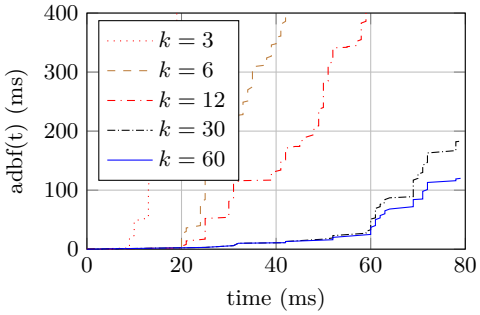
However, according to Corollary 18, $\Omega_1(\rho_{i+1}, \alpha^+)$ should be a boundary speed, which contradicts with (29). ◀

Putting Lemma 17 and Lemma 19 together implies that the DRT graph obtained by using the above speed partitioning is a tight model. It is also worth noting that, based on the definition of ρ in (28) and the definition of Ω_i in (8), the number of speed ranges obtained by the partitioning described in Lemma 19 is $O(M \times \frac{\omega_{max}^2 - \omega_{min}^2}{2\alpha^+})$.

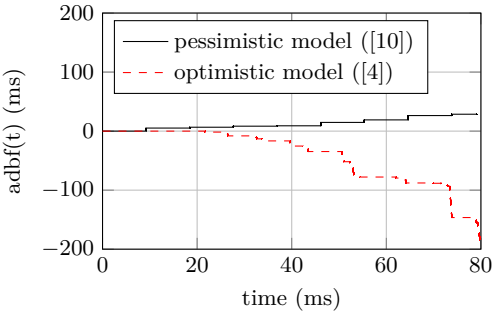
7 Evaluation results

For evaluation, the AVR task adopted in [5] is used. The parameters ω_{min}, ω_{max}, α^-, and α^+ get the same values as in Section 4. Six modes are attributed to the task, described in Table 1 [5]. We first applied the method presented in Section 6 to obtain a speed range partitioning which reveals a DRT model with a tight characterization. The result is a partitioning with 70 sub-intervals, depicted in Fig. 6. The boundary speeds of the intervals are shown on the x-axis. To each boundary speed, a vertical line is associated which shows the length of the interval whose starting point is that boundary speed. In higher speeds, the intervals are typically shorter since the amount of speed evolution during one full rotation is smaller for larger speed values.

In the following, we first examine the accuracy and runtime of the models with different levels of granularity, as well as the models obtained by pessimistic and optimistic approaches, in Subsection 7.1. Next, in Subsection 7.2, we show how the pessimistic and optimistic approaches lead to inaccurate schedulability tests using two counter examples. All the analyses is done using the Python library implemented for timing analysis of DRT tasks [20].



(a) The result for DRT tasks with different granularity in speed partitioning.



(b) Models obtained by the pessimistic and optimistic approaches.

Figure 7 Accuracy of DRT tasks obtained by different methods compared to the exact one.

7.1 Accuracy and run-time performance

As mentioned earlier, the accuracy of a DRT graph can be improved by using more vertices. In our experiments, we extract five models with 3, 6, 12, 30, and 60 vertices. In each case, the speed range $[\omega_{min}, \omega_{max}]$ is divided into equal-length intervals. The result obtained for the task with the tight characterization is used as a reference. This task is obtained by applying our DRT construction method to the speed partitioning of Fig. 6.

To measure the accuracy, we compute the accumulated difference between the demand-bound function (dbf) [17] of each model and the reference one. We define this measure as $adbf(t) = \sum_{i=0}^{t} \left(dbf_k(t) - \hat{dbf}(t) \right)$, where $dbf_k(t)$, for $k \in \{3, 6, 12, 30, 60\}$, denotes dbf of the task with k vertices, and $\hat{dbf}(t)$ denotes that of the reference task model. The results are plotted in Fig. 7a, where the x-axis is the window size for which adbf is computed. As seen, by increasing the number of vertices, the accuracy of the model considerably improves.

We also apply the methods proposed in [10] and [4] to the speed partitioning obtained by our method in Section 6, depicted in Fig. 6. Compared to the tight DRT graph, which is used as a reference, the resultant DRT graphs contain the same number of vertices. However, for the model of [10], i.e., the pessimistic model, minimum inter-release times are equal to or smaller than the values obtained by our method. In contrast, minimum inter-release times for the model of [4], referred to as the optimistic model, are equal to or larger than those of the reference one. The comparison result is shown in Fig. 7b, demonstrating that the pessimistic and optimistic methods for calculation of minimum inter-release times introduce inaccuracies to the analysis, even when applied to an appropriate speed partitioning. We investigate the impact of this inaccuracy on schedulability tests in the next subsection.

We also examine the runtime of the methods. For this, the code is run using the pypy-5.1 compiler on a quad-core processor with 2.40 GHz frequency and a memory of 8 GB. Although the reported results are platform-dependent, they provide a suitable view to the relative complexity and scalability of the analyses. Table 2 gives the time needed for computing the dbf for each model for the interval $[0, 80000]\mu s$. As expected, the runtime grows as the granularity is increased from $k = 3$ vertices to $k = 60$ vertices. Runtime for the tight model is also larger than that of the model with $k = 60$ vertices since it contains 70 vertices.

Additionally, the runtime for the tight model is less than the pessimistic one, and larger than the optimistic one. To reason about this, we point out that DRT graphs in all of these three cases contain the same number of vertices and edges. Meanwhile, to obtain dbf for any time instant t, those paths of the graph that can be traversed within a time interval of

Table 2 Runtime of computing dbf for different models.

DRT model	$k=3$	$k=6$	$k=12$	$k=30$	$k=60$	Tight	Pessimistic	Optimistic
Run-time (s)	0.09	0.5	0.71	1.93	10.36	15.82	16.45	14.01

t need to be enumerated. Hence, smaller minimum inter-release times potentially lead to more number of paths for a specified t. Based on this observation, the pessimistic approach exhibits larger runtime since it reveals smaller minimum inter-release times. On the other hand, the optimistic method consists of equal or larger values of minimum inter-release times, which has implied a shorter running time. Finally, it is worth mentioning that speed partitioning and DRT graph construction steps take negligible time (less than 1%) compared to the time needed for computing dbfs.

7.2 Comparison with respect to schedulability

In order to illustrate the influence of using imprecise models on schedulability tests, we investigate EDF schedulability of a task set which consists of two tasks: an AVR task with the above-described parameters, and a sporadic task whose WCET, relative deadline, and period are denoted by C, D, and P, respectively. We recall that a set of independent real-time tasks τ is EDF schedulable if, and only if, for all $t \geq 0 : \sum_{T \in \tau} dbf_T(t) \leq t$, where $dbf_T(.)$ denotes the dbf of task T [20]. We denote dbf of the tight, pessimistic, and optimistic models, respectively, by $\hat{dbf}(.)$, $\hat{dbf}_P(.)$, and $\hat{dbf}_O(.)$. Further, we denote that of the sporadic task with $dbf_S(.)$. In what follows, the time unit is microsecond.

We first focus on the pessimistic approach, i.e., the method proposed in [10]. For this, the sporadic task parameters are assumed as $C = 8980$, $D = 9210$, and $P = 20000$. To investigate the pessimism, we are interested in an instant t for which $\hat{dbf}_P(t) > \hat{dbf}(t)$. Based on the computed dbfs, we consider $t = 9210\mu s$, for which $\hat{dbf}_P(9210) = 246$, $\hat{dbf}(9210) = 0$, and $dbf_S(9210) = 8980$. When using the pessimistic model, we get $\hat{dbf}_P(t) + dbf_S(t) = 9226 > 9210 = t$, suggesting unschedulability. Besides, it can be verified that, for all $t \geq 0$, $\hat{dbf}(t) + dbf_S(t) \leq t$ holds, meaning that the task set is, in fact, schedulable under EDF.

Next, the optimistic approach proposed in [4] is examined. For this case, we assume the sporadic task parameters as: $C = 25720, D = 26400, P = 50000$. Considering the time instant $t = 26400$, we have $\hat{dbf}_O(26400) = 620$, $\hat{dbf}(26400) = 686$, and $dbf_S(26400) = 25720$. If we use the dbf obtained from the tight model, we get $\hat{dbf}(t) + dbf_S(t) = 26406 > 26400 = t$, which implies unschedulability of the task set. However, writing this condition using the optimistic dbf yields $\hat{dbf}_O(t) + dbf_S(t) = 26340 \leq 26400 = t$. Furthermore, by running a feasibility test using the optimistic dbf, the task set is revealed schedulable, which is not a valid result. The same optimistic assumption has been used in analyses in [5, 6], which can lead to similar unsafe results. To the best of authors knowledge, except for the mentioned studies, other work in the literature have not been impacted by this optimistic assumption.

8 Conclusion and future work

In this paper, we studied engine control applications in which real-time task parameters are functions of the crankshaft speed. For timing analysis of such tasks, we employed the DRT task model [17]. We formulated minimum inter-release time calculation as an optimal control problem where the goal is to find the control input (i.e., the acceleration) that leads to the minimum time for taking a specified change in the angular position. We used a technique from the calculus of variations to solve this minimization problem and determine inter-release

times in the corresponding DRT task. To construct the DRT task, one needs to partition the engine's speed range. In this work, we also proposed a speed range partitioning method which leads to a tight workload characterization in the sense of timing analysis. We showed that our approach provides faithful and tight timing analyses compared to the existing methods.

This work can be extended in multiple directions. In the current study, we focused on independent AVR tasks. The model can be extended to specify tasks which depend on a common rotation source [9, 2]. While the proposed method provides a pessimistic approach to analyze this extended model, it can be improved by employing more expressive models, e.g., the recently proposed synchronous DRT [14] task model. Another direction of extension is considering cyber-physical systems with more general dynamic behavior, for instance, those specified by *hybrid automata* [1]. One can extend our proposed approach to abstract the behavior of such systems for efficient timing analysis, observing potential limitations.

References

1 R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theor. Comput. Sci.*, 138(1):3–34, February 1995. doi:10.1016/0304-3975(94)00202-T.

2 A. Biondi and G. Buttazzo. Engine control: Task modeling and analysis. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 525–530, March 2015. doi:10.7873/DATE.2015.0147.

3 A. Biondi and G. Buttazzo. Real-time analysis of engine control applications with speed estimation. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 193–198, March 2016. doi:10.3850/9783981537079_0273.

4 A. Biondi, G. Buttazzo, and S. Simoncelli. Feasibility analysis of engine control tasks under edf scheduling. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 139–148, July 2015. doi:10.1109/ECRTS.2015.20.

5 A. Biondi, A. Melani, M. Marinoni, M. D. Natale, and G. Buttazzo. Exact interference of adaptive variable-rate tasks under fixed-priority scheduling. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 165–174, July 2014. doi:10.1109/ECRTS.2014.38.

6 A. Biondi, M. Di Natale, and G. Buttazzo. Response-time analysis for real-time tasks in engine control applications. In *International Conference on Cyber-Physical Systems (ICCPS)*, pages 120–129, New York, NY, USA, 2015. ACM. doi:10.1145/2735960.2735963.

7 G. C. Buttazzo, E. Bini, and D. Buttle. Rate-adaptive tasks: Model, analysis, and design issues. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, March 2014. doi:10.7873/DATE.2014.266.

8 R. I. Davis, T. Feld, V. Pollex, and F. Slomka. Schedulability tests for tasks with variable rate-dependent behaviour under fixed priority scheduling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 51–62, April 2014. doi:10.1109/RTAS.2014.6925990.

9 T. Feld and F. Slomka. Sufficient response time analysis considering dependencies between rate-dependent tasks. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 519–524, March 2015. doi:10.7873/DATE.2015.0150.

10 Z. Guo and S. K. Baruah. Uniprocessor EDF scheduling of AVR task systems. In *International Conference on Cyber-Physical Systems (ICCPS)*, pages 159–168. ACM, 2015. doi:10.1145/2735960.2735976.

11 Lino Guzzella and Christopher Onder. *Introduction to Modeling and Control of Internal Combustion Engine Systems*. Springer Science & Business Media, 2009.

12 J. Kim, K. Lakshmanan, and R. Rajkumar. Rhythmic tasks: A new task model with continually varying periods for cyber-physical systems. In *International Conference on Cyber-Physical Systems (ICCPS)*, pages 55–64, April 2012. doi:10.1109/ICCPS.2012.14.

13 Donald E. Kirk. *Optimal Control Theory: An Introduction.* Dover Publications, 1998.

14 M. Mohaqeqi, J. Abdullah, N. Guan, and W. Yi. Schedulability analysis of synchronous digraph real-time tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 176–186, July 2016. doi:10.1109/ECRTS.2016.17.

15 V. Pollex, T. Feld, F. Slomka, U. Margull, R. Mader, and G. Wirrer. Sufficient real-time analysis for an engine control unit. In *International Conference on Real-Time Networks and Systems (RTNS)*, pages 247–254. ACM, 2013. doi:10.1145/2516821.2516838.

16 V. Pollex, T. Feld, F. Slomka, U. Margull, R. Mader, and G. Wirrer. Sufficient real-time analysis for an engine control unit with constant angular velocities. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1335–1338, March 2013. doi:10.7873/DATE.2013.275.

17 M. Stigge, P. Ekberg, N. Guan, and W. Yi. The digraph real-time task model. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 71–80, April 2011. doi:10.1109/RTAS.2011.15.

18 M. Stigge, N. Guan, and W. Yi. Refinement-based exact response-time analysis. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 143–152, July 2014. doi:10.1109/ECRTS.2014.29.

19 M. Stigge and W. Yi. Combinatorial abstraction refinement for feasibility analysis. In *Real-Time Systems Symposium (RTSS)*, pages 340–349, 2013. doi:10.1109/RTSS.2013.41.

20 Martin Stigge. *Real-Time Workload Models : Expressiveness vs. Analysis Efficiency.* PhD thesis, Uppsala University, Division of Computer Systems, 2014.

21 E. Velenis and P. Tsiotras. Optimal velocity profile generation for given acceleration limits: theoretical analysis. In *American Control Conference (ACC)*, pages 1478–1483 vol. 2, June 2005. doi:10.1109/ACC.2005.1470174.

22 H. Zeng and M. Di Natale. Computing periodic request functions to speed-up the analysis of non-cyclic task models. *Real-Time Syst.*, 51(4):360–394, July 2015. doi:10.1007/s11241-014-9209-5.

The Multi-Domain Frame Packing Problem for CAN-FD

Prachi Joshi[1], Haibo Zeng[2], Unmesh D. Bordoloi[3], Soheil Samii[4], S. S. Ravi*[5], and Sandeep K. Shukla[6]

1 **Virginia Tech, Blacksburg, VA, USA**
`prachi@vt.edu`
2 **Virginia Tech, Blacksburg, VA, USA**
`hbzeng@vt.edu`
3 **General Motors, USA**
`unmesh.bordoloi@gm.com`
4 **General Motors, USA; and**
Linköping University, Linköping, Sweden
`soheil.samii@gm.com`
5 **Virginia Tech,Blacksburg, VA, USA; and**
University at Albany – SUNY, NY, USA
`ssravi@vt.edu`
6 **IIT Kanpur, Kanpur, India**
`sandeeps@cse.iitk.ac.in`

Abstract

The Controller Area Network with Flexible Data-Rate (CAN-FD) is a new communication protocol to meet the bandwidth requirements for the constantly growing volume of data exchanged in modern vehicles. The problem of frame packing for CAN-FD, as studied in the literature, assumes a single sub-system where one CAN-FD bus serves as the communication medium among several Electronic Control Units (ECUs). Modern automotive electronic systems, on the other hand, consist of several sub-systems, each facilitating a certain functional domain such as powertrain, chassis and suspension. A substantial fraction of all signals is exchanged across sub-systems. In this work, we study the frame packing problem for CAN-FD with multiple sub-systems, and propose a two-stage optimization framework. In the first stage, we pack the signals into frames with the objective of minimizing the bandwidth utilization. In the second stage, we extend Audsley's algorithm to assign priorities/identifiers to the frames. In case the resulting solution is not schedulable, our framework provides a potential repacking method. We propose two solution approaches: (a) an Integer Linear Programming (ILP) formulation that provides an optimal solution but is computationally expensive for industrial-size problems; and (b) a greedy heuristic that scales well and provides solutions that are comparable to optimal solutions. Experimental results show the efficiency of our optimization framework in achieving feasible solutions with low bandwidth utilization. The results also show a significant improvement over the case when there is no cross-domain consideration (as in prior work).

1998 ACM Subject Classification C.3 [Special-Purpose and Applicaton-Based Systems] Real-Time and Embedded Systems

Keywords and phrases frame packing, CAN-FD, integer linear programming, Audsley's algorithm

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.12

* S. S.Ravi was supported in part by NSF Grants DIBBS ACI-1443054 and BIG DATA IIS-1633028.

1 Introduction

Modern automotive electronic systems consist of several sub-systems, each facilitating a certain functional domain such as powertrain, chassis, suspension, steering, etc. Over the years, there has been a steady increase in the number of messages exchanged among such *sub-systems*, also called *domains* in the paper. A few reasons for this trend are: 1) increase in the number of features enabled by software and electronics; 2) integration of functionalities onto System-on-Chip (SoCs) allowing up-integration of hardware capacity into fewer (but more computationally capable) Electronic Control Units (ECUs); and 3) consolidation of ECUs for cost reduction. The proliferation of such cross-domain traffic can significantly contribute to bandwidth bottlenecks, and as we show in this paper, it leads to a non-trivial optimization problem. In this paper, we propose a frame packing algorithm for multiple sub-systems that are served by CAN-FD (Controller Area Network with Flexible Data-Rate) field buses. To the best of our knowledge, this is the first attempt to formulate and propose a solution to what we term as the problem of *multi-domain frame packing for CAN-FD*.

Since its development in the 1990s, CAN (Controller Area Network) has attracted a significant amount of research from the real-time systems community. The CAN protocol adopts a collision detection and resolution scheme, where the message to be transmitted is chosen according to its identifier. When multiple nodes need to transmit over the same bus, the message with the lowest identifier is selected for transmission. This arbitration protocol allows encoding of the message priority into the identifier field and the implementation of priority-based scheduling. The analysis of the CAN message response time [22, 5] was derived using an analogy to the results on CPU scheduling, providing an exact evaluation and a safe approximation of the worst-case message response times.

In recent years, automotive features have been growing, thereby demanding an increase in bandwidth requirements of the communication network. In order to bridge the gap between CAN and other higher data rate communication protocols (such as TTEthernet, MOST150, etc.), two major improvements were added to CAN to develop CAN-FD [9]: 1) the increase of bit-rate (up to 8 Mbps); and 2) the increase of payload sizes (up to 64 bytes). The physical layer of CAN was unchanged: it still uses a bitwise arbitration method of contention resolution based on message identifiers.

Related Work. As mentioned earlier, the frame packing problem has been considered in the literature only for a single domain in CAN-FD. Bordoloi and Samii [2] present a dynamic programming approach for packing the signals followed by a priority assignment step. Urul's thesis [23] points out that schedulability of frames can be improved by packing same period signals in each frame. Di Natale et al. [16] present a single-step Integer Linear Programming (ILP) formulation to achieve both optimal bandwidth utilization and schedulability. However, its applicability is limited to medium-size problems. In [25], the authors map signals to frames on CAN as part of their task allocation and priority assignment problem to optimize end-to-end latency using an MILP.

The frame packing problem is related to the classical bin packing problem (BPP) which is known to be NP-hard. For CAN-FD, a particularly relevant subclass of BPP is the *variable-sized bin packing problem* (VSBPP). While the classical bin packing problem has been studied extensively (e.g., [8]), VSBPP has received relatively less attention. Friesen and Langston [7] propose and formally analyze the performance of three heuristics for VSBPP. Murgolo [15] presents a polynomial-time approximation scheme for VSBPP. We note that the approximation algorithms for VSBPP cannot be directly applied to the frame packing

problem for CAN-FD since the goal of the latter problem is to minimize bandwidth utilization instead of the number of bins (frames). In addition, the frame packing problem must consider both the size and the period of each signal.

For standard CAN, both [17] and [20] present frame packing approaches inspired by the *next fit decreasing* heuristic for BPP. The difference is that the algorithm in [17] sorts the signals according to their periods, while the one in [20] sorts them based on their deadlines. Saket and Navet [19] present a frame packing heuristic which sorts the signals by their bandwidth utilization and then packs this list of sorted signals alternately from both sides of the list (to increase the chances of signals with similar periods to be packed together).

The frame packing problem has also been considered under other communication protocols that are time-triggered (such as FlexRay static segment [13, 24, 21, 11, 3]) or mixed event/time-triggered [18]. The nature of these communication protocols, and thus the frame packing problem, is very different from that for CAN and CAN-FD. Hence, the corresponding approaches and results are not directly applicable here.

Contributions. The frame packing problem in CAN and CAN-FD has been addressed in the literature [2, 23, 16]. However, unlike our paper, these references consider only a single domain. Even for a single domain, the problem is already challenging: the above references have pointed out its relationship to the bin packing problem which is known to be NP-hard.

In this paper, we study the multi-domain frame packing problem for CAN-FD, and develop a two-stage optimization procedure. In the first stage, we propose an ILP based approach to generate an optimal solution for frame packing as well as a heuristic that scales to large problem sizes. Our ILP and heuristic approaches capture the details of inter-domain communication and gating over multiple CAN-FD networks. In the second stage, we propose an extension to Audsley's algorithm [1] for optimal priority assignment with multi-domain frames. In case the priority assignment does not lead to a feasible solution, we provide an effective strategy to re-pack the frames. We conduct experiments on synthetic systems (whose characteristics are close to real systems) and show that our heuristic runs extremely fast compared to the computationally expensive (in terms of both time and memory) ILP and yet returns solutions that are on average within 3% of those produced by the ILP in terms of bandwidth utilization per domain. Our experiments also show that the repacking strategy is effective in that it often leads to schedulable solutions. Compared to the approach without cross-domain consideration, our approach can typically save 6%–10% bandwidth utilization per domain.

The rest of the paper is organized as follows. Section 2 provides a brief overview of the CAN-FD protocol. Section 3 defines the multi-domain frame packing problem. Section 4 provides an overview for the two-stage iterative framework. Section 5 presents the approach using ILP formulation and Section 6 describes the greedy heuristic algorithm. Section 7 provides the experimental results and compares the ILP and heuristic approaches. Finally, Section 8 summarizes our contributions and presents some concluding remarks.

2 CAN-FD Overview

In this section, we briefly describe the main features of CAN-FD. The CAN-FD frame format is shown in Figure 1. For a more detailed description, readers are referred to [2]. Like CAN, a **dominant** bit is a logical 0 and a **recessive** bit is a logical 1. As in the figure, a CAN-FD frame is partitioned into two phases: arbitration phase and data phase.

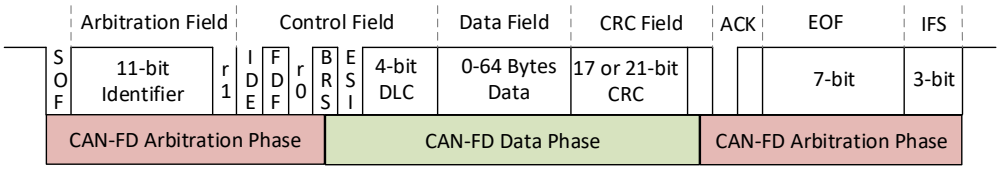


Figure 1 CAN-FD Frame Format (from [9]).

Arbitration Phase. The arbitration phase in the CAN-FD frame contains the following fields: SOF (Start Of Frame), arbitration, part of the control field, ACK (Acknowledgment), EOF (End OF Frame), and IFS (Inter-Frame Space). The 11-bit (or 29-bit in case of extended format) identifier represents the priority of the frame: the lower the value of the identifier, the higher the priority. The arbitration for transmission happens as follows. During the idle state of the bus, all the nodes with some ready frames send the 11-bit identifier after the SOF bit. During the transmission of the identifier bits, if a node transmits a recessive bit but finds a dominant bit on the bus, it stops transmission due to the presence of a higher priority node contesting for transmission. In the end, the node with the highest priority message wins the arbitration and continues the transmission.

The transmission of bits in the arbitration phase occurs at the arbitration bit-rate, and the duration of transmission for each bit is denoted as t_a. For example, if the arbitration rate is chosen as 500 Kbps, then $t_a = 2\mu s$.

Data Phase. The BRS (Bit-Rate Switch) bit is one of the additions to the CAN-FD frame format. It is used to decide whether the bit-rate in the data phase is the same as that of the arbitration phase (BRS = 0) or it switches to the increased bit rate (BRS = 1). Since our focus is on CAN-FD, we consider the BRS bit in the frames to be recessive (i.e., BRS = 1). At the increased rate of data transmission, each bit transmission occurs with a duration denoted by t_d. For example, if the data rate is chosen as 2 Mbps, $t_d = 0.5\mu s$. The 4-bit DLC (data-length code) field specifies the payload size (in bytes) of the data field. CAN-FD offers 16 distinct payload sizes: 0 through 8, 12, 16, 20, 24, 32, 48 and 64 bytes.

The data field is followed by the Cyclic Redundancy Check (CRC) field, which has 17 bits for payloads up to 16 bytes, and 21 bits otherwise. The CRC delimiter bit (recessive) is transmitted next. After this, the bit rate is changed back to that of the arbitration phase.

Transmission Time. The worst-case transmission time (WCTT) of a CAN-FD frame is a function of its payload size (i.e., the size of the data field) and the data rates. As in [2], if p is the payload size (in bytes) of a CAN-FD frame, its WCTT is given by:

$$\text{WCTT}(p) \;=\; 32\,t_a + \Big(28 + 5\left\lceil\frac{p-16}{64}\right\rceil + 10p\Big)t_d\,. \tag{1}$$

In this work we have assumed the arbitration and data rates to be same for all the domains, however our approach can be easily adapted to the scenario where each domain/network has a different bit-rate. The bit-rates affect the WCTT expression (Equation 1), and therefore for the latter case we can compute the WCTT for each domain and use it in the bandwidth calculation corresponding to the domain. Similarly the schedulability analysis can be updated with the inclusion of the appropriate WCTT expression for determining the response time for a frame on each domain.

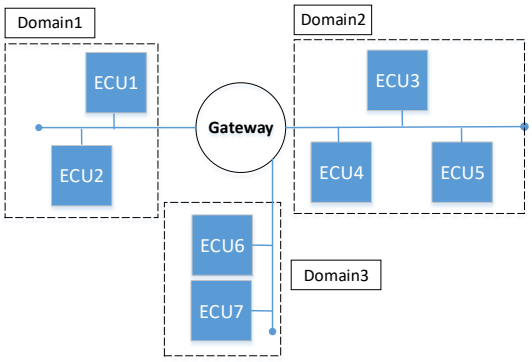


Figure 2 A multi-domain CAN-FD architecture.

Table 1 Parameters of each signal and frame.

Notation	Significance
$t(\sigma)$	Period of signal σ
$d(\sigma)$	Deadline of signal σ
$p(\sigma)$	Size (in bytes) of signal σ
$\delta(\sigma)$	Domains of signal σ

Notation	Significance
$T(\gamma)$	Period of frame γ
$D(\gamma)$	Deadline of frame γ
$P(\gamma)$	Payload size (in bytes) of frame γ
$\Delta(\gamma)$	Set of domains of signals packed in frame γ
$\mathcal{S}(\gamma)$	Set of signals packed in frame γ
$C(\gamma)$	Worst-case transmission time (WCTT) of frame γ
$\pi(\gamma)$	Priority level of frame γ

3 Problem Definition

We assume a network topology where several CAN-FD sub-systems, typically serving different domains, are connected to a central gateway. We use the terms "domain" and "sub-system" interchangeably. The ECUs in each domain generate signals which must be packed into frames and transmitted to their destination domains. Each domain uses a CAN-FD bus for data communication. The gateway is responsible for forwarding the frames to their respective destination domains *without repacking or reassigning frame identifiers*. Such an architecture is relevant in the automotive industry [10]. Figure 2 provides an example of a system where 3 domains are connected by a gateway.

Table 1 summarizes the parameters of each signal and frame. In the following, we define the problem and review the schedulability analysis for CAN-FD.

Problem Description: Let $\mathbb{D} = \{\Delta_1, \Delta_2, \ldots, \Delta_{|\mathbb{D}|}\}$ denote the set of *domains*. Let n_i denote the number of ECUs in domain Δ_i, $1 \leq i \leq |\mathbb{D}|$. The jth ECU from domain Δ_i is denoted by $\psi_{i,j}$. The set of signals generated by ECU $\psi_{i,j}$ is represented by $\mathcal{S}(\psi_{i,j}) = \{\sigma_{i,j}^k \mid k = 1, \ldots, |\mathcal{S}(\psi_{i,j})|\}$. Each signal $\sigma \in \mathcal{S}(\psi_{i,j})$ is specified as a quadruple $\langle\, t(\sigma), d(\sigma), p(\sigma), \delta(\sigma)\,\rangle$, whose components denote respectively the period, deadline, size (in bytes) and domains (including the source and destinations) of signal σ.

The output of the multi-domain frame packing problem is a set of frames $\Gamma = \{\gamma_1, \gamma_2, \ldots\}$ satisfying *all* of the following conditions.

1. Each signal σ is placed in exactly one frame γ.
2. For each frame γ, all the signals in γ are from the *same* ECU, and the periods of all the signals in γ are **harmonic**[1] (i.e., $\forall \sigma_i, \sigma_j \in \gamma$, there exists an integer k such that either $t(\sigma_i) = k \cdot t(\sigma_j)$ or $t(\sigma_j) = k \cdot t(\sigma_i)$).
3. The sum of the sizes of all signals in a frame γ is at most the payload size of γ.
4. The packed frames are schedulable in all the domains in which they are transmitted.

Each frame γ is characterized by a tuple $\langle \mathcal{S}(\gamma), \Delta(\gamma), T(\gamma), D(\gamma), P(\gamma), C(\gamma), \pi(\gamma) \rangle$, where $\mathcal{S}(\gamma)$ is the set of signals packed into γ and $\Delta(\gamma)$ is the set of domains of the signals in γ. The quantities $T(\gamma)$, $D(\gamma)$, $P(\gamma)$, and $C(\gamma)$ are respectively the period, deadline, payload size (in bytes), and WCTT of the frame γ. Given $\mathcal{S}(\gamma)$, the other parameters of the frame γ are determined as follows.

- $\Delta(\gamma) = \bigcup_{\sigma \in \mathcal{S}(\gamma)} \delta(\sigma)$; i.e., the set of domains of γ is the union of those for all the signals in γ.
- $T(\gamma) = \gcd\{t(\sigma) \,:\, \sigma \in \mathcal{S}(\gamma)\}$; i.e., the period of γ is the greatest common divisor (gcd) of the periods of the signals in γ.
- $D(\gamma) = \min\{d(\sigma) \,:\, \sigma \in \mathcal{S}(\gamma)\}$; i.e., the deadline of γ is the smallest deadline among the signals in γ.
- $P(\gamma) \geq \sum_{\sigma \in \mathcal{S}(\gamma)} p(\sigma)$; i.e., the payload of γ is large enough to contain its constituent signals. The CAN-FD standard [9] restricts $P(\gamma)$ to be one of the following values: 0 through 8, 12, 16, 20, 24, 32, 48 and 64 bytes.
- The WCTT $C(\gamma)$ of a frame γ is determined by Equation (1), with the variable p being replaced by $P(\gamma)$.
- $\pi(\gamma)$ represents the unique priority (across all domains) assigned to frame γ.

The bandwidth utilization $U(\gamma)$ of frame γ is defined as

$$U(\gamma) \;=\; \frac{C(\gamma)}{T(\gamma)}\,. \tag{2}$$

The objective is to minimize the total bandwidth utilization over all the domains. This is motivated by extensibility to accommodate possible future functions [2].

CAN-FD Schedulability: The schedulability analysis for CAN-FD follows that of CAN [5], where the worst-case response time is always inside the busy period. The busy period of priority level-i is a contiguous interval of time that starts at the critical instant, during which any frame of priority lower than γ_i is unable to win arbitration. The length of the busy period $L(\gamma_i)$ and the index $q^{\max}(\gamma_i)$ of the last instance are calculated as

$$L(\gamma_i) = B(\gamma_i) + \sum_{j \in hp(i) \bigcup \{i\}} \left\lceil \frac{L(\gamma_i)}{T(\gamma_j)} \right\rceil C(\gamma_j), \qquad q^{\max}(\gamma_i) = \left\lceil \frac{L(\gamma_i)}{T(\gamma_i)} \right\rceil \tag{3}$$

[1] Note that our approach is valid even without this condition.

where $hp(i)$ is the set of frames with priority higher than γ_i, and $B(\gamma_i)$ is the *blocking time*, i.e., the maximum time spent on waiting for the transmission of a lower priority message already on the bus when γ_i becomes ready.

The response time $R(\gamma_{i,q})$ of the q-th instance $\gamma_{i,q}$ in the busy period is given by

$$R(\gamma_{i,q}) = w(\gamma_{i,q}) - (q-1)T(\gamma_i) + C(\gamma_i) \tag{4}$$

where q ranges from 1 to the last instance $q^{\max}(\gamma_i)$ of γ_i inside the busy period. The worst-case queuing delay $w(\gamma_{i,q})$ for the q-th instance in the busy period is

$$w(\gamma_{i,q}) = B(\gamma_i) + (q-1)C(\gamma_i) + \sum_{j \in hp(i)} \left\lceil \frac{w(\gamma_{i,q})}{T(\gamma_j)} \right\rceil C(\gamma_j)\,. \tag{5}$$

In Equation (5), $w(\gamma_{i,q})$ appears on both sides. However, the right hand side is a monotonic non-decreasing function of $w(\gamma_{i,q})$. Hence, $w(\gamma_{i,q})$ can be solved using the iterative procedure defined by the equation below.

$$w^{n+1}(\gamma_{i,q}) = B(\gamma_i) + (q-1)C(\gamma_i) + \sum_{j \in hp(i)} \left\lceil \frac{w^n(\gamma_{i,q})}{T(\gamma_j)} \right\rceil C(\gamma_j)\,. \tag{6}$$

The calculation can start with an initial value of $w^0(\gamma_{i,q}) = B(\gamma_i) + (q-1)C_i$, and stop when $w^{n+1}(\gamma_{i,q}) = w^n(\gamma_{i,q})$ or $w^{n+1}(\gamma_{i,q}) - (q-1)T(\gamma_i) + C(\gamma_i) > D(\gamma_i)$, the latter condition indicating that γ_i is unschedulable. The worst-case response time of γ_i, denoted by $R(\gamma_i)$, is the maximum among all its instances in the busy period; that is,

$$R(\gamma_i) = \max_{q=1,\ldots,q^{\max}(\gamma_i)} \{R(\gamma_{i,q})\} \tag{7}$$

4 Overview of the Two-Stage Optimization Procedure

The multi-domain frame packing problem for CAN-FD is NP-hard. This follows directly from the NP-hardness of the special case of single-domain frame packing problem [2]. To cope with this complexity, we consider an optimization procedure that consists of two stages, as illustrated in Figure 3. In the first stage, we try to find a signal-to-frame packing with minimum total bandwidth utilization. In the second stage, given the signal-to-frame packing, we perform priority assignment to all the frames such that the response time of each frame falls within its deadline. We choose to minimize the total bandwidth in the first stage for two reasons: one is that this is also the overall objective, the other is that smaller bus bandwidth utilization generally leads to better schedulability. However, the latter is not always the case. When a candidate frame packing produced by Stage 1 is not schedulable, all the frames or a subset thereof are repacked, this time focusing on improving schedulability. The two-stage procedure iterates until a feasible solution is found or after a certain number of iterations of repacking have been completed.

As discussed in subsequent sections, we present two instantiations of the optimization procedure. One instantiation formulates the problem in the first stage as an integer linear program (ILP) and leverages existing solvers to find an optimal solution (i.e., one with minimum total bandwidth utilization). In case of unschedulability, the second stage iteratively produces another packing by sacrificing optimality by a certain amount, with additional constraints in the ILP model. As will be seen in Section 5, the ILP formulation is somewhat intricate due to the discontinuous nature of the frame sizes available under CAN-FD. Due to the number of constraints in the ILP formulation, this approach does not scale to large

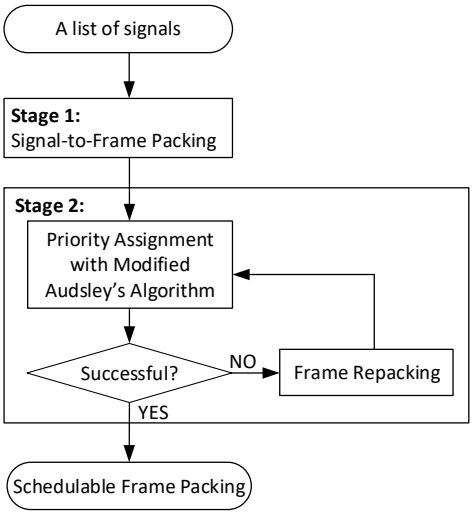


Figure 3 The two-stage optimization procedure.

systems. The second instantiation of the procedure in Figure 3 uses a fast heuristic in the first stage. Based on the observation that the first stage, namely the problem of signal-to-frame packing, is related to the bin packing problem, we develop a bandwidth-based best-fit (greedy) approach, where each signal is packed into a candidate frame that minimizes the total bandwidth utilization over all the domains. The second stage directly repacks a selected list of frames in case of unschedulability.

The problem of frame priority assignment can be solved efficiently. We propose a modified version of the Audsley's algorithm [1] that only needs to check a quadratic number of candidate priority assignments. Similar to Audsley's algorithm, it iteratively picks a frame that can be assigned a particular priority level starting from the lowest priority. However, when choosing a candidate frame, it should guarantee that assigning the priority does not violate the schedulability in any domain to which the frame will be transmitted. Here, the priority order of frames remains the same across all domains, as the gateway is assumed to forward the frames without repacking or reassigning frame identifiers.

5 ILP-based Approach

5.1 ILP formulation for Signal-to-Frame Packing

Since each frame may only contain signals sent by the same ECU and we do not consider schedulability in the first stage, it is sufficient to perform frame packing for each ECU separately. Hence, we present the ILP formulation considering the set of signals $\mathcal{S}(\psi_{i,j})$ generated by ECU $\psi_{i,j}$. The total bandwidth utilization is the sum of the utilization values over all the ECUs.

We generate a set of virtual frames $\Gamma_{i,j} = \{\gamma_l \mid l = 1, ..., |\mathcal{S}(\psi_{i,j})|\}$, one for each signal in $\mathcal{S}(\psi_{i,j})$. Thus, $\Gamma_{i,j}$ consists of the maximum number of frames that could be used for packing the signals in $\mathcal{S}(\psi_{i,j})$; a packing may use only a subset of $\Gamma_{i,j}$. Each virtual frame $\gamma_l \in \Gamma_{i,j}$ is represented as a tuple $\langle T(\gamma_l), D(\gamma_l), P(\gamma_l)\rangle$, which specifies respectively the period, deadline and size of the frame. Here, we fix the period of each frame to be the period of its corresponding signal. Thus, $\forall l$, $T(\gamma_l) = t(\sigma_l)$. However, the deadline $D(\gamma_l)$ and size $P(\gamma_l)$

depend on the signals packed in γ_l. We use a binary *parameter* to denote whether a signal shall be transmitted in a particular domain.

$$Y_{k,e} = \begin{cases} 1 & \text{if the destination of } \sigma_k \text{ is in domain } \Delta_e \\ 0 & \text{otherwise.} \end{cases}$$

Note that this information is available as part of the input. Thus, $Y_{k,e}$ is *not* a variable in the ILP formulation. We define a binary (decision) variable $x_{k,l}$ to indicate the mapping of signals to frames:

$$x_{k,l} = \begin{cases} 1 & \text{if signal } \sigma_k \text{ is packed into frame } \gamma_l \\ 0 & \text{otherwise.} \end{cases}$$

Each signal should be assigned to one and only one frame:

$$\forall k: \quad \sum_{l=1}^{|\mathcal{S}(\psi_{i,j})|} x_{k,l} = 1 \tag{8}$$

The period of a frame should be a divisor of the period of any signal assigned to the frame:

$$\forall k, l \text{ such that } t(\sigma_k) \bmod T(\gamma_l) \neq 0: \quad x_{k,l} = 0 \tag{9}$$

If two signals have non-harmonic periods, they should not be packed in the same frame:

$$\forall k, m \text{ such that } t(\sigma_k) \geq t(\sigma_m) \bigwedge t(\sigma_k) \bmod t(\sigma_m) \neq 0, \ \forall \gamma_l: \ x_{k,l} + x_{m,l} \leq 1 \tag{10}$$

Since some frames may not be assigned any signals during the packing, we must ensure that they are not taken into account while computing the bandwidth utilization. To do this, we use a binary variable ρ_l to indicate if γ_l contains any signals. Hence,

$$\frac{\sum_{k=1}^{|\mathcal{S}(\psi_{i,j})|} x_{k,l}}{|\mathcal{S}(\psi_{i,j})|} \leq \rho_l \bigwedge \rho_l \leq \sum_{k=1}^{|\mathcal{S}(\psi_{i,j})|} x_{k,l} \tag{11}$$

For each frame γ_l, we introduce a variable z_l that represents the sum of the sizes (in bytes) of the signals packed in γ_l:

$$z_l = \sum_{k=1}^{|\mathcal{S}(\psi_{i,j})|} x_{k,l} \cdot p(\sigma_k) \tag{12}$$

The maximum size of a frame is 64 bytes. Hence, the total size of the signals assigned to a frame should be no more than 64 bytes:

$$\forall l: \ z_l \leq 64 \tag{13}$$

As described in Section 2, CAN-FD allows 16 different frame payload sizes (0 through 8, 12, 16, 20, 24, 32, 48 and 64 bytes). Hence, the size $P(\gamma_l)$ of any frame γ_l can be modeled as a discontinuous function defined by

$$P(\gamma_l) = \begin{cases} z_l, & 0 \leq z_l \leq 8 \\ 12, & 8 < z_l \leq 12 \\ 16, & 12 < z_l \leq 16 \\ 20, & 16 < z_l \leq 20 \\ 24, & 20 < z_l \leq 24 \\ 32, & 24 < z_l \leq 32 \\ 48, & 32 < z_l \leq 48 \\ 64, & 48 < z_l \leq 64 \end{cases}$$

We define eight new binary variables λ_1, λ_2, …, λ_8, which determine the ranges of the z_l variables.

$$\begin{cases} z_l \leq 8 + M(1-\lambda_1) & \\ z_l \leq 12 + M(1-\lambda_2) & \bigwedge\ z_l + M(1-\lambda_2) > 8 \\ z_l \leq 16 + M(1-\lambda_3) & \bigwedge\ z_l + M(1-\lambda_3) > 12 \\ z_l \leq 20 + M(1-\lambda_4) & \bigwedge\ z_l + M(1-\lambda_4) > 16 \\ z_l \leq 24 + M(1-\lambda_5) & \bigwedge\ z_l + M(1-\lambda_5) > 20 \\ z_l \leq 32 + M(1-\lambda_6) & \bigwedge\ z_l + M(1-\lambda_6) > 24 \\ z_l \leq 48 + M(1-\lambda_7) & \bigwedge\ z_l + M(1-\lambda_7) > 32 \\ z_l \leq 64 + M(1-\lambda_8) & \bigwedge\ z_l + M(1-\lambda_8) > 48 \end{cases} \tag{14}$$

where M is a large enough constant. Hence, the size of a frame can be expressed as

$$P(\gamma_l) = \lambda_1 \cdot z_l + 12\lambda_2 + 16\lambda_3 + 20\lambda_4 + 24\lambda_5 + 32\lambda_6 + 48\lambda_7 + 64\lambda_8\,. \tag{15}$$

However, there is a product term, namely $\lambda_1 \cdot z_l$, in Equation (15). This can be linearized by introducing a new variable v_l as follows.

$$v_l = \lambda_1 \cdot z_l \;\;\Rightarrow\;\; v_l \;\leq\; z_l + M(1-\lambda_1) \bigwedge\; z_l \;\leq\; v_l + M(1-\lambda_1) \bigwedge\; v_l \;\leq\; M \cdot z_l\,. \tag{16}$$

Therefore, Equation (15) can be rewritten as the following linear constraint:

$$P(\gamma_l) = v_l + 12\lambda_2 + 16\lambda_3 + 20\lambda_4 + 24\lambda_5 + 32\lambda_6 + 48\lambda_7 + 64\lambda_8\,. \tag{17}$$

To calculate the WCTT of frame γ_l using Equation (1), we note that the ceiling function $\left\lceil \frac{P(\gamma_l)-16}{64} \right\rceil$ can only take on two values: 0 if $P(\gamma_l) \leq 16$ and 1 otherwise. Hence, we introduce a binary variable u_l to represent it. The constraints on u_l are as follows:

$$P(\gamma_l) + M(1-u_l) \;>\; 16 \bigwedge\; P(\gamma_l) \;\leq\; 16 + M \cdot u_l\,. \tag{18}$$

Now, the expression for WCTT becomes

$$C(\gamma_l) \;=\; 32\,t_a + (28 + 5u_l + 10P(\gamma_l))\,t_d\,. \tag{19}$$

The total bandwidth utilization for all the frames for an ECU $\psi_{i,j}$ in the CAN-FD network can be expressed as follows:

$$\sum_{l}^{|\mathcal{S}(\psi_{i,j})|} \Big[\rho_l \cdot \frac{C(\gamma_l)}{T(\gamma_l)} + \sum_{e \neq i} \Big(\eta_{l,e} \cdot \frac{C(\gamma_l)}{T(\gamma_l)}\Big)\Big] \tag{20}$$

where the first part $\rho_l \cdot \frac{C(\gamma_l)}{T(\gamma_l)}$ corresponds to the bandwidth utilization over the source domain Δ_i, and the second part $\sum_{e \neq i} \left(\eta_{l,e} \cdot \frac{C(\gamma_l)}{T(\gamma_l)}\right)$ corresponds to the bandwidth utilization over all the destination domains. In Equation (20), $\eta_{l,e}$ is a binary variable to determine whether the frame γ_l has any signal with a destination in domain Δ_e. Using the binary parameter $Y_{k,e}$ defined earlier, $\eta_{l,e}$ can be defined as

$$\eta_{l,e} = \begin{cases} 1 & \text{if } \sum_{k=1}^{|\mathcal{S}(\psi_{i,j})|}(x_{k,l} \cdot Y_{k,e}) \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

The linear constraints that enforce the definition of $\eta_{l,e}$ are as follows:

$$\frac{\sum_{k=1}^{|\mathcal{S}(\psi_{i,j})|}(x_{k,l} \cdot Y_{k,e})}{|\mathcal{S}(\psi_{i,j})|} \;\leq\; \eta_{l,e} \;\leq\; 1 \bigwedge\; \eta_{l,e} \;\leq\; \sum_{k=1}^{|\mathcal{S}(\psi_{i,j})|} (x_{k,l} \cdot Y_{k,e})\,. \tag{21}$$

Algorithm 1 Modified Audsley's Algorithm for Multi-Domain CAN-FD

```
1:  procedure AudsleyMultiDomain (Γ)
2:      Let N = |Γ|, Create a list Q containing all the frames in Γ
3:      for π = N downto 1 do
4:          for each frame γ ∈ Q do
5:              flag_found = FALSE
6:              for each Δ_i ∈ Δ(γ) do
7:                  R(γ) = ComputeResponseTime (γ, Δ_i, Γ)
8:              if γ is schedulable in all Δ_i ∈ Δ(γ) then
9:                  Assign priority level π to γ
10:                 Remove γ from Q
11:                 flag_found = TRUE
12:                 break
13:         if flag_found is FALSE then
14:             Report unschedulability and return
15:     Report schedulability
```

The objective is to minimize the total bandwidth utilization of all the frames:

$$\min \sum_{i}^{|\mathbb{D}|} \sum_{j}^{n_i} \sum_{l}^{|\mathcal{S}(\psi_{i,j})|} \left[\rho_l \cdot \frac{C(\gamma_l)}{T(\gamma_l)} + \sum_{e \neq i} \left(\eta_{l,e} \cdot \frac{C(\gamma_l)}{T(\gamma_l)} \right) \right]. \tag{22}$$

In Equation (22), $\rho_l \cdot \frac{C(\gamma_l)}{T(\gamma_l)}$ and $\eta_{l,e} \cdot \frac{C(\gamma_l)}{T(\gamma_l)}$ are both a product of a binary variable and a real variable. They can be linearized in a manner similar to that of Equation (16).

5.2 Modified Audsley's Algorithm for Priority Assignment

In order to assign priority identifiers to all the frames, we extend Audsley's algorithm [1] to the multi-domain case (Algorithm 1). The input to the algorithm is the set of frames Γ for all the domains. Similar to Audsley's algorithm, priority levels are assigned iteratively to all the frames starting from lowest to highest (Lines 3–15). At each iteration, a priority level is assigned to the first frame γ that satisfies the schedulability constraints over all the domains belonging to $\Delta(\gamma)$ (Lines 8–12). If a priority level cannot be assigned to any of the frames (i.e., flag_found is FALSE), the algorithm reports unschedulability (Lines 13–14). If all the frames are assigned a unique priority, the algorithm is successful in finding a schedulable priority assignment.

Using the approach in [6], it can be easily shown that the schedulability of a multi-domain frame (i.e., whether it meets the deadline requirement in all its domains) satisfies all the three conditions which are necessary and sufficient to provide an optimal priority assignment. Hence, our extension to Audsley's algorithm for the multi-domain CAN-FD system is optimal for finding a schedulable priority assignment.

5.3 Handling Infeasibility

Although in principle our frame packing scheme supports schedulability (since it minimizes bandwidth utilization which indirectly helps to reduce network traffic), there are cases when the modified Audsley's algorithm returns infeasibility.

In the case of ILP, when we encounter infeasibility, we call the solver again after relaxing the optimal value of the objective function (by doubling the optimality gap in each iteration) and setting a time limit of one hour for each iteration. We report infeasibility if no feasible priority assignment is found even after a given number of iterations.

6 Greedy Algorithm-based Approach

The ILP approach discussed in the previous section provides an optimal packing of the signals into frames with respect to bandwidth utilization. However, due to its exponential time complexity, it does not scale well to large sets of signals. Therefore, we propose a greedy heuristic (Algorithm 2) for the frame packing step. The heart of the algorithm presents the steps for packing the signals from one ECU; the outermost loop ensures that the steps are iterated over all the ECUs.

6.1 Description of the Heuristic for Signal-to-Frame Packing

The algorithm first sorts the input signals for each ECU (Line 3 in Algorithm 2) on the basis of a parameter such as the period, size or the input bandwidth utilization (which is given by the size/period) of signals. It then uses a Bandwidth Best-Fit approach to pack the signals into frames as follows. Starting from the first signal, each signal is placed in a frame that minimizes the total bandwidth utilization of the system (*over all the domains*). The steps shown in lines 6 and 7 create a new frame and add the signal to it. To obtain the bandwidth utilization for a frame, we use Equation (2) to compute the utilization over each of its destination domains and then take their sum. The total bandwidth utilization of the system is the sum of the bandwidth utilization over all the frames (Equation (20)). Further, lines 8–14 compute and store the total bandwidth utilization by temporarily adding the signal to an existing frame. Before a signal is assigned to an existing frame, the "if(σ_k can be added to F_j)" condition (Line 9 in Algorithm 2) checks (i) whether the frame can accommodate the new signal (i.e., the total size of all the signals in the frame is at most 64); and (ii) whether the period of the new signal is harmonic with the periods of the other signals in that frame. The steps in lines 15 and 16 decide whether it is beneficial to add the current signal to a new frame or to one of the existing frames. The output of the algorithm is a list of frames (Γ) which stores the frames created in each step.

The quality of the packing depends on the sorting criterion used in Line 3. In our experiments, we compare different sorting methods using each of the above parameters (i.e., period, size and size/period) in both increasing and decreasing orders.

Time Complexity Analysis: For each ECU ψ_i, we show that the above heuristic runs in $O(s^2 f)$ time, where $s = |S_i|$ is the number of signals for the ECU and f is the number of frames in the resulting packing. To begin with, sorting the set $\mathcal{S}(\psi_i)$ can be done in $O(s \log s)$ time. Now, for each signal $\sigma \in \mathcal{S}(\psi_i)$, the time for finding the best placement into a frame can be estimated as follows. Let $\Gamma(\psi_i) = \{\gamma_1, \gamma_2, \ldots, \gamma_r\}$ denote the current set of frames when σ is considered. Creating a new frame containing just σ can be done in $O(1)$ time. As mentioned above, testing whether σ can be added to a frame γ_i involves two checks involving the size of the frame and the harmonicity of periods of the signals currently in the frame. It is easy to see that each of these checks can be done in time $O(|\gamma_i|)$, where $|\gamma_i|$ denotes the number of signals in γ_i. Thus, the total time for checking whether σ can be added to each of the existing frames is $O(\sum_{i=1}^{r} |\gamma_i|) = O(s)$, since all the frames together contain at most s signals. For each placement of σ in a frame, it is also easy to see that computing the

Algorithm 2 Greedy Algorithm

```
1: procedure GREEDY-BW-BEST-FIT (Ψ, S)
2:     for each ECU ψ_i ∈ Ψ do
3:         Sort(S(ψ_i))
4:         Number of frames n = 0, list of frames Γ = ∅
5:         for each signal σ_k in S(ψ_i) do
6:             Create a new frame F_{n+1} containing only σ_k
7:             Compute the total BW utilization u_{n+1} of frames F_1, ..., F_n, F_{n+1}
8:             for j = 1 to n do
9:                 if (σ_k can be added to F_j) then
10:                    Add σ_k to F_j
11:                    Compute the total BW utilization u_j of frames F_1, ..., F_n
12:                    Remove σ_k from F_j
13:                else
14:                    Set u_j to infinity
15:            Find the smallest u_j among u_1, ..., u_{n+1} and pack σ_k in F_j
16:            if (j == n + 1) then add F_{n+1} to Γ and set n = n + 1
17:    Return Γ
```

bandwidth utilization (as explained in the description of the heuristic) for signal σ can be done in $O(r + \sum_{i=1}^{r} |\gamma_i|) = O(s)$ time since $r \leq f \leq s$ and as observed earlier, $\sum_{i=1}^{r} |\gamma_i| \leq s$. As we need to compute the bandwidth utilization for at most $f + 1$ alternatives (including the new frame containing only σ), the time used for this step is $O(sf)$. In other words, for each signal, the greedy heuristic uses $O(sf)$ time. So, over all the s signals in $\mathcal{S}(\psi_i)$, the time complexity of the heuristic is $O(s^2 f)$.

6.2 Handling Infeasibility

In the second stage of the optimization, in case Algorithm 1 (i.e., the modified Audsley's Algorithm) fails to find a schedulable priority assignment, we propose a repacking method, with three variations, so that the frames may become schedulable. Our repacking strategy consists of two parts: the first part unpacks a selected set of frames based on certain conditions, and the second part repacks the signals removed from frames. The first part (unpacking of frames) is based on the following two methods.

1. **Unpacking Based on Destination Domains:** In case of a multi-domain system, Algorithm 1 reports infeasibility when a particular priority level cannot be assigned to any frame. This infeasibility could occur due to certain domains. Therefore, our first unpacking method attempts to separate the signals destined for different domains. The intuition behind such an unpacking can be understood from the following example. Consider a frame with 14 signals: σ_1, σ_2, …. σ_{14}. Suppose the first 12 signals have a total size of 40 bytes and their destination domain is D_1 while signals σ_{13} and σ_{14} have a total size of 2 bytes and their destination domain is D_2. Thus, such a frame carries an extra payload of 40 bytes to domain D_2. It could be beneficial to remove the signals intended for domain D_2 from the frame so that the overall schedulability (in particular for D_2) may be improved.
2. **Unpacking Based on Deadline:** Increasing the deadline of a frame is another method we adopt in order to satisfy the schedulability constraints. We apply this in our unpacking

scheme by separating the signals with the smallest deadline in a frame, thereby increasing the frame's deadline and its schedulability.

We apply at most one unpacking scheme per frame in an iteration. If the first criterion (destination domain based unpacking) is applicable to a frame, then we unpack the corresponding signals and do no further unpacking for this frame. If the first criterion does not apply to a frame (i.e., all the signals in the frame have the same destination domain), then we check the second criterion (deadline based unpacking). If neither of the criteria is met for a frame, we do not unpack that frame. After unpacking the signals, we repack them into existing frames using first-fit, best-fit and worst-fit heuristics. The repacking should satisfy the previously stated constraints of the problem (i.e., a frame should only have signals from the same ECU, all the signals in a frame should have harmonic periods and the total size of the signals in a frame should not exceed 64 bytes). The repacking step may suitably expand or shrink the size of a frame to handle the addition and removal of signals respectively.

We consider three different sets of frames as candidates for the unpacking and repacking steps. For each signal in the set, we unpack signals based on the above criteria and then repack them into existing frames (or generate new frames) using first-fit (FF), best-fit (BF) and worst-fit (WF) methods.

1. *All the frames*: In this case, we consider all the frames. We refer to this variation as the "All" heuristic.
2. *Unassigned frames from Audsley's method:* Here, we unpack only those frames for which Algorithm 1 could not assign a priority level. That is, at the priority level where Algorithm 1 fails to find a schedulable frame from the remaining set of frames, we consider this set for unpacking. We call this variation the "Unassigned" heuristic.
3. *Irreducible subset:* The idea behind computing an irreducible subset is similar to the computation of a minimal unsatisfiable core of a Boolean formula in conjunctive normal form [14]. When Algorithm 1 reports infeasibility, we compute a set of frames Γ', called an **irreducible subset**, satisfying the following condition: the set Γ' does not satisfy the schedulability constraints, but for any frame $\gamma \in \Gamma'$, the set $\Gamma' - \{\gamma\}$ becomes schedulable. We only unpack the frames which form an irreducible subset, based on the above two criteria. We refer this variation as "Irreducible subset" heuristic.

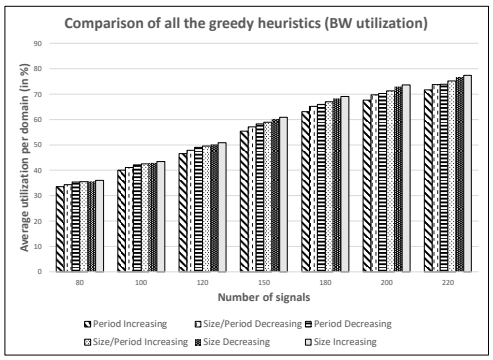
The reason for developing three variations of the repacking strategy is due to the complexity of making a given set of frames schedulable over the network. Since schedulability depends on a number of factors such as deadlines, traffic congestion over the network, sizes of frames, etc., there is no single factor which can be manipulated in order to obtain schedulability. The three different methods of repacking target different input sets. For example, for a particular input, it might be beneficial to unpack and repack a smaller set of "problematic" frames whereas another input might require a larger set of frames to be repacked. In the former case, the "Irreducible subset" approach could be more beneficial, and in latter case, the "All' approach might yield better results.

7 Experimental Results

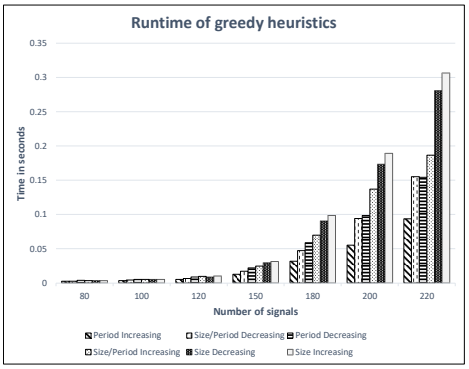
In this section, we present a detailed evaluation of the proposed algorithms using synthetic systems. For these experiments, we generated synthetic systems according to the guidelines on real-world automotive benchmarks [12], with minor modifications. Specifically, we redistributed the share of signals with size larger than 64 bytes to the bin "33-64 bytes" (as for this work we only consider signals with size up to 64 bytes), and the share of signals sent

Table 2 Signal parameters and their distribution.

Period (ms)	**Share**	**Size** (Bytes)	**Share**
1	4%	1	35%
2	3%	2	49%
5	3%	4	13%
10	31%	5–8	0.8%
20	31%	9–16	1.3%
50	3%	17–32	0.5%
100	20%	33–64	0.4%
200	1%		
1000	4%		



(a) Comparison of the bandwidth utilization of all the proposed greedy heuristics.



(b) Comparison of the runtime for all the proposed greedy heuristics.

Figure 4 Comparison of bandwidth utilization and runtime of greedy algorithm.

by engine control tasks to those with periods between 1 and 20ms (as we do not consider the signals with angle-synchronous periods). Table 2 summarizes the distribution of signal periods and payload sizes used for generating the synthetic systems. Each signal is randomly assigned a source and a destination domain (from the set of domain IDs) with a probability of $1/|\mathbb{D}|$, where $|\mathbb{D}|$ is the total number of domains. Therefore, the probability of a signal being cross-domain (i.e., the probability that its source and destination domains are different) is $1 - 1/|\mathbb{D}|$. In all our experiments, we use a system with three domains, 10 ECUs (in total) and vary the number of signals from 80 to 220. Hence for our experiments, the probability of a signal being cross-domain is 2/3.

Our experiments are conducted using a high performance computing cluster at Virginia Tech. This cluster has 4 x E7-8867v4 2.4 GHz (Broadwell) processors and 3TB, 2400MHz memory on a Unix platform. We used IBM's CPLEX as the ILP solver and implemented the algorithms in C++.

7.1 Comparison of Greedy Packing Heuristics

For all of our experiments in this section, we generated 5000 benchmarks for each system size (in terms of the number of signals).

We first evaluate the different greedy packing heuristic approaches by comparing the bandwidth utilization they provide after packing. We note that our greedy approach

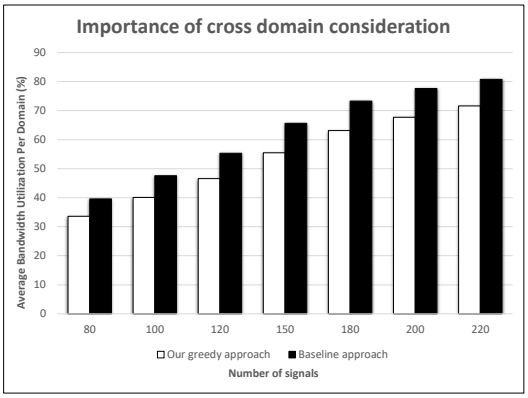


Figure 5 Comparing the greedy heuristic with a baseline packing approach which does not consider cross domain bandwidth while packing.

(Algorithm 2) leads to different bandwidth utilization values depending on the sorting criterion. We used the following parameters: period, size, and size/period (with increasing and decreasing orders). After the sorting step, each algorithm packs the signals in a greedy manner to optimize the bandwidth utilization.

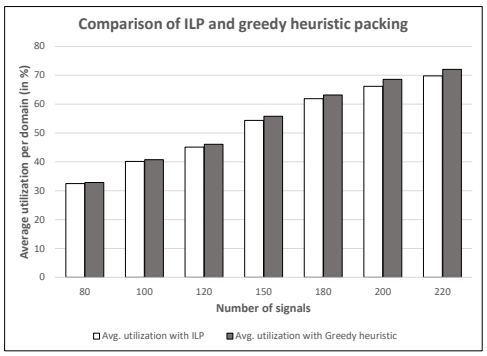
Figure 4a shows the bandwidth utilization per domain of the greedy heuristics, where the utilization is averaged over all those systems which are schedulable. As seen from the figure, there is small but noticeable variation in bandwidth utilization among the different heuristics. However, in all cases, sorting by increasing period performs the best. This is consistent with the observation that it is beneficial to pack signals with similar periods together, which in general reduces the total bandwidth utilization.

We also plot the runtime of the heuristics for each system size in Figure 4b. It is clear from this figure that the heuristic of sorting the signals in increasing order of periods has the smallest runtime; that is, it has an advantage over the other sorting approaches in terms of algorithmic efficiency as well. This is due to the fact that (as pointed out in the time complexity analysis for the greedy approaches) the runtime of the greedy algorithm is a function of the number of frames created, and the algorithm that sorts the signals in increasing order of periods creates the least number of frames for each signal size (as compared to the other heuristics). Hence, for the rest of the experiments, we use the heuristic that sorts the signals in increasing order of periods to represent all the greedy heuristics and compare it with the ILP-based approach.

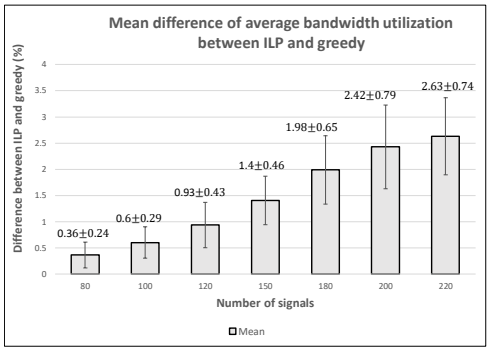
7.2 Importance of Cross-Domain Consideration

In this experiment, we demonstrate the importance of considering the cross-domain utilization during packing of the signals. Since there is no existing work for multi-domain frame packing in CAN-FD, we compare our greedy heuristic to a baseline approach which does not consider cross domain bandwidth utilization. Specifically, the baseline approach is the same as the greedy heuristic, except that it takes into account only the first part of Equation (20) (the source domain bandwidth utilization) but not the second part (cross-domain bandwidth utilization). For each system size, we tried 5000 benchmarks, and the bandwidth utilization represented is the average per domain over the 5000 benchmarks.

Figure 5 shows the significance of taking into account the cross-domain utilization for packing. We observe that there is a considerable reduction in bandwidth utilization per



(a) Greedy vs. ILP: bandwidth utilization per domain.



(b) Average and standard deviation for differences on utilization between ILP and greedy.

Figure 6 Comparison of ILP and greedy algorithms.

domain when frames are packed using our approach as opposed to the baseline approach: the typical reduction is in the range of 6% to 10% for utilization per domain. Also, the gap becomes larger as the number of signals increases.

7.3 Comparison of the Greedy Heuristic with ILP

In this experiment, we compare the ILP-based approach and the greedy heuristic in terms of their bandwidth utilization and the runtime. For these experiments, we used 100 synthetic systems for each size due to the excessively long runtime of ILP. We stopped at systems with 220 signals as ILP cannot scale to any larger systems: for systems with 4 domains, 15 ECUs and 250 signals, each of them takes about 7 hours on average.

Figure 6a illustrates the average bandwidth utilization per domain over the systems that are schedulable (either in the first packing attempt or after iteration/repacking). As can be observed from the figure, the bandwidth utilization of the greedy approach is quite close to that of ILP. The maximum mean difference on bandwidth utilization per domain is about 2.7% for systems with 220 signals.

Since Figure 6a gives only the average bandwidth utilization over all the systems, to better compare the ILP and the greedy approaches, we present the variability of the difference between the utilization values reported by them in Figure 6b. The gray bars represent the mean difference in percentage (over the 100 random systems, which are schedulable) between the bandwidth utilization (per domain) given by the ILP and the greedy approaches. The error bars represent the standard deviation of the difference. Figure 7 presents the distribution of the number of systems (that are schedulable) into different bins which represent the range of the difference between the average bandwidth utilization of the ILP and greedy approaches (as indicated in the legend). As the number of signals increases, the number of systems assigned to the larger bins also increases. Thus, Figures 6b and 7 show that with increasing number of signals, the difference in the bandwidth utilization given by the ILP and greedy approaches increases.

Figure 8a presents the average runtime of the systems for the ILP and greedy heuristic (in log scale). It is evident that the ILP would have scalability issues for larger systems, as the average runtime for the systems with 220 signals is already over 2.5 hours. The greedy approach on the other hand runs about 6 orders of magnitude faster than the ILP. Thus, we

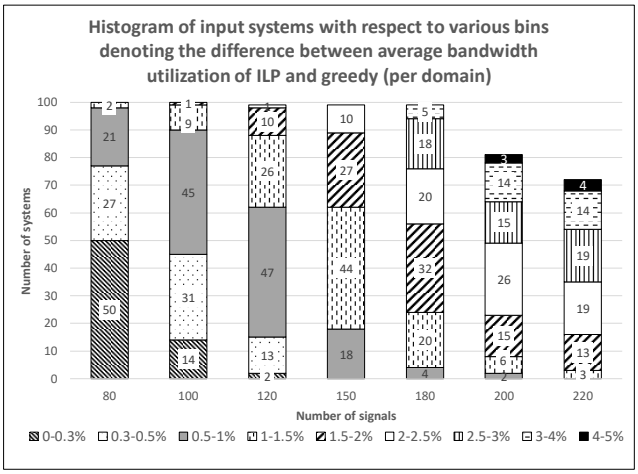
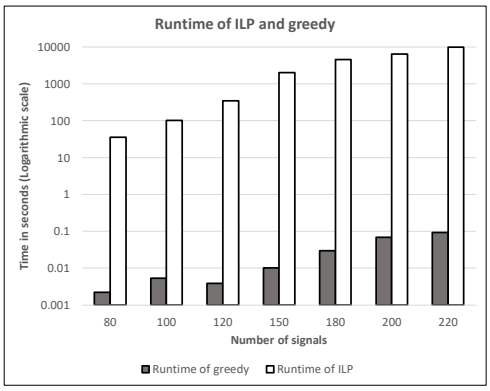
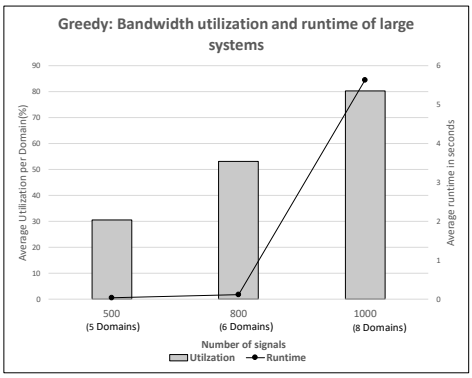


Figure 7 Distribution of systems within the various bin sizes (difference between greedy and ILP utilizations per domain).



(a) Greedy vs. ILP: runtime.



(b) Scalability of greedy: bandwidth utilization and runtime per domain for large sized systems.

Figure 8 Scalability of greedy with respect to ILP and industry sized systems.

can conclude that the greedy algorithm provides a packing whose bandwidth utilization is comparable to that of the ILP with a much smaller runtime. We note that in Figure 8a the average runtime of the greedy heuristic for 100 signals is slightly higher than that for the subsequent case of 120 signals. This is because one of the systems (out of 100) turned out to be unschedulable and thus the heuristic runs 10 iterations for this case, thereby increasing the average runtime. On the other hand, the unschedulable system in the case of 120 signals becomes schedulable in just one iteration with our heuristic (please refer to Figure 10).

In order to check the scalability of the greedy heuristic for industry sized systems we also conducted experiments (with 5000 systems) having 500, 800 and 1000 signals with 5, 6 and 8 domains and 5, 7 and 10 ECUs per domain respectively. We plot the bandwidth utilization and runtime in Fig.8b. The runtime of these systems was observed to be less than 6 seconds per system on the cluster (even in the largest size of 1000 signals), which shows that the

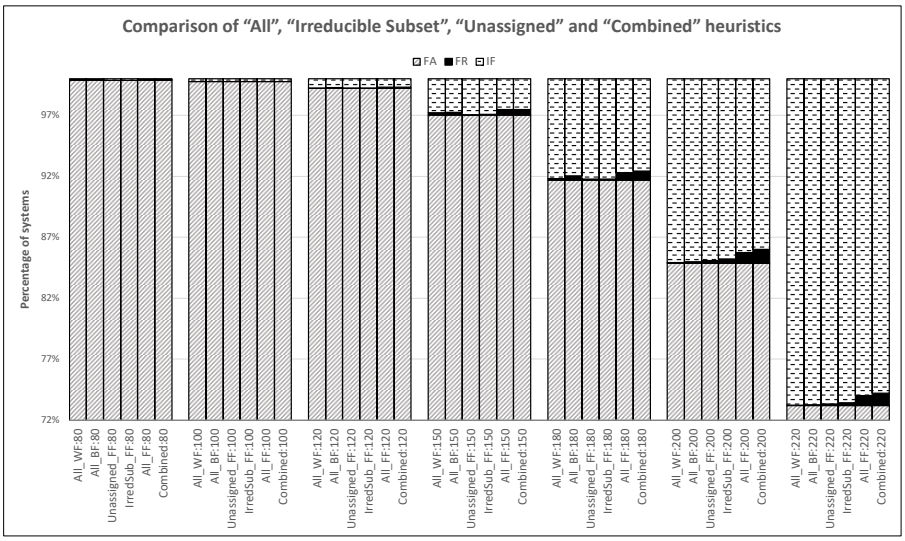


Figure 9 Comparison of "All","Irreducible subset","Unassigned" and "Combined" heuristics with respect to schedulability.

heuristic is easily scalable to large size systems. However for these experiments we used a slightly modified data generation scheme, where the contribution of signals from periods 1, 2 and 5 ms which was 10% was distributed to signals with period 10 and 20 ms equally.

7.4 Handling Infeasibility

In this set of experiments, we compare the approaches for handling infeasibility for the greedy heuristic. We generated 5000 random systems for each system size (number of signals in the system). As described in Section 6.2, we have implemented three variations of the repacking heuristic, namely "All", "Unassigned" and "Irreducible subset", and each of these variations uses three types of repacking algorithms: first-fit (FF), best-fit (BF) and worst-fit (WF). We also combined all the three variations (with first-fit), which resulted in (slightly) better results with respect to feasibility. We refer to this heuristic as the "Combined" heuristic.

In the experiments, we find that FF consistently outperforms the other two repacking algorithms BF and WF. Among the three variations, "All" heuristic gives the best results most of the time with respect to the number of feasible systems after the iterative procedure. To minimize clutter, in Figure 9 we only present the results for six approaches: "All" with FF, WF and BF, "Unassigned" with FF, "Irreducible Subset" with FF, and "Combined" with FF. The rectangular bars represent the percentage of the total number of systems, the gray section gives the number of systems feasible in first attempt (FA), the black section gives the number of systems feasible after repacking (FR), and the section with horizontal dash pattern gives the number of infeasible systems (IF).

We observe that the scheme "All" with first-fit (labeled as All-FF) is able to get the maximum number of systems to become feasible after the iterative step (for each system size). This is due to the fact that the "Irreducible subset" and "Unassigned" heuristics unpack a subset of the frames unpacked by the "All" heuristic. Therefore "All" is able to remove potentially "problematic" signals from all the frames and thus provide more schedulable cases. Also, by combining all our repacking heuristics, more than 92% of the systems become

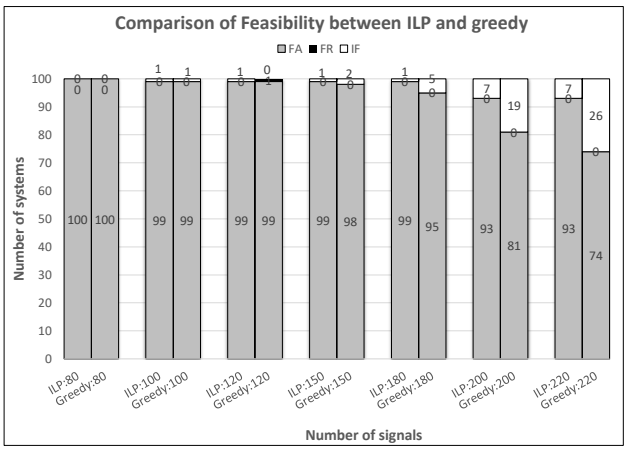


Figure 10 Infeasibility handling of ILP and greedy heuristic.

feasible (after repacking) for signal sizes 180 and below. For larger signal sizes, namely 200 and 220, 86% and 74% of the systems become feasible (after repacking) respectively.

Finally, we compare the infeasibility handling of the ILP and the greedy heuristic. For ILP, we use the iterative procedure described in Section 5.3, and for the greedy heuristic we use the "All" with first-fit scheme, since it was found to give the best results for infeasibility handling. Due to the long runtime of ILP, we generated 100 random systems for each system size. As in Figure 10, the performance of the greedy heuristic is comparable to that of ILP with respect to the number of feasible cases for small systems (namely, with number of signals below 150). However, for system size of 180 the greedy packing results in about 5% infeasible cases whereas the ILP delivers just 1% infeasible cases. Furthermore, for system size 200 and 220, the ILP gives 7% infeasible cases whereas the greedy approach gives 19% and 26% infeasible cases respectively. Due to its optimal packing (lower bandwidth utilization over the network), the ILP provides better feasibility at the cost of a much longer runtime.

8 Conclusions

In this paper, we motivate and propose solutions for the problem of frame packing for multi-domain CAN-FD systems. Existing work on frame packing for CAN-FD systems has not considered the problem from a multi-domain perspective. Our experiments show the significance of considering the multi-domain aspect (i.e., the inter-domain communication) for packing signals into frames. We present two approaches for the problem, namely ILP and greedy heuristic, both of which pack signals into frames with the goal of minimizing the bandwidth utilization over all the domains. In addition, we proposed an extension to Audsley's algorithm for assigning priority identifiers to the frames in the multi-domain case. In case of infeasibility, we developed several repacking heuristics so that the system may become feasible. Our experimental results show that the performance of the greedy heuristic is comparable to that of the ILP with respect to the bandwidth utilization as well as feasibility of the packed frames. However it is much faster than the ILP.

One line of future work is to investigate the frame packing problem for a heterogeneous multi-domain system where domains may be served by different communication protocols such as CAN, switched Ethernet, etc. In this work we have targeted optimization of bandwidth

utilization which is an important metric for conserving network bandwidth (for future feature additions) and obtaining better schedulability. In our future work we would be interested in considering other aspects such as optimization of extensibility [26] and robustness [4].

Acknowledgments. The authors would like to thank Sudhakaran M. from GM R&D as the need for multi-domain problem formulation was triggered after an initial discussion with him on the opportunities for bandwidth optimization in automotive bus topologies. The work in this paper is supported by GM R&D. The authors also acknowledge Advanced Research Computing at Virginia Tech for providing computational resources and technical support that have contributed to the results reported within this paper.

References

1 N. C. Audsley. On priority assignment in fixed priority scheduling. *Information Processing Letters*, 79(1):39–44, 2001. doi:10.1016/S0020-0190(00)00165-4.

2 Unmesh Dutta Bordoloi and Soheil Samii. The frame packing problem for CAN-FD. In *Proceedings of the IEEE 35th IEEE Real-Time Systems Symposium (RTSS)*, pages 284–293, December 2014. doi:10.1109/RTSS.2014.8.

3 Armaghan Darbandi, Sungoh Kwon, and Myung Kyun Kim. Scheduling of time triggered messages in static segment of FlexRay. *International Journal of Software Engineering and its Applications*, 8(6):195–208, 2014. doi:10.14257/ijseia.2014.8.6.16.

4 Robert I. Davis and Alan Burns. Robust priority assignment for messages on controller area network (can). *Real-Time Systems*, 41(2):152–180, 2009. doi:10.1007/s11241-008-9065-2.

5 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.

6 Robert I Davis, Liliana Cucu-Grosjean, Marko Bertogna, and Alan Burns. A review of priority assignment in real-time systems. *Journal of Systems Architecture*, 65:64–82, 2016. doi:10.1016/j.sysarc.2016.04.002.

7 Donald K. Friesen and Michael A. Langston. Variable sized bin packing. *SIAM Journal on Computing*, 15(1):222–230, 1986. doi:10.1137/0215016.

8 Michael R Garey and David S Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman and Company, San Francisco, CA, 1979.

9 Florian Hartwich. CAN with flexible data-rate. In *Proc. 13th International CAN Conference (iCC)*, pages 14:1–14:9. CAN in Automation (CiA), 2012.

10 Hogenmüller and Triess. Cost efficient gateway architecture for deterministic automotive networks, 2013. URL: https://standards.ieee.org/events/automotive/12_Hogenmueller_Triess_EDE_Handout.pdf.

11 Minkoo Kang, Kiejin Park, and Myong-Kee Jeong. Frame packing for minimizing the bandwidth consumption of the FlexRay static segment. *IEEE Transactions on Industrial Electronics*, 60(9):4001–4008, 2013. doi:10.1109/TIE.2012.2208433.

12 S. Kramer, D. Ziegenbein, and A. Hamann. Real world automotive benchmark for free. In *Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS)*, 2015.

13 Martin Lukasiewycz, Michael Glaß, Jürgen Teich, and Paul Milbredt. FlexRay schedule optimization of the static segment. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis (CODES)*, pages 363–372. IEEE-ACM, 2009. doi:10.1145/1629435.1629485.

14 Inês Lynce and Joao P Marques-Silva. On computing minimum unsatisfiable cores. In *Proceedings of the International Symposium on Theory and Applications of Satisfiability Testing*, pages 305–310, 2004.

15 Frank D Murgolo. An efficient approximation scheme for variable-sized bin packing. *SIAM Journal on Computing*, 16(1):149–161, 1987. doi:10.1137/0216012.

16 Marco Di Natale, Celso Luiz Mendes da Silva, and Max Mauro Dias Santos. On the applicability of an MILP solution for signal packing in CAN-FD. In *Proceedings of the IEEE 14th International Conference on Industrial Informatics (IEEE-INDIN)*, July, 2016. doi:10.1109/INDIN.2016.7819350.

17 Florian Polzlbauer, Iain Bate, and Eugen Brenner. Optimized frame packing for embedded systems. *IEEE Embedded Systems Letters*, 4(3):65–68, 2012. doi:10.1109/LES.2012.2208094.

18 Paul Pop, Petru Eles, and Zebo Peng. Schedulability-driven frame packing for multicluster distributed embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(1):112–140, 2005. doi:10.1145/1053271.1053276.

19 Rishi Saket and Nicolas Navet. Frame packing algorithms for automotive applications. *Journal of Embedded Computing*, 2(1):93–102, 2006. doi:10.1.1.5.1953.

20 Kristian Sandstrom, C Norstom, and Magnus Ahlmark. Frame packing in real-time communication. In *Proceedings of the Seventh International Conference on Real-Time Computing Systems and Applications*, pages 399–403. IEEE, 2000. doi:10.1109/RTCSA.2000.896418.

21 Bogdan Tanasa, Unmesh Dutta Bordoloi, Petru Eles, and Zebo Peng. Reliability-aware frame packing for the static segment of FlexRay. In *Proceedings of the Ninth ACM international conference on Embedded software*, pages 175–184. ACM, 2011. doi:10.1145/2038642.2038670.

22 KW Tindell, Hans Hansson, and Andy J Wellings. Analysing real-time communications: controller area network (CAN). In *Proceedings of Real-Time Systems Symposium*, pages 259–263. IEEE, 1994. doi:10.1109/REAL.1994.342710.

23 Gökhan Urul. *A Frame Packing Method to Improve the Schedulability of CAN and CAN-FD.* PhD thesis, Middle East Technical University, Turkey, 2015.

24 Haibo Zeng, Marco Di Natale, Arkadeb Ghosal, and Alberto Sangiovanni-Vincentelli. Schedule optimization of time-triggered systems communicating over the FlexRay static segment. *IEEE Transactions on Industrial Informatics*, 7(1):1–17, 2011. doi:10.1109/TII.2010.2089465.

25 Wei Zheng, Qi Zhu, Marco Di Natale, and Alberto Sangiovanni Vincentelli. Definition of task allocation and priority assignment in hard real-time distributed systems. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 161–170. IEEE, 2007. doi:10.1109/RTSS.2007.40.

26 Qi Zhu, Yang Yang, Eelco Scholte, Marco Di Natale, and Alberto Sangiovanni-Vincentelli. Optimizing extensibility in hard real-time distributed systems. In *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, pages 275–284. IEEE, 2009. doi:10.1109/RTAS.2009.37.

Semi-Partitioned Scheduling of Dynamic Real-Time Workload: A Practical Approach Based on Analysis-Driven Load Balancing

Daniel Casini[1], Alessandro Biondi[2], and Giorgio Buttazzo[3]

1 **Scuola Superiore Sant'Anna, Pisa, Italy**
daniel.casini@santannapisa.it
2 **Scuola Superiore Sant'Anna, Pisa, Italy**
alessandro.biondi@santannapisa.it
3 **Scuola Superiore Sant'Anna, Pisa, Italy**
giorgio.buttazzo@santannapisa.it

Abstract

Recent work showed that semi-partitioned scheduling can achieve near-optimal schedulability performance, is simpler to implement compared to global scheduling, and less heavier in terms of runtime overhead, thus resulting in an excellent choice for implementing real-world systems. However, semi-partitioned scheduling typically leverages an off-line design to allocate tasks across the available processors, which requires a-priori knowledge of the workload. Conversely, several simple global schedulers, as global earliest-deadline first (G-EDF), can transparently support dynamic workload without requiring a task-allocation phase. Nonetheless, such schedulers exhibit poor worst-case performance.

This work proposes a semi-partitioned approach to efficiently schedule dynamic real-time workload on a multiprocessor system. A linear-time approximation for the C=D splitting scheme under partitioned EDF scheduling is first presented to reduce the complexity of online scheduling decisions. Then, a load-balancing algorithm is proposed for admitting new real-time workload in the system with limited workload re-allocation. A large-scale experimental study shows that the linear-time approximation has a very limited utilization loss compared to the exact technique and the proposed approach achieves very high schedulability performance, with a consistent improvement on G-EDF and pure partitioned EDF scheduling.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

Keywords and phrases semi-partitioned scheduling, dynamic workload, real-time

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.13

1 Introduction

Many real real-time systems are characterized by a dynamic workload where computational activities (tasks) can join and leave the system, e.g., depending on the occurrence of specific events in their operating environment. Representative examples are modern multimedia software systems [17] (including those widely available in smartphones and tablets), cloud services [28], real-time databases, and open environments in which software components may join the system while the rest of the components continue to operate. Indeed, the possibility to spawn tasks at runtime is given by several real-time operating systems, including VxWorks, QNX and Linux. Such operating systems offer *global* scheduling policies such as *global fixed-priority* (G-FP) and *global earliest-deadline first* (G-EDF), which have the precious benefit of providing an automatic load balancing across the available processors, thus providing to the

29th Euromicro Conference on Real-Time Systems (ECRTS 2017).
Editor: Marko Bertogna; Article No. 13; pp. 13:1–13:23

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

application designer a simple and application-transparent scheduling mechanism. This benefit likely determined the popularity of such schedulers; however, they have been demonstrated being not optimal and generally exhibit poor worst-case performance due to several difficulties that have been identified in the literature [18]. For this reason, numerous efforts have been spent in studying and analyzing different techniques for scheduling real-time workload on multiprocessor systems. In particular, several optimal multiprocessor scheduling algorithms have been proposed, such as RUN [37], U-EDF [33], QPS [32] and LLREF [16], which are generally more complex (and hence more difficult to implement) and more expensive in terms of run-time overhead with respect to G-FP and G-EDF. Besides global schedulers, alternative approaches have been proposed based on *partitioned* and *semi-partitioned* scheduling. The former class of schedulers relies on a static allocation of the workload to the processors, which is generally suitable for hard real-time systems with fixed task sets. Semi-partitioned scheduling allows improving the performance of partitioned schedulers when a valid workload allocation cannot be found or simply does not exist. While some tasks are statically allocated to the processors, others are *split* across multiple processors, i.e., they are subject to a controlled (and limited) migration at specific time instants during their execution. As for partitioned schedulers, semi-partitioned schedulers typically leverage an off-line phase for allocating the tasks, which makes them less prone to support dynamic workload. Recently, Brandenburg and Gül [12] demonstrated that by clever task partitioning, semi-partitioned EDF scheduling with C=D splitting [13] allows achieving near-optimal performance, while being a much simpler and lighter (in terms of run-time overhead) approach with respect to global schedulers. As most of the papers targeting multiprocessor real-time scheduling, their work focused on static task sets only. However, the relevance of such a result suggests that also dynamic workload may benefit of semi-partitioned scheduling. Nonetheless, considerable challenges arise when aiming at supporting the C=D semi-partitioned scheduling of dynamic workload. In particular, the C=D splitting algorithm has a high computational complexity and therefore it cannot be adopted on-line without incurring in high overheads. Furthermore, load-balancing algorithms are needed to support the dynamic allocation and splitting of incoming workload.

Contribution. This paper makes the following three contributions. First, it proposes linear-time approximate methods for performing the C=D splitting, which enable making practically viable online scheduling decisions. Second, it presents load-balancing algorithms for C=D semi-partitioned scheduling to admit new workload, and performing limited re-allocations to facilitate the admission of future workload. Third, it reports on two large-scale experimental studies that have been conducted to assess the performance of the proposed methods.

Paper structure. The rest of the paper is organized as follows. Section 2 introduces the system model, reviews the essential background, and presents the adopted notation. Section 3 proposes three linear-time approximate methods for performing the C=D splitting. Section 4 presents some load-balancing algorithms for admitting new workload and performing limited workload re-allocations. Section 5 reports on the experimental results. Section 6 discusses the related work. Finally, Section 7 concludes the paper and illustrates some future work.

2 System Model and Background

This paper considers the problem of scheduling dynamic workload consisting of an arbitrary number of *reservation servers* upon m identical processors. A reservation r_i is characterized by

a *budget* of execution time C_i and a minimum inter-replenishment time T_i. Such reservation servers can arbitrarily enter or leave the system. However, each of them must pass an *acceptance test* (based on parameters C_i and T_i) before being admitted into the system; those that do not pass the test are rejected (i.e., ignored). At any point in time, $\mathcal{R}$ denotes the set of reservations that are currently admitted for execution. Each reservation server $r_i \in \mathcal{R}$ generates a potentially infinite number of *instances*: in each instance, the server executes for at most C_i time units and then is de-descheduled. An instance of the server begins when (i) the budget of the server is replenished and (ii) the server has pending workload to execute. An instance terminates either (i) when the budget is exhausted or (ii) the server does not have anymore pending workload to execute. Note that the beginning times of the instances follow a *sporadic* pattern. Reservations are considered to be independent (i.e., they do not share resources other than the processors). The results presented in this work are not limited to a specific reservation algorithm, but the server behavior has to comply with some requirements that are discussed in the next section. We say that a reservation r_i is *schedulable* if, in every instance of the server, the system is able to guarantee the execution of its entire budget C_i (used to serve pending workload running upon the server) before the time at which the budget will be replenished. The goal of the acceptance test is to ensure that *all* the reservations into the set $\mathcal{R}$ are always schedulable. In this work, the acceptance test employs an on-line *load balancing* algorithm that allocates the reservations to the processors, which will be presented in Section 4. Each reservation can be used for manifold purposes, including (i) serving the execution of a single periodic/sporadic real-time task; (ii) implementing a *hierarchical scheduling framework* [39], i.e., managing a local scheduler upon the reservation that in turn manages a set of real-time tasks; and (iii) serving the execution of non-real-time (i.e., best-effort) workload. Note that the adoption of reservation servers also provides the benefit of guaranteeing a *temporal isolation* of the workload, protecting the system from tasks' overruns or processor-eager best-effort computational activities. This feature is particularly suited for systems running dynamic workload, for which – conversely to static, safety-critical real-time systems – accurate estimates of the tasks' *worst-case execution time* (WCET) are often not available. Such a computational model is also of practical relevance, as it is analogous to the one supported by the `SCHED_DEADLINE` scheduling class of Linux, today available in the main distribution of the kernel and hence present in billions of machines and devices around the world. Nonetheless, the approach proposed in this paper is also valid for sporadic tasks with implicit deadlines. In this work, the reservations that are admitted for execution are scheduled under *semi-partitioned* EDF scheduling with the C=D splitting scheme [12, 13], which is briefly reviewed in the next section.

2.1 C=D Semi-partitioned Scheduling of Reservations

Semi-partitioned scheduling allows improving the schedulability performance of pure partitioned scheduling when valid static allocations of the reservations cannot be found or simply do not exist. With this approach, some reservations are statically allocated to processors, while others are *split* across multiple processors, thus involving the migration of the workload executing upon such reservations. More specifically, the budget of some reservations is divided into multiple portions (i.e., chunks) that are executed on different processors with precedence constraints. Among the various methods that one may imagine to split the budget, the so called C=D splitting scheme has been found to perform particularly well. One of the first proposals of this method is due to Kato and Yamasaki [26]: the authors assumed partitioned fixed-priority scheduling as a baseline scheduling algorithm while ensuring that the split portions of budget are executed with the highest priority on each processor. Since a

reservation scheduled at the highest priority does not suffer temporal interference from the other reservations allocated on the same processor, it is guaranteed that its budget C will be always consumed within a deadline of $D = C$ time units from its release. The authors exploited this property to facilitate the splitting phase. Later, Burns et al. [13] proposed an improved C=D scheme under partitioned EDF scheduling, which exploits scheduling deadlines to guarantee the system schedulability in the presence of splitting. Following their approach, a budget is split into n portions, each allocated on n different processors. The first $n-1$ portions are scheduled with a scheduling deadline equal to the corresponding duration of the portion – i.e., they have always zero laxity. Differently from [26], the last portion is scheduled with a deadline greater than or equal to the duration of the portion, hence it may suffer temporal interference from other reservations. Recently, Brandenburg and Gül [12] proposed an extension of the Burns et al.'s approach where the execution order of the portions of budget is flipped. This approach allows taking advantage of slack reclamation, which in turn provides the benefit of reducing the number of migrations in the average-case. In this paper, the latter splitting scheme is considered for the run-time scheduling mechanism.

Run-time scheduling mechanism. As soon as a server is admitted, its budget is immediately replenished. If an instance of a server r_i begins at a time t, the next budget replenishment is set at time $t + T_i$. As typical for EDF scheduling, each reservation server r_i is assigned a *relative deadline* D_i. Each instance of r_i beginning at time t is scheduled with absolute deadline $t + D_i$. The servers execute without self-suspensions: i.e., the budget is discharged if the server has pending workload that is not ready to execute and is depleted when the server stops having pending workload. Following semi-partitioned scheduling, some reservations servers are statically allocated to processors (i.e., they never migrate across processors) – for this reason they are referred to as *partitioned reservations*. Partitioned reservations have a relative deadline equal to their minimum inter-replenishment time, that is $D_i = T_i$. Other reservations are split across multiple processors and are referred to as *semi-partitioned reservations*. Consider a semi-partitioned reservation r_i whose budget C_i is split into two portions, say C'_i and C''_i such that $C_i = C'_i + C''_i$. Following the approach proposed in [12], the first portion of budget is scheduled on a processor P' with relative deadline $D'_i = T_i - C''_i$ and minimum inter-replenishment time T_i, while the second one is scheduled on a different processor $P'' \neq P'$ with relative deadline $D''_i = C''_i$ and minimum inter-replenishment time T_i. This split gives rise to two sub-reservations, denoted as *head reservation* and *tail reservation*, respectively. At run-time, the execution of the workload executing upon a semi-partitioned reservation r_i is subject to the following rules. Suppose that an instance of r_i begins at time t and that the server has continuously pending workload to execute. The first C'_i units of budget of r_i are served by its head reservation, i.e., on processor P'. Then, every time the budget C'_i is exhausted, the workload executing upon r_i is *migrated* to processor P'', where it will be served by the r_i's tail reservation. If the head reservation is schedulable within its relative deadline D'_i, this event is guaranteed to happen at a time $t' \leq t + D'_i$. Contextually, the head reservation is de-scheduled and its budget will be replenished at time $t + T_i$. If the tail reservation is schedulable within its relative deadline $D''_i = C''_i$, the C=D approach [12] guarantees that C''_i units of time are served before time $t + T_i$, thus guaranteeing the schedulability of r_i. Once the budget of the tail reservation is exhausted, also this server is de-scheduled and its budget will be replenished at time $t'' + T_i$, where t'' is the arrival time of its last instance. The pending workload upon r_i will then be able to restart the execution from processor P' (thus involving another migration) at time $t + T_i$. Note that, although the two sub-reservations have the same minimum inter-replenishment

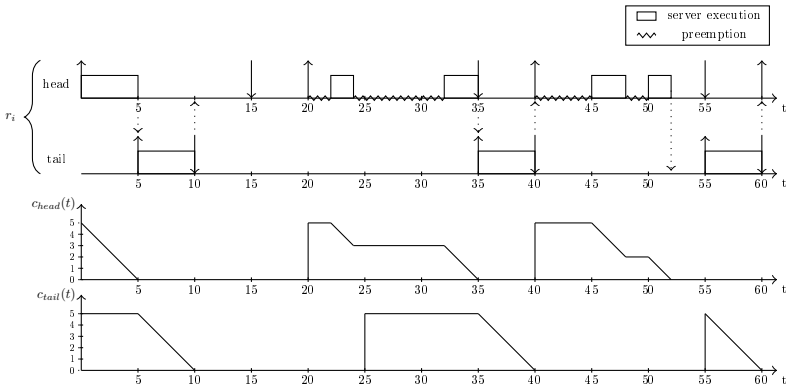


Figure 1 Example of semi-partitioned scheduling of a reservation r_i ($C_i = 10, T_i = 20$) under C=D splitting. The budget of r_i is split into two portions of length 5 time units, executing on two processors. Up-arrows denote the beginning of an instance of the servers. Down-arrows denote the absolute deadlines of each instance. Dotted arrows denote the migration of the workload executing upon r_i across the two processors.

time, their replenishment times are generally not synchronized. The approach can be further generalized by considering budget splits in more than two portions: in this case, a reservation is split into one head reservation and *multiple* tail reservations. For each processor P, at each point in time the system selects for execution the reservation allocated to P that has (i) a pending instance and (ii) the earliest absolute deadline. To better clarify the scheduling mechanism, consider a reservation r_i with $C_i = 10$ and $T_i = 20$ that is split into: (i) one head reservation configured with $C'_i = 5$ and $D'_i = 15$; (ii) one tail reservation configured with $C''_i = 5$, $D''_i = 5$. A possible schedule of such sub-reservations is illustrated in Figure 1, together with the evolution of their budgets over time (indicated by functions $c_{head}(t)$ and $c_{tail}(t)$, respectively).

It is worth observing that the C=D approach implicitly poses the limitation that *no more than one* tail reservation can be allocated on each processor. For the sake of simplicity, in this work we also pose this limitation for head reservations: this is reflected only in a restriction of the possible allocation configurations. One of the main issues with semi-partitioned scheduling consists in splitting and allocating the reservations. Previous work assumed a static workload and leveraged an off-line design phase to solve this problem. This phase typically consists in the combination of (i) bin-packing heuristics (such as variants of first-fit and worst-fit) to allocate the reservations and (ii) a splitting algorithm to decide how to size the budget portions of the semi-partitioned reservations. The next section briefly reviews the C=D splitting algorithm proposed by Burns et al. [13], which has also been adopted by Brandenburg and Gül in [12].

2.2 Burns et al.'s C=D Splitting Algorithm

Whenever a reservation r_i cannot be statically allocated to a single processor, Burns et al. [13] proposed to accomplish the splitting with the following two-phase approach:

(i) Given a processor P_k, an algorithm is used to compute the *maximum* $C''_i < C_i$ for which a tail reservation with budget C''_i, deadline $D''_i = C''_i$ and minimum inter-replenishment time T_i can be allocated to P_k *such that all the reservations running on* P_k *are schedulable.*

(ii) The remaining portion of budget $C'_i = C_i - C''_i$ is then allocated to another processor $\neq P_k$ following a bin-packing heuristic (or is in turn selected for being split).

The core of their proposal consists in the algorithm adopted in phase (i). Such an algorithm starts from the value of C_i'' for which the selected processor P_k is fully utilized (i.e., such that $\sum_{r_i \in \mathcal{R}_k} C_i/T_i = 1$) after allocating the tail reservation; then, it allocates the tail reservation to P_k and applies the following steps:

1. Perform the Quick convergence Processor-demand Analysis (QPA) [42] to determine whether the set of reservations allocated to P_k is schedulable.
2. If not, recompute a reduced value of C_i'' by means of a *fixed-point iteration* based on the failure point of the QPA (please refer to [13] for further details). Then, re-iterate the procedure from step 1 until the QPA does not fail.
3. If, at any iteration, the computed value of C_i'' reduces to 0, then the tail reservation cannot be allocated to processor P_k.

This algorithm is optimal, in the sense that it founds the maximum value of C_i'' for which a tail reservation can be safely allocated to processor P_k. However, it suffers from a high computational complexity. The QPA has a pseudo-polynomial time complexity when the utilization of the analyzed processor is strictly lower than one, while has exponential complexity in the case of a fully-utilized processor. Note that the latter case corresponds to the starting condition of the algorithm and that the QPA is applied multiple times. In addition, it requires the execution of fixed-point iterations that further increase the algorithm complexity. To the best of our knowledge, the actual complexity of this algorithm is unknown: anyway, it is clearly *unsuitable for performing on-line decisions* concerning the splitting of the reservations, especially if multiple alternatives for the splitting must be evaluated by a load balancing algorithm – which is the primary objective of this work.

2.3 Notation and Table of Symbols

The m processors are referred to as $P_1, P_2, \ldots, P_m$. The set of n_k reservations allocated to processor P_k (both statically or resulting from a split) is denoted by $\mathcal{R}_k$, with $\bigcap_{k=1}^{m} \mathcal{R}_k = \emptyset$. The utilization of a reservation r_i is denoted as $U_i = C_i/T_i$. Two functions $tail(P_k) = \{true, false\}$ and $head(P_k) = \{true, false\}$ are used to indicate whether a tail and a head reservation is allocated to P_k, respectively. If $tail(P_k) = true$, then $r_{tail,k} \in \mathcal{R}_k$ denotes the tail reservation allocated to P_k. Similarly, if $head(P_k) = true$, then $r_{head,k} \in \mathcal{R}_k$ denotes the head reservation allocated to P_k. The set of n_k^P partitioned reservations allocated to P_k is denoted as $\mathcal{R}_k^P \subseteq \mathcal{R}_k$. Given a tail reservation $r_{tail,k}$ (resp., head reservation $r_{head,k}$), the *father* reservation from which the split has been originated is denoted as $\mathcal{F}(r_{k,tail})$ (resp., $\mathcal{F}(r_{k,head})$). The notation adopted in this paper is summarized in Table 1.

3 An Approximated Algorithm For C=D Splitting

This section presents a new approach for computing the C=D splitting discussed in Section 2.2. The proposed algorithm provides an approximate solution to compute a safe lower-bound on the maximum zero-laxity portion of budget that can be allocated to a processor. The algorithm has been designed to have a *linear time* complexity in order to be efficiently applied for on-line load balancing. The baseline approach is first presented in Section 3.1. Then, two possible extensions are proposed in Section 3.2 to improve the algorithm precision. Finally, Section 3.3 discusses some implementation issues and the algorithm complexity.

3.1 The Baseline Approach

The method proposed in this paper is based on the *processor-demand criterion* (PDC) proposed by Baruah et al. [6]. The PDC analysis is based on the notion of *demand bound*

Table 1 Main notation adopted throughout the paper.

Symbol	Description
$\mathcal{R}$	set of reservations admitted into the system
P_k	k-th processor
$\mathcal{R}_k$	set of reservations allocated to processor P_k
$\mathcal{R}_k^P$	set of partitioned reservations allocated to processor P_k
n_k	number of reservations allocated to processor P_k
n_k^P	number of partitioned reservations allocated to processor P_k
r_i	i^{th} reservation
C_i	budget of r_i
T_i	minimum inter-replenishment time of r_i
D_i	relative deadline of r_i
U_i	utilization of r_i
$r_{head,k}$	head reservation allocated to P_k
$r_{tail,k}$	tail reservation allocated to P_k
$\mathcal{F}(r_i)$	father reservation of a tail or head reservation r_i

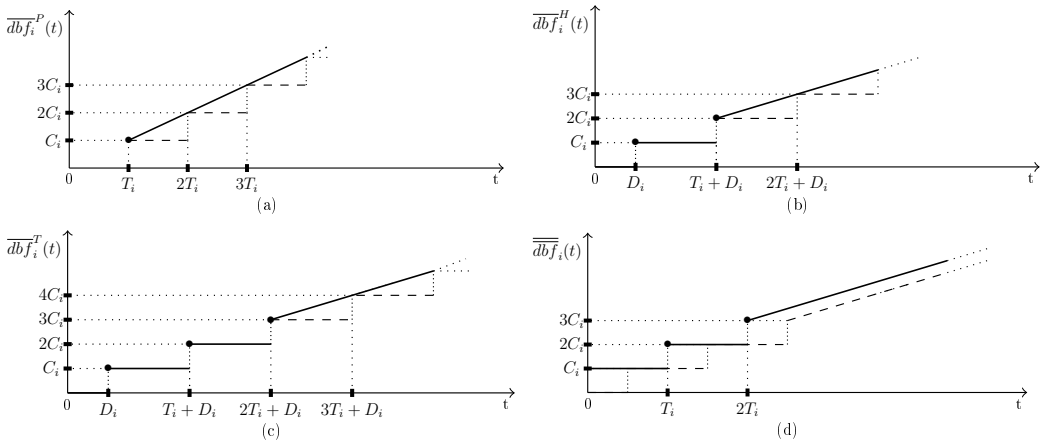


Figure 2 Illustrations of the demand bound functions introduced in Section 3.1 (solid lines). The dashed lines in insets (a), (b) and (c) depict functions $dbf_i(t)$, while the dashed line in inset (d) depicts function $\overline{dbf_i}^T(t)$.

function and provides an exact schedulability test for a set of constrained-deadline sporadic tasks executing upon a single processor under EDF scheduling. Since the reservation servers considered in this work behave as sporadic tasks [12], the schedulability of the reservations allocated to a given processor P_k can be verified by checking the PDC as $\forall t \geq 0,\ \sum_{r_i \in \mathcal{R}_k} dbf_i(t) \leq t$, where $dbf_i(t)$ is the demand bound function of r_i, defined as

$$dbf_i(t) = \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor C_i.$$

To design the approximate splitting algorithm, the demand bound function of each reservation is first approximated by an upper bound, which is a particular case of the one proposed by Fisher et al. [20]. In particular, three types of upper bounds are distinguished depending on whether a reservation is partitioned, head or tail.

Partitioned reservation. The demand bound function of a partitioned reservation r_i is upper-bounded with function $\overline{dbf_i}^P(t)$, defined as

$$\overline{dbf_i}^P(t) = \begin{cases} 0 & \text{if } t < T_i, \\ U_i t & \text{if } t \geq T_i. \end{cases}$$

Head reservation. The demand bound function of a head reservation r_i is upper-bounded with function $\overline{dbf_i}^H(t)$, defined as

$$\overline{dbf_i}^H(t) = \begin{cases} 0 & \text{if } t < D_i, \\ C_i & \text{if } D_i \leq t < T_i + D_i, \\ 2C_i + U_i(t - T_i - D_i) & \text{if } t \geq T_i + D_i. \end{cases}$$

Tail reservation. Finally, the demand bound function of a tail reservation r_i is upper-bounded with function $\overline{dbf_i}^T(t)$, defined as

$$\overline{dbf_i}^T(t) = \begin{cases} 0 & \text{if } t < D_i, \\ kC_i & \text{if } (k-1)T_i + D_i \leq t < kT_i + D_i,\ k = 1, 2 \\ 3C_i + U_i(t - 2T_i - D_i) & \text{if } t \geq 2T_i + D_i. \end{cases}$$

Graphical representations of these three functions are reported in Figures 2(a), 2(b) and 2(c). To keep a compact notation, the following function is also defined:

$$\overline{dbf_i}(t) = \begin{cases} \overline{dbf_i}^P(t) & \text{if } r_i \text{ is partitioned}, \\ \overline{dbf_i}^H(t) & \text{if } r_i \text{ is head}, \\ \overline{dbf_i}^T(t) & \text{if } r_i \text{ is tail}. \end{cases}$$

Based on these bounds, it is possible to formulate a sufficient PDC-based condition to verify the schedulability of the reservations allocated to a processor, which is expressed by the following theorem.

▶ **Theorem 1.** *A set of reservations $\mathcal{R}_k$ is EDF-schedulable on a single processor if $\sum_{r_i \in \mathcal{R}_k} U_i \leq 1$ and*

$$\forall t \in \bigcup_{r_i \in \mathcal{R}_k} \xi(r_i), \quad \sum_{r_i \in \mathcal{R}_k} \overline{dbf_i}(t) \leq t$$

where

$$\xi(r_i) = \begin{cases} \{T_i\} & \textit{if } r_i \textit{ is partitioned}, \\ \{D_i, T_i + D_i\} & \textit{if } r_i \textit{ is head}, \\ \{D_i, T_i + D_i, 2T_i + D_i\} & \textit{if } r_i \textit{ is tail}. \end{cases}$$

Proof. By construction, $\forall t \geq 0, \overline{dbf_i}(t) \geq dbf_i(t)$. Hence, if $\forall t \geq 0, \sum_{r_i \in \mathcal{R}_k} \overline{dbf_i}(t)) \leq t$ holds, then also the original PDC condition is satisfied. Note that $\overline{dbf_i}(t)$ is a stepwise not-decreasing function composed by either constant segments or linear segments, so also $W(t) = \sum_{r_i \in \mathcal{R}_k} \overline{dbf_i}(t)$ has the same shape with discontinuities in correspondence to the discontinuities of functions $\overline{dbf_i}(t)$. Let $\Delta_1, \Delta_2, \ldots, \Delta_j, \ldots$ be the ordered sequence of the

points t in which function $W(t)$ has a discontinuity. Consider one of such points $t = \Delta_j$. If $W(t)$ is constant in $[\Delta_j, \Delta_{j+1})$ (constant segment), then it is sufficient to check that $W(\Delta_j) \leq \Delta_j$ to guarantee that $\forall t \in [\Delta_j, \Delta_{j+1}),\ W(t) \leq t$. Now, consider the other case in which $W(t)$ has a linear segment in $[\Delta_j, \Delta_{j+1})$. By the hypothesis $\sum_{r_i \in \mathcal{R}_k} U_i \leq 1$, it follows that every linear segment of $W(t)$ has a slope that cannot be larger than 1. Hence, it is again sufficient to check that $W(\Delta_j) \leq \Delta_j$. Overall, the condition $W(t) \leq t$ must be checked only in the discontinuities of functions $\overline{dbf_i}(t)$, which occur for the points expressed by the set $\xi(r_i)$. Hence, the theorem follows. ◀

With the above theorem in place, the considered optimization problem can be defined. Consider a set of reservations $\mathcal{R}_k$ allocated to a processor P_k that *does not* already include a tail reservation. By Theorem 1, a safe budget C_{tail} for a tail reservation r_{tail} with minimum inter-replenishment time T_{tail}, such that r_{tail} can be safely allocated to P_k, can be computed by solving the following optimization problem:

$$\begin{aligned} &\text{maximize} \quad C_{\text{tail}} \\ &\text{subject to} \quad \sum_{r_i \in \mathcal{R}_k} \frac{C_i}{T_i} + \frac{C_{\text{tail}}}{T_{\text{tail}}} \leq 1 \\ &\qquad \sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(t) + \overline{dbf_i}^T(t) \leq t,\ \forall t \in \bigcup_{r_i \in \{\mathcal{R}_k \cup r_{tail}\}} \xi(r_i) \end{aligned}$$

This optimization problem can be manipulated to obtain a sub-optimal solution in a **closed-form**. To this end, the problem is rewritten by means of $J+1$ constraints $C_{\text{tail}} \leq V_j(\mathcal{R}_k, T_{tail})$ (with $j = 0, \ldots, J$) in which the functions $V_j(\mathcal{R}_k, T_{tail})$ are *independent* of C_{tail}, so that the solution can be easily computed as $C_{\text{tail}} = min_{j=0,\ldots,J}\ V_j(\mathcal{R}, T_{\text{tail}})$. In other words, given the parameters of the reservations in set $\mathcal{R}_k$ and the minimum inter-replenishment time T_{tail} of the tail reservation, the expressions $V_j(\mathcal{R}_k, T_{tail})$ must result in *constant terms*. First of all, note that the constraint $\sum_{r_i \in \mathcal{R}_k} \frac{C_i}{T_i} + \frac{C_{\text{tail}}}{T_{\text{tail}}} \leq 1$ (corresponding to a very simple necessary condition for feasibility) originates a trivial upper bound on the value of C_{tail}, that is

$$C_{\text{tail}} \leq C_{\text{tail}}^{\texttt{MAX}} = \left(1 - \sum_{r_i \in \mathcal{R}_k} U_i\right) T_{\text{tail}}.$$

Leveraging the bound $C_{\text{tail}}^{\texttt{MAX}}$, the terms $V_j(\mathcal{R}, T_{tail})$ can be derived by considering the constraints originated by the check-points in the set $\bigcup_{r_i \in \{\mathcal{R}_k \cup r_{tail}\}} \xi(r_i)$. First, note that functions $\overline{dbf}_i(t)$ are piece-wise defined in intervals that depend on the check-point t. When considering the check-points of the tail reservations (i.e., those in the set $\xi(r_{tail})$), the value of functions $\overline{dbf}_i(t)$ for the partitioned and the head reservations cannot be expressed in a closed-form as their value depend on the optimization variable C_{tail} (that is unknown), thus introducing a sort of circular dependency in the equations. The following lemma allows overcoming this issue.

▶ **Lemma 2.** *If the three conditions*

$$\begin{cases} C_{tail} \leq min_{r_i \in \mathcal{R}_k}(D_i) - \epsilon & (a) \\ \sum_{r_i \in \mathcal{R}_k} \overline{dbf_i}(T_{tail} + C_{tail}^{\texttt{MAX}}) + 2C_{tail} \leq T_{tail} + D_{tail} & (b) \\ \sum_{r_i \in \mathcal{R}_k} \overline{dbf_i}(T_{tail} + C_{tail}^{\texttt{MAX}}) + 3C_{tail} \leq 2T_{tail} + D_{tail} & (c) \end{cases}$$

hold (with $\epsilon > 0$ arbitrary small), then

$$\forall t \in \xi(r_{tail}),\ \sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(t) + \overline{dbf_i}^T(t) \leq t.$$

Proof. Each of the three conditions corresponds to an element of the set $\xi(r_{tail})$. Condition (a) is needed for verifying the constraint $\sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(D_{\text{tail}}) + \overline{dbf}_i^T(D_{\text{tail}}) \leq D_{\text{tail}}$. If the tail reservation (configured with $C_{\text{tail}} = D_{\text{tail}}$) does not have the smallest deadline among the reservations allocated to P_k, then it may be preempted, thus inevitably missing its deadline. Therefore, a solution exists only if $C_{\text{tail}} = D_{\text{tail}} < min_{r_i \in \mathcal{R}_k}(D_i)$, which then gives $\sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(D_{\text{tail}}) = 0$ and the constraint for point D_{tail} is implicitly verified. Conditions (b) and (c) verify the constraint $\sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(t) + \overline{dbf}_i^T(t) \leq t$ for points $t = T_{tail} + D_{tail}$ and $t = 2T_{tail} + D_{tail}$. Since functions $\overline{dbf}_i(t)$ are monotonic non-decreasing and $C_{\text{tail}} \leq C_{\text{tail}}^{\texttt{MAX}}$, then $\overline{dbf}_i(T_{\text{tail}} + D_{\text{tail}}) \leq \overline{dbf}_i(T_{tail} + C_{\text{tail}}^{\texttt{MAX}})$. Similarly, also $\overline{dbf}_i(2T_{\text{tail}} + D_{\text{tail}}) \leq \overline{dbf}_i(2T_{tail} + C_{\text{tail}}^{\texttt{MAX}})$. The lemma follows by noting that, in the two points, the value of $\overline{dbf}_i^T(t)$ corresponds to $2C_{\text{tail}}$ and $3C_{\text{tail}}$, respectively. ◀

Before proceeding with the constraints originated by the check-points of the head and partitioned reservations, it is necessary to introduce a new demand bound function $\overline{\overline{dbf}}_{tail}(t)$, which is explicitly conceived to deal with the contribution originated by the tail reservation. This function is illustrated in Figure 2(d) and allows removing the circular dependency that would have been introduced by the use of $\overline{dbf}_i^T(t)$.

▶ **Lemma 3.**

$$\forall t \geq 0,\ \overline{\overline{dbf}}_{tail}(t) \geq \overline{dbf_i}^T(t),$$

where

$$\overline{\overline{dbf}}_{tail}(t) = \begin{cases} C_{tail} & \text{if } t < T_{tail} \\ 2C_{tail} & \text{if } T_{tail} \leq t < 2T_{tail} \\ 3C_{tail} + U_{tail}(t - 2T_{tail}) & \text{if } t \geq 2T_{tail} \end{cases}$$

Proof. Let us consider separately the three cases in which $\overline{\overline{dbf}}_{tail}(t)$ is defined. If $t < T_{\text{tail}}$, then $\overline{dbf}_i^T(t)$ can be either equal to 0 or C_{tail}; hence $\overline{dbf}_i^T(t) \leq C_{\text{tail}}$. Since $0 < D_{\text{tail}} < T_{\text{tail}}$, if $T_{\text{tail}} \leq t < 2T_{\text{tail}}$, then $\overline{dbf}_i^T(t)$ can be either equal to C_{tail} or $2C_{\text{tail}}$; hence $\overline{dbf}_i^T(t) \leq 2C_{\text{tail}}$. For the same reason, if $t \geq 2T_{\text{tail}}$, then $\overline{dbf}_i^T(t)$ can be either equal to $2C_{\text{tail}}$ or $3C_{\text{tail}} + U_{tail}(t - 2T_{\text{tail}} - D_{\text{tail}})$. Since for $t \geq 2T_{\text{tail}}$ we have $U_{tail}(t - 2T_{\text{tail}}) \geq 0$, then $\overline{dbf}_i^T(t) \leq 3C_{\text{tail}} + U_{tail}(t - 2T_{\text{tail}})$. Hence the lemma follows. ◀

Thanks to this upper bound, it is now possible to remove the circular dependency in the constraints originated by the check-points of the head reservation (i.e., those in the set $\xi(r_{head,k})$).

▶ **Lemma 4.** *If the two conditions*

$$\begin{cases} \sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(D_{head}) + \overline{\overline{dbf}}_{tail}(D_{head}) \leq D_{head} \\ \sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(T_{head} + D_{head}) + \overline{\overline{dbf}}_{tail}(T_{head} + D_{head}) \leq T_{head} + D_{head} \end{cases}$$

hold, then

$$\forall t \in \xi(r_{head,k}),\ \sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(t) + \overline{dbf_i}^T(t)(t) \leq t.$$

Proof. The lemma directly follows from Lemma 3 and the definition of the set $\xi(r_i)$. ◀

Similarly, the same bound can be applied to the constraints originated by the check-points of the partitioned reservations.

▶ **Lemma 5.** *If*

$$\forall t \in \left\{T_i \mid r_i \in \mathcal{R}_k^P\right\}, \sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(T_i) + \overline{\overline{dbf}}_{tail}(T_i) \leq T_i$$

holds, then

$$\forall t \in \bigcup_{r_i \in \mathcal{R}_k^P} \xi(r_i), \sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(t) + \overline{dbf_i}^T(t) \leq t.$$

Proof. The lemma directly follows from Lemma 3 and the definition of the set $\xi(r_i)$. ◀

Finally, the results of Lemma 2, Lemma 4 and Lemma 5 are combined in the following theorem, which provides a *closed-form expression* for computing a safe bound on C_{tail}.

▶ **Theorem 6.** *A set of reservations $\mathcal{R}_k$ composed of n_k^P partitioned reservations, at most one head reservation and a tail reservation with minimum inter-replenishment time T_{tail}, can be safely EDF-scheduled on a single processor P_k if*

$$C_{tail} = D_{tail} = min_{j=0,\ldots,J}\{V_j(\mathcal{R}_k, T_{tail})\}$$

where $V_0(\mathcal{R}_k, T_{tail}), \ldots, V_J(\mathcal{R}_k, T_{tail})$ are defined as in Table 2.

Proof. The set of reservations $\mathcal{R}_k$ is schedulable if the conditions of Theorem 1 hold. Lemma 2, 4 and 5 provide sufficient conditions for which Theorem 1 holds. The terms in Table 2 are obtained by simple algebraic transformations of the conditions of such lemmas, which have been reformulated in the form $\forall j = 0, \ldots, J,\ C_{\text{tail}} \leq V_j(\mathcal{R}_k, T_{\text{tail}})$.[1] All of such constraints are verified if $C_{\text{tail}} = min_{j=0,\ldots,J}\{V_j(\mathcal{R}_k, T_{\text{tail}})\}$. Hence the theorem follows. ◀

To avoid incurring in algebraic errors, the derivation of the equations reported in Table 2 has also been mechanized with the Wolfram Mathematica tool: the corresponding file is publicly available on-line [15].

3.2 Extensions

The method presented in the previous section can be extended to further increase the precision of the solution provided by Theorem 6. Two of such extensions are presented here, which are not discussed in details due to lack of space: a detailed discussion is available in an on-line appendix of this paper [15].

Extension 1. As argued in Lemma 3, the bound $\overline{\overline{dbf}}_{tail}(t)$ on the demand bound function of the tail reservation has been derived by considering zero as a lower-bound on C_{tail}. Leveraging a different lower bound C_{tail}^{LB}, it is possible to obtain a demand bound function tighter than

[1] In particular, terms V_1 and V_2 are derived from Lemma 2, terms $V_3, \ldots, V_{2+n_k^P}$ are derived from Lemma 5, and terms $V_{3+n_k^P}$ and $V_{4+n_k^P}$ are derived from Lemma 4.

Table 2 List of terms $V_j(\mathcal{R}_k, T_{\text{tail}})$ $(j = 0, \ldots, J)$ for Theorem 6, where $J = n_k + 2$ if $head(P_k) = false$ or $J = n_k + 3$ if $head(P_k) = true$.

Constraint for point $t = D_{\mathbf{tail}}$
$V_0(\mathcal{R}_k, T_{\text{tail}}) = min\left\{C_{\text{tail}}^{\texttt{MAX}}, min_{r_i \in \mathcal{R}_k}\{D_i\} - \epsilon\right\}$
Constraints for points $t = zT_{\mathbf{tail}} + D_{\mathbf{tail}}$, z = 1,2
$V_z(\mathcal{R}_k, T_{\text{tail}}) = \frac{1}{z}\left(zT_{\text{tail}} - \sum_{r_i \in \mathcal{R}_k^P} \overline{dbf_i}\left(zT_{\text{tail}} + C_{\text{tail}}^{\texttt{MAX}}\right)\right)$
Constraints for points $t = T_{\mathbf{i}}$, $\forall r_i \in \mathcal{R}_k^P$ $(z = 1, \ldots, n_k^P)$
$V_{2+z}(\mathcal{R}_k, T_{\text{tail}}) = \begin{cases} \frac{1}{2}\left(t - \sum_{r_i \in \mathcal{R}_k} \overline{dbf_i}(t)\right) & \text{if } T_{\text{tail}} \le t < 2T_{\text{tail}} \\ \frac{T_{\text{tail}}}{t+T_{\text{tail}}}\left(t - \sum_{r_i \in \mathcal{R}_k} \overline{dbf_i}(t)\right) & \text{if } t \ge 2T_{\text{tail}} \end{cases}$
Constraints for points $t = zT_{\mathbf{head}} + D_{\mathbf{head}}$, z = 1,2
$V_{2+n_k^P+z}(\mathcal{R}_k, T_{\text{tail}}) = \begin{cases} \frac{1}{2}\left(t - zC_{\text{head}} - \sum_{r_i \in \mathcal{R}_k^P} \overline{dbf_i}(t)\right) & \text{if } T_{\text{tail}} \le t < 2T_{\text{tail}} \\ \frac{T_{\text{tail}}}{t+T_{\text{tail}}}\left(t - zC_{\text{head}} - \sum_{r_i \in \mathcal{R}_k^P} \overline{dbf_i}(t)\right) & \text{if } t \ge 2T_{\text{tail}} \end{cases}$

$\overline{\overline{dbf}}_{tail}(t)$, thus increasing the precision of the result provided by Theorem 6. Such an improved function can be expressed as follows (its derivation is analogous to Lemma 3):

$$\overline{\overline{dbf}}_{tail}\left(t, C_{\text{tail}}^{LB}\right) = \begin{cases} 0 & \text{if } t < C_{\text{tail}}^{LB} \\ C_{\text{tail}} & \text{if } C_{\text{tail}}^{LB} \le t < T_{\text{tail}} + C_{\text{tail}}^{LB}, \\ 2C_{\text{tail}} & \text{if } T_{\text{tail}} + C_{\text{tail}}^{LB} \le t < 2T_{\text{tail}} + C_{\text{tail}}^{LB}, \\ 3C_{\text{tail}} + U_{tail}(t - 2T_{\text{tail}} - C_{\text{tail}}^{LB}) & \text{if } t \ge 2T_{\text{tail}} + C_{\text{tail}}^{LB} \end{cases}$$

A sequence of safe lower bounds C_{tail}^{LB} can be obtained in an iterative fashion. That is, it is possible to design an algorithm that starts with $C_{\text{tail}}^{LB} = 0$, use Theorem 6 to compute a value of C_{tail} and then sets $C_{\text{tail}}^{LB} = C_{\text{tail}}$. This latter value can in turn be used to repeat the computation of a new (possibly higher) value of C_{tail} by means of Theorem 6, and so on for a desired number λ of times or until converging to a desired tolerance. Surprisingly, the experiments reported in Section 5.1 show that just two iterations ($\lambda = 2$) provide a significant improvement.

Extension 2. The demand bound functions introduced in the previous section approximate the exact functions by considering a limited number of discontinuities and then a linear upper-bound. Specifically one, two, and three discontinuities have been adopted for the demand bound functions of the partitioned, head, and tail reservations, respectively. The precision of the method can be increased by refining such approximations, i.e., considering additional x discontinuities for each of such functions. Then, it is sufficient to iterate the number x from zero up to a desired value β applying Theorem 6 at each iteration, and finally taking the maximum value obtained for C_{tail}. Also in this case, the experiments reported in Section 5.1 show that just two iterations ($\beta = 2$) provide a significant improvement.

3.3 Implementation and Complexity

In this work, the methods proposed in the previous sections were derived to be used *on-line* for admitting a new reservation by means of C=D splitting. Therefore, considering the case

in which a set of reservations $\mathcal{R}_k$ is already allocated to P_k, the value of C_{tail} has to be computed for evaluating the possibility of allocating a tail reservation to P_k. In this case, the baseline approach presented in Section 3.1 allows implementing a *linear-time* algorithm for computing the C=D splitting. In fact, all the terms in the constraints $C_{\text{tail}} \leq V_j(\mathcal{R}_k, T_{\text{tail}})$ (see Table 2) that *do not depend* on T_{tail} can be pre-computed and stored in a table each time a reservation (partitioned or head) is allocated to P_k (e.g., as using dynamic programming): this operation can be done in $\mathcal{O}(n_k)$ time. Then, the resulting splitting algorithm only consists in computing (i) the upper bound $C_{\text{tail}}^{\texttt{MAX}}$, which can be done in constant time, (ii) the sum of demand bound functions in $V_1(\mathcal{R}_k, T_{\text{tail}})$ and $V_2(\mathcal{R}_k, T_{\text{tail}})$, which can be done in $\mathcal{O}(n_k)$ time, and (iii) the minimum required by Theorem 6, which can be done in $\mathcal{O}(n_k)$ time. Extension 1 discussed in Section 3.2 consists in repeating the baseline approach λ times, hence has complexity $\mathcal{O}(\lambda n_k)$: fixing a constant number of iterations λ, the resulting algorithm has again linear-time complexity. The same applies to Extension 2 with respect to the number of iterations β.

4 Load Balancing

This section presents a load balancing algorithm for managing the allocation and the splitting of the reservations under C=D semi-partitioned scheduling. The algorithm has been designed to be *as simple as possible* (to be practically used online) and employs a minimal number of re-allocations of the reservations. At a high level, the algorithm reacts to two events: (i) the *arrival* of a new reservation, where its *admission* must be evaluated by finding a proper allocation; and (ii) the *exit* of a reservation, which consists in performing some re-allocations in order to favor the admission of future reservations. It is worth observing that the admission of a new reservation cannot be generally done immediately when another reservation leaves the system or it is reconfigured during a re-allocation (so freeing some utilization bandwidth). This is because the leaving (or modified) reservation may have already affected the execution of the other reservations, and hence the system is subject to a *transient* (also referred to as mode-change by some authors). However, note that this issue is not specifically related to semi-partitioned scheduling, as it also occurs in uniprocessor systems [14, 36] (and hence under partitioned scheduling) and under global scheduling [35, 29]. Several solutions are available for analyzing the transient [14, 35], which allow deriving a safe bound on the time that must be waited before admitting a new reservation or let re-allocations to take effect. The design of improved methods that are tailored to C=D semi-partitioned scheduling is out of the scope of this paper and is left as future work. The following two sections discuss how to handle the arrival and the exit of a reservation. Then, Section 4.3 discusses some extensions that allow improving the performance of the load balancing algorithm, but at the cost of increasing its computational complexity.

4.1 Admission of a New Reservation

Whenever the system receives a request for admitting a new reservation r_i, the following operations are performed:

1. First, the algorithm tries to find a static allocation of r_i to a processor (i.e., as with standard partitioned scheduling) by using a partitioning heuristic. In particular, in our experiments the *best-fit* heuristic has been found to perform best. If a valid allocation is found, then r_i is admitted into the system.
2. If step 1 fails, then r_i is split into a head reservation r_{head} and a tail reservation r_{tail}. The method presented in Section 3 is used for computing the value of C_{tail} for each processor

P_k ($k = 1, \ldots, m$): the *maximum* of such values is selected as the budget of r_{tail} and the tail reservation is allocated to the corresponding processor. Then, the head reservation r_{head} is configured with budget $C_{head} = C_i - C_{\text{tail}}$ and relative deadline $D_{head} = T_i - C_{\text{tail}}$. Finally, the algorithm tries to allocate r_{head} to a processor following the same strategy used in step 1. If the allocation of the head reservation fails, then r_i is *rejected*; otherwise it is admitted into the system.

Note that both the steps require evaluating whether a reservation can safely be allocated to a processor. If a processor P_k contains only partitioned reservations, then a simple utilization test is adopted: this operation can be performed in constant time (storing the processor utilization in an incremental fashion). Otherwise, Theorem 1 is used, whose cost is $\mathcal{O}(n_k)$. As discussed in Section 3.3, the computation of C_{tail} for a processor P_k has $\mathcal{O}(n_k)$ complexity. Hence, the overall computational cost of the above operations is $\mathcal{O}(mn^{\text{MAX}})$, where $n^{\text{MAX}} = \max_{k=1,\ldots,m}\{n_k\}$.

4.2 Handling the Exit of a Reservation

Whenever a partitioned reservation $r_i \in \mathcal{R}_k^P$ (i.e., allocated to processor P_k) leaves the system, then the following operations are performed:

1. If $tail(P_k) = true$, let $r_j = \mathcal{F}(r_{k,tail})$ and try to allocate r_j to P_k after removing $r_{k,tail}$. That is, the algorithm tries to re-assemble the semi-partitioned reservation r_j. If r_j cannot be allocated to P_k, then the method presented in Section 3 is used for re-computing the value of C_{tail} for processor P_k to inflate the budget of $r_{k,tail}$, contextually decreasing the budget of the corresponding head reservation of r_j.
2. If $head(P_k) = true$, let $r_j = \mathcal{F}(r_{k,head})$ and try to allocate r_j to P_k after removing $r_{k,head}$.

Whenever a semi-partitioned reservation $r_i \in \mathcal{R}$ leaves the system, let $r_{k,head}$ (allocated to P_k) and $r_{z,tail}$ (allocated to P_z) be its head and tail reservations, respectively. Then, $r_{k,head}$ is removed from $\mathcal{R}_k$ and step 1 is performed on P_k. Also, $r_{z,tail}$ is removed from $\mathcal{R}_z$ and step 2 is performed on P_z. These operations require (i) checking at most twice whether a reservation can be allocated to a processor, which costs $\mathcal{O}(n^{\text{MAX}})$ time, and (ii) computing C_{tail} for a single processor (see step 1), which can also be performed in $\mathcal{O}(n^{\text{MAX}})$ time.

4.3 Extensions

This section describes three extensions that have been found to be effective for improving the performance of the load balancing algorithm.

(TAS) – Try all possible splits. Step 2 in Section 4.1 can be improved to increase the chances of admitting a new reservation by means of splitting. The algorithm can be modified as follows: for each processor P_k ($k = 1, \ldots, m$) in decreasing order with respect to their utilization $U^{(k)} = \sum_{r_i \in \mathcal{R}_k} U_i$, (i) compute the value of C_{tail} and use it for configuring the tail reservation, and (ii) try to allocate the resulting head reservation to a processor $\neq P_k$ (following the same strategy used in step 1). Since for each processor the algorithm tries to allocate the head reservation on the other processors, this approach increases the computational cost of the load balancing algorithm, which results $\mathcal{O}(m^2 n^{\text{MAX}})$.

(MS) – Multi-splitting. The splitting schemes discussed above consider the splitting in only two sub-reservations (one head and one tail). In the presence of several reservations that have a heavy utilization (i.e., greater than 0.5), the performance of the load balancing algorithm can be improved by employing an enhanced splitting scheme that considers multiple tail

reservations. The algorithm can be modified as follows. First, try to admit the new reservation r_i by splitting it into two sub-reservations. If this fails, let ${C_{\text{tail}}}^{(1)}, \ldots, {C_{\text{tail}}}^{(m)}$ be the sequence of the values of C_{tail} for each processor P_k $(k = 1, \ldots, m)$ in *decreasing* order. Then, find the maximum index $x < m$ such that $\sum_{j=1}^{x} {C_{\text{tail}}}^{(j)} < C_i$ and configure a head reservation with budget $C_{head} = C_i - \sum_{j=1}^{x} {C_{\text{tail}}}^{(j)}$ and relative deadline $D_{head} = T_i - \sum_{j=1}^{x} {C_{\text{tail}}}^{(j)}$. Subsequently, configure x tail reservations with budgets ${C_{\text{tail}}}^{(1)}, \ldots, {C_{\text{tail}}}^{(x)}$, allocate them to the corresponding processors and finally try to allocate the head reservations on one of the remaining processors. If this strategy fails, check also if $\sum_{j=1}^{x+1} {C_{\text{tail}}}^{(j)} \geq C_i$: in this case, the first x tail reservations are allocated as previously discussed and the head reservation is configured with $C_{head} = D_{head} = C_i - \sum_{j=1}^{x} {C_{\text{tail}}}^{(j)}$.[2] This approach does not increase the asymptotic complexity of the load balancing algorithm, but it increases the run-time overhead due the multiple migrations incurred by the multi-split reservations.

(RPR) – Re-allocate partitioned reservations. Whenever the algorithm does not find a valid allocation for a new reservation r_i, the chances of admitting r_i can be increased by trying to re-allocate a previously-allocated partitioned reservation. In particular, the following heuristic has been found to be effective while employing minimal re-allocations limited to a *single* reservation. For each processor P_k $(k = 1, \ldots, m)$, check if after de-allocating the partitioned reservation $r_j \in \mathcal{R}_k^P$ that has the highest utilization (i.e., $r_j \in \mathcal{R}_k^P \mid U_j = \max_{r_x \in \mathcal{R}_k^P} U_x$) it is possible to allocate r_i to P_k. If yes, then try to re-allocate r_j following steps 1 and 2 in Section 4.1. When the first valid re-allocation is found, r_j is re-allocated, r_i is allocated to P_k and the algorithm terminates. The computational complexity of this extension depends on the technique selected for splitting r_j. If the baseline approach (presented in Section 4.1) is used, then the algorithm has $\mathcal{O}(m^2 n^{\text{MAX}})$ complexity. Conversely, if this extension is adopted in conjunction with the TAS extension, then the algorithm has $\mathcal{O}(m^3 n^{\text{MAX}})$ complexity, while has $\mathcal{O}(m^2 n^{\text{MAX}})$ complexity if it is adopted in conjunction with the MS extension.

5 Experimental Results

This section presents the results of two large-scale experimental studies that have been conducted to evaluate the approach presented in this paper. The first study, discussed in Section 5.1, has been carried out to assess the performance of the approximate C=D splitting algorithms presented in Section 3 with respect to the exact algorithm proposed by Burns et al. in [13]. The second study, discussed in Section 5.2, has been carried out to evaluate the performance of the load balancing algorithms presented in Section 4 (adopted in conjunction with the C=D splitting approximation of Section 3), comparing them to G-EDF and partitioned EDF scheduling under different configurations.

5.1 C=D splitting: Approximated vs. Exact

A first experimental study has been carried to evaluate the utilization loss introduced by the approximate C=D splitting algorithms presented in Section 3 with respect to the exact Burns et al.'s [13] method. The study considers a single processor on which a set of reservations is

[2] This is a special case where a head reservation is allocated as if it would be a tail reservation, which allows overcoming the limitation that only at most one head reservation can be allocated to a processor.

already allocated and aims at computing the maximum zero-laxity budget (with different approaches) to allocate a tail reservation on the considered processor.

Reservation set generation. Given n reservations and a target utilization $U = \sum_{i=1}^{n} U_i$, the utilizations U_i of the n reservations are generated with the UUnifast algorithm [9]. For each reservation, the minimum inter-replenishment time T_i is randomly generated in the range [1, 1000] ms with uniform distribution and the budget is then computed as $C_i = U_i T_i$. Among the n reservations, one, say r_i, is randomly selected to be the head reservation: the relative deadline of r_i is then randomly generated with uniform distribution in the interval $[C_i + \beta(T_i - C_i), T_i]$, with $\beta = 0.9$ (that makes the interval representative of the configurations generated by C=D splitting). The remaining $n - 1$ reservations are partitioned and hence are configured with implicit deadline.

Experiments. The utilization U has been varied in the range $[0.05, 0.95]$ with step 0.05 and the number n of reservations has been varied from 2 to 20. [3] For each combination of these two parameters, 5000 reservation sets have been tested, for a total of almost 2 million reservation sets. For each reservation set $\mathcal{R}$, a random period T_{tail} for a tail reservation r_{tail} has been randomly generated in the range [1, 1000] ms with uniform distribution. Then, the value of C_{tail} such that the set of reservations $r_{tail} \cup \mathcal{R}$ can be safely EDF-scheduled on a single processor has been computed by (i) the exact method of [13] and (ii) the approximate methods proposed in this paper under four different approaches (all of them have *linear-time complexity*). Specifically, `BASELINE` is adopted to refer the approach presented in Section 3.1; `EXT1` to refer Extension 1 of Section 3.2 configured with $\lambda = 2$; `EXT2` to refer Extension 2 of Section 3.2 configured with $\beta = 2$; and `EXT1+EXT2` to refer `EXT1` and `EXT2` applied in conjunction. The approximate methods have then been compared to the exact method in terms of *utilization loss*: that is, given the exact value C_{tail}^{EXA} (by [13]) and an approximate value $C_{tail}^{\mathrm{APP}} \leq C_{tail}^{\mathrm{EXA}}$, the utilization loss introduced by the approximation is defined as $\left(C_{tail}^{\mathrm{EXA}}/T_{tail}\right) - \left(C_{tail}^{\mathrm{APP}}/T_{tail}\right)$.

The experimental results for four different configurations are reported in Figure 3. The complete set of results is available online [15]. As it can be observed from Figures 3(a) and 3(b), the utilization loss introduced by all the tested approaches decreases as the number of reservation increases, approaching values lower than 1% for $n > 18$. In particular, the combination of `EXT1` and `EXT2` (denoted as `EXT1+EXT2`) originates a very limited average utilization loss that is always lower than about 2%. Figures 3(c) and 3(d) show the dependency of the results on the utilization U: `EXT1` is particularly effective for low values of utilization, while `EXT2` increases its effectiveness as U increases. Also varying the utilization, the `EXT1+EXT2` approach exhibits a very good performance. By looking at the complete results collected in this study, it is possible to derive some guidelines for designing an efficient algorithm that – empirically speaking – introduces an average utilization loss *lower than 3%*, that is: use `EXT1+EXT2` for $n \in \{2, 3\}$, use `EXT1` for $n \geq 4$ and $U \leq 0.45$, and use `EXT2` for $n \geq 4$ and $U > 0.45$. Furthermore, `BASELINE` can be adopted in the presence of a higher number of reservations (e.g., $n > 12$).

[3] In the special case of a single reservation ($n = 1$), the *exact* maximum portion of zero-laxity budget that can be safely allocated to a processor can be computed by solving a simple equation (the details are available in an on-line appendix of this paper [15]). The number of reservations has been limited to 20 because the results show that the error introduced by the proposed approximation decreases as the number of reservations increases, approaching very low values for more than 20 reservations.

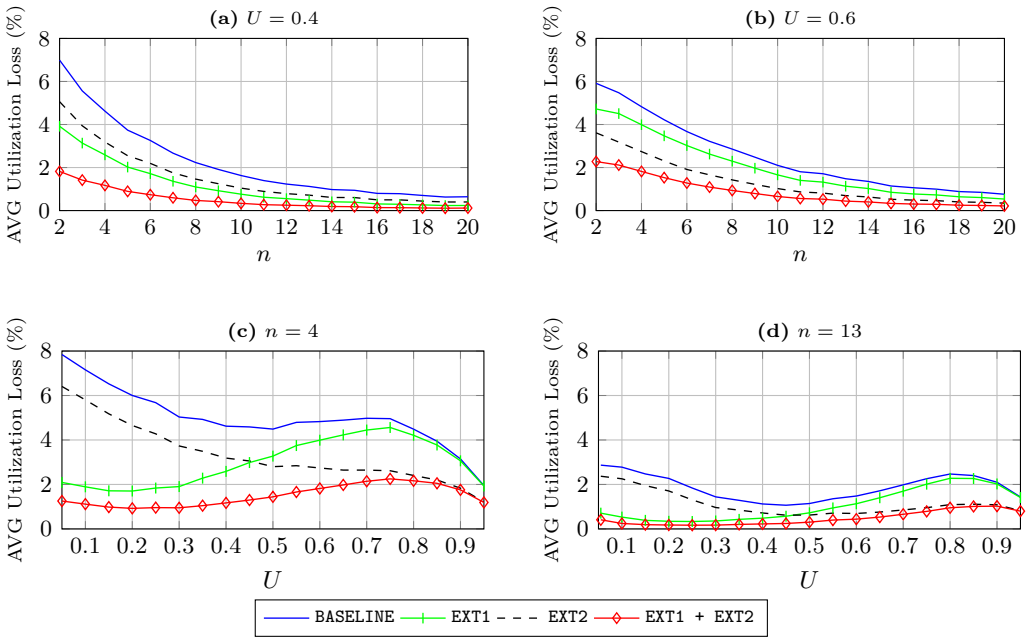


Figure 3 Average utilization loss introduced by the approximate algorithms for C=D splitting (presented in Section 3) as a function of the number of reservations n (insets (a) and (b)) and the utilization U (insets (c) and (d)). The results are related to four representative configurations identified by the fixed parameter reported in the caption above the graphs.

Running Times. Another experiment has been carried out to evaluate the running times of the proposed methods against the one of the exact C=D splitting algorithm. The tests have been executed on a machine equipped with an Intel Core i7-6700K @ 4.00GHz. The Microsoft VC++2015 compiler has been used to compile literal implementations (i.e., not designed for being extremely efficient) of the algorithms. The exact C=D splitting algorithm exhibited maximum running times in the order of a few seconds, with an increasing trend as a function of the utilization and the number of reservations. The proposed approximate methods showed running times under the precision offered by the Windows API for measuring the time executed by a process with performance counters.

5.2 Proposed Approach vs. G-EDF and P-EDF

A second experimental study has been done to evaluate the performance of C=D semi-partitioned scheduling managed by the load balancing algorithms presented in Section 4 – that in turn make use of the approximate splitting algorithms of Section 3 – against G-EDF and partitioned EDF (P-EDF) scheduling. For G-EDF scheduling, a relatively favorable condition has been considered in which the acceptance test is performed by combining four state-of-the-art polynomial-time tests (suitable for being executed on-line), which are: GFB [24], BAK [3], a polynomial-time approximation of LOAD [5, 22] and I-BCL [8] (configured with 3 iterations, as suggested by the authors). In other words, if *any* of these tests is passed, then a new reservation is admitted. For P-EDF, three common partitioning heuristics have been tested: first-fit, best-fit and worst-fit. The study is based on synthetic dynamic workload, which have been generated as follows.

Generation of Dynamic Workload. A *sequence* of N^E events is generated, where each event can be of type `ARRIVAL` or `EXIT`. An `ARRIVAL` event consists in a new reservation r_i that is tried to be admitted into the system. The utilization of each reservation has been generated with the *beta distribution* [4], which allows ensuring a given average U_{AVG} and a given variance U_σ, thus controlling the statistical validity across all the generated values. The distribution has been configured for generating the utilization of each reservation in the fixed range $[U_{min}, U_{max}] = [0.01, 0.9]$.

The minimum inter-replenishment time T_i of each reservation was generated in the range [1,1000] ms with uniform distribution, and the budget was then computed as $C_i = U_i T_i$. The `EXIT` event corresponds to the exit of a *random* reservation among those previously generated. Each sequence s is generated as follows: a random real number $x \in [0, 1]$ is generated N^E times with uniform distribution, then if $x \in [0, \Lambda]$, an `ARRIVAL` event is generated and enqueued to s, else an `EXIT` event is generated and enqueued to s. The term Λ is a variable *threshold* that controls the generation and has been set to $\Lambda = (1 - U_{opt}/m) + \psi(U_{opt}/m)$ with the following interpretation. The parameter U_{opt} is the utilization accepted by an *optimal* scheduling algorithm that has been stimulated by the previously-generated events. The first term in the definition of Λ is provided to increase the probability of generating an `ARRIVAL` event when the system load is low. The second term depends on a parameter $\psi \in [0, 1]$, which is used to control the tendency of a sequence to loading the processors; i.e., the higher ψ the higher the average load demanded by a sequence.

Experiments. The average U_{AVG} of the utilizations of the generated reservations has been varied in the range [0.1, 0.7] with step 0.05, whereas the variance U_σ has been varied in the range [0.05, 0.50], with step 0.05. The number of processors m has been varied in the set {4, 8, 16, 32} and the parameter ψ in the set {0.6, 0.7, 0.8, 0.9}. For each combination of the varied parameters, 1000 sequences of 10000 events have been generated, for a total of more than 4 billion events. Each generated sequence has been tested with G-EDF, P-EDF and the approaches proposed in this paper, measuring the *average* load accepted by each algorithm across the whole sequence. This measure is subsequently normalized to the hypothetical average load that would have been accepted by an optimal scheduling algorithm. This index expresses the quality of an algorithm in terms of acceptance rate (the higher the better and 100% corresponds to the performance of an optimal algorithm).[4] Figure 4 reports the results for six representative configurations with $\psi = 0.9$. The complete set of results is available online [15]. The labels `P-EDF-FF`, `P-EDF-WF` and `P-EDF-BF` in the legend indicate first-fit, worst-fit and best-fit partitioning, respectively; `C=D-LB` indicates the proposed approach based on load balancing with no extensions enabled; `C=D-LB+EXT` refers to `C=D-LB` applied in conjunction with all the extensions presented in Section 4.3. As can be observed from Figures 4(a), 4(b) and 4(c), the performance of the algorithms is significantly affected by the utilization of the tested reservations (as also previously observed in other works). The `C=D-LB+EXT` approach allows achieving high performance, keeping the average accepted load above the 87% in all the configurations, even in the presence of several reservations with high utilization. In particular, it allows achieving a performance improvement up to 40% over G-EDF and up to 25% over P-EDF. The algorithms based on P-EDF show relatively

[4] Note that the typical schedulability ratio metric makes little sense in the presence of dynamic workload, as the behavior of the different algorithms may significantly differ depending on the previous workload. For instance, an algorithm may reject a lot of "small" (low utilization) reservations because it previously accepted a "heavy" (high utilization) reservation.

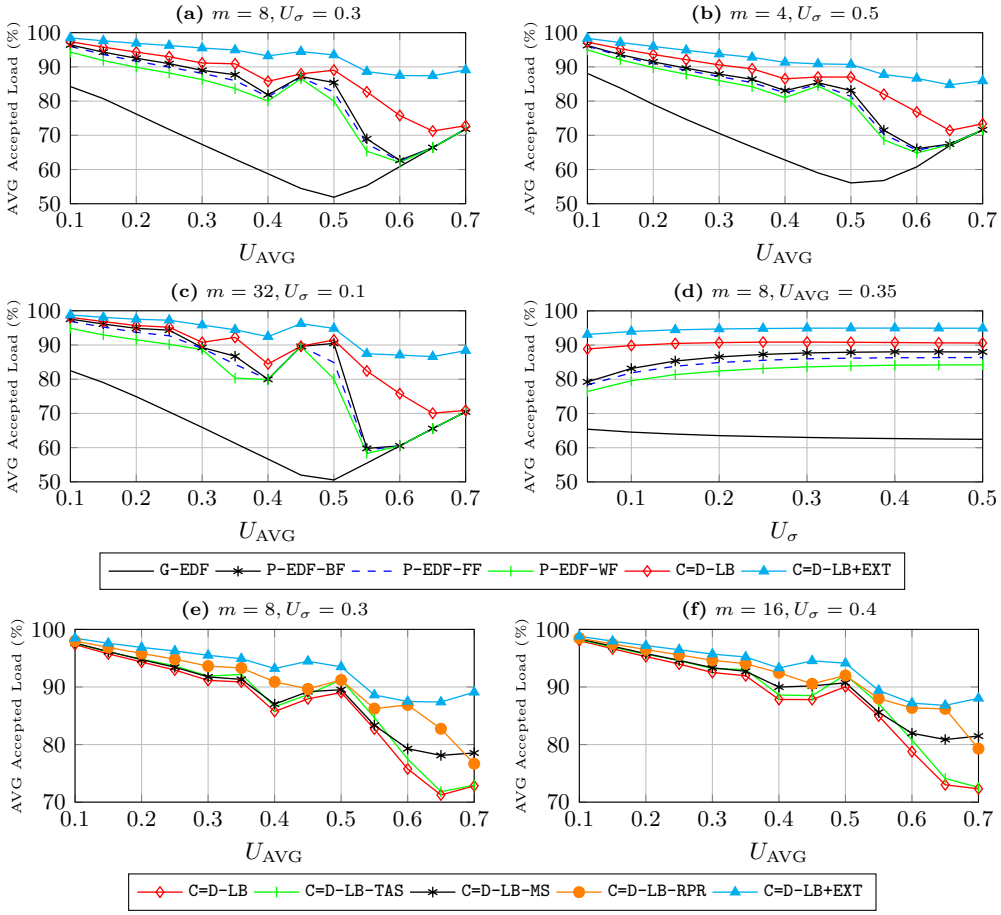


Figure 4 Average accepted load obtained by different scheduling approaches as a function of the average U_{AVG} of the utilizations of the generated reservations (insets (a), (b), (c), (e) and (f)) and the variance U_σ (inset (d)). The results are related to six representative configurations identified by the fixed parameters reported in the caption above the graphs.

good performance up to values of U_{AVG} that are close to 0.5. Surprisingly, basic partitioned scheduling with simple heuristics has been found to always outperform G-EDF. Figure 4(d) shows the dependency on the variance U_σ of the utilizations, which is found to be limited for the `C=D-LB+EXT`. In general, the proposed approach has been found to be robust to the presence of reservations with heterogeneous utilization. Finally, Figures 4(e) and 4(f) show the performance of the baseline load-balancing approach (`C=D-LB`) in conjunction with a single extension. As it can be observed from the graphs, the adoption of the RPR extension provides the highest performance. It is worth observing that the curves tend to show a non-monotonic behavior for the following reason. In the presence of high values of U_{AVG}, the acceptance or the rejection of a reservation corresponds to a significant difference in terms of instantaneous accepted load. Since this phenomenon also occurs in the case of an optimal scheduling algorithm (to which the performance is normalized), the processors tend to be less loaded across a sequence independently of the tested algorithm, which is a situation that favors non-optimal algorithms. The non-monotonic behavior of the performance of G-EDF has been found to depend on the combination of multiple acceptance tests; in particular, the I-BCL test tends to perform better than the others at high values of U_{AVG}.

6 Related Work

The problem of scheduling real-time workload on a multicore platform has been extensively investigated. A detailed discussion of all the results proposed in the literature is too vast to fit in the space available in this paper and readers interested in the topic can refer to the survey written by Davis and Burns [18]. For this reason, this section focuses on techniques based on semi-partitioned scheduling, which are more relevant to the proposed approach. Semi-partitioned scheduling has been firstly introduced by Anderson et al. [1] in 2005. Later, numerous semi-partitioned scheduling algorithms have been presented, including the proposals of Andersson et al. [2] and Kato et al. [25, 26, 27]. In 2011, Bastoni et al. [7] presented a thorough comparison of several semi-partitioned scheduling algorithms, illustrating their benefits with respect to other scheduling approaches. The method described in this paper has been motivated by a recent development due to Brandenburg and Gül [12], who showed that, by adopting clever task-allocation heuristics, the C=D splitting algorithm proposed by Burns et al. [13] allows achieving a near-optimal performance in the presence of static real-time workload. As in [12], the proposed approach also combines C=D scheduling with processor reservations, but in a more dynamic environment where reservations can be created and destroyed at runtime. Brandenburg and Gül also reports on a solid evaluation of the overhead introduced by C=D scheduling demonstrating its practical effectiveness. An overhead-aware analysis for semi-partitioned scheduling algorithms has been also proposed by Souto et al. [40]. The problem of taking online scheduling decisions for real-time workload has been investigated in many works. In particular, the difficulty of the problem has been discussed in the seminal work of Deterzous and Mok [19] and by Fisher et al. [21]. Lee and Shin [29] and Nélis et al. [35] proposed techniques for analyzing the effect of system transients under global scheduling, which may also be very useful C=D scheduling. Block and Anderson [11] and Block et al. [10] addressed dynamic workload in the context of task reweighting under partitioned and P-Fair scheduling, respectively. Manimaran and Murthy [23] proposed a scheduling algorithm for parallel and divisible real-time workload, however the authors did not provide any analysis and assessed the algorithm performance only by means of scheduling simulations. Mamat et al. [31, 30] addressed the problem of performing load balancing of aperiodic tasks with known arrival times in clustered-based computing. Other authors addressed the same problem in the context of uniprocessor systems: most relevant to us are the works by Stoimenov et al. [41], Santinelli et al. [38] and Nie et al. [34].

7 Conclusions and Future Work

This work addressed the problem of scheduling real-time dynamic workload upon a symmetric multiprocessor platform. The workload executes upon reservation servers that can arbitrarily join and leave the system, but each of them must pass an admission test before being admitted for execution. The reservations are scheduled under C=D semi-partitioned scheduling. A set of linear-time approximate methods for performing the C=D splitting have been presented to reduce the complexity of online scheduling decisions. Then, load balancing algorithms have been proposed for admitting new real-time workload in the system and performing limited workload re-allocation for facilitating the admission of future reservations. Both the contributions have been evaluated with large-scale experimental studies. The linear-time approximate splitting methods have been shown to originate a very limited utilization loss with respect to the exact technique previously proposed by Burns et al. [13]. In particular, a combination of such methods originates an average utilization loss that is below the 3%. The proposed scheduling approach based on C=D semi-partitioning and

load balancing algorithms allows achieving very high schedulability performance, with a consistent improvement over G-EDF and partitioned EDF scheduling with different bin-packing heuristics. As a representative result, the proposed approach allows keeping the average system load above the 87% in most of the tested scenarios, even in the presence of reservations with very high utilizations, with an improvement up to 40% over G-EDF and up to 25% over P-EDF. Considered the simplicity and the limited overhead of C=D semi-partitioned scheduling, as identified in [12], the results concluded in this work suggest its usage even in the presence of dynamic workload. Future work include the derivation of methods that are tailored to C=D semi-partitioned scheduling for handling system transients, the support for elastic reservations [14] to favor the admission of new workload and the consideration of synchronization issues. Moreover, additional research on load balancing algorithms may allow to further increase the performance of the proposed approach.

References

1. J. Anderson, V. Bud, and U.C. Devi. An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In *17th Euromicro Conference on Real-Time Systems (ECRTS 05)*, Palma de Mallorca, Spain, July 6-8 2005.
2. B. Andersson, K.Bletsas, and S. Baruah. Scheduling arbitrary-deadline sporadic task systems on multiprocessors. In *Real-Time Systems Symposium, 2008*, Barcelona, Spain, Nov 30 – Dec 3 2008.
3. T. P. Baker. Multiprocessor EDF and deadline monotonic schedulability analysis. In *24th IEEE International Real-Time Systems Symposium (RTSS 03)*, Cancun, Mexico, Dec, 3-5 2003.
4. N. Balakrishnan and V. B. Nevzorov. *A Primer on Statistical Distributions*. Wiley, 2003.
5. S. Baruah and T. P. Baker. Global EDF schedulability analysis of arbitrary sporadic task systems. In *Euromicro Conference on Real-Time Systems (ECRTS 08)*, Prague, Czech Republic, July, 2-4 2008.
6. S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-time systems*, 2(4):301–324, 1990.
7. A. Bastoni, B. B. Brandenburg, and J. H. Anderson. Is semi-partitioned scheduling practical? In *23rd Euromicro Conference on Real-Time Systems*, Porto, Portugal, July, 5-8 2011.
8. M. Bertogna, M. Cirinei, and G. Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on Parallel and Distributed Systems*, 20(4):553–566, April 2009.
9. E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1):129–154, May 2005.
10. A. Block, J. H. Anderson, and G. Bishop. Fine-grained task reweighting on multiprocessors. In *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*, Hong Kong, China, July, 17-19 2005.
11. A. Block and J. H. Anderson. Accuracy versus migration overhead in real-time multiprocessor reweighting algorithms. In *12th International Conference on Parallel and Distributed Systems – (ICPADS'06)*, Minneapolis, USA, July, 12-15 2006.
12. B. Brandenburg and M. Gül. Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations. In *Proceedings of the 37th IEEE Real-Time Systems Symposium (RTSS 2016)*, Porto, Portugal, November 29 – December 2 2016.

13 A. Burns, R. Davis, P. Wang, , and F. Zhang. Partitioned EDF scheduling for multiprocessors using a C=D task splitting scheme. *Real-Time Systems*, 48:3–33, 2012.

14 G. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3):289–302, March 2002.

15 D. Casini, A. Biondi, and G. Buttazzo. Semi-partitioned scheduling of dynamic real-time workload: A practical approach based on analysis-driven load balancing, online material. URL: `https://retis.sssup.it/~d.casini/sp-dyn`.

16 H. Cho, B. Ravindran, and E. D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 2006)*, Rio de Janeiro, Brazil, 5-8 December 2006.

17 T. Cucinotta, L. Abeni, L. Palopoli, and G. Lipari. A robust mechanism for adaptive scheduling of multimedia applications. *Journal ACM Transactions on Embedded Computing Systems*, 10(4):1–24, Nov. 2011.

18 R. Davis and A. Burns. A Survey of Hard Real-Time Scheduling for Multiprocessor Systems. *ACM Computing Surveys*, 43(4):35:1–35:44, 2011.

19 M. L. Dertouzos and A. K. Mok. Multiprocessor On-Line Scheduling of Hard-Real-Time Tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, Dec. 1989.

20 N. Fisher, T. P. Baker, and S. Baruah. Algorithms for determining the demand-based load of a sporadic task system. In *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)*, Sydney, Australia, Aug, 16-18 2006.

21 N. Fisher, J. Goossens, and S. Baruah. Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. *Real-Time Systems*, 45(1):26–71, June 2010.

22 N. W. Fisher. *The Multiprocessor Real-Time Scheduling of General Task Systems*. PhD thesis, University of North Carolina at Chapel Hill, 2007.

23 G.Manimaran and C. S. R. Murthy. An efficient dynamic scheduling algorithm for multiprocessor real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(3):312–319, Mar. 1998.

24 J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2):187–205, Sept. 2003.

25 S. Kato and N. Yamasaki. Portioned static-priority scheduling on multiprocessors. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, Miami, Florida, USA, April, 14-18 2008.

26 S. Kato and N. Yamasaki. Semi-partitioned fixed-priority scheduling on multiprocessors. In *15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2009)*, San Francisco, CA, USA, April 13-16 2009.

27 S. Kato, N. Yamasaki, and Y. Ishikawa. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *21st Euromicro Conference on Real-Time Systems*, Dublin, Ireland, July, 1-3 2009.

28 K. Konstanteli, T. Cucinotta, K. Psychas, and T. Varvarigou. Admission control for elastic cloud services. In *2012 IEEE Fifth International Conference on Cloud Computing*, Honolulu, HI, USA, June, 24-29 2012.

29 J. Lee and K. G. Shin. Schedulability analysis for a mode transition in real-time multi-core systems. In *Proceedings of the 2013 IEEE 34th Real-Time Systems Symposium (RTSS 2013)*, Washington, DC, USA, December 2013.

30 A. Mamat, Y. Lu, J. Deogun, and S. Goddard. An efficient algorithm for real-time divisible load scheduling. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium*, Stockholm, Sweden, April, 12-15 2010.

31 A. Mamat, J. Deogun Y. Lu, and S. Goddard. Real-time divisible load scheduling with advance reservations. In *2008 Euromicro Conference on Real-Time Systems*, Prague, Czech Republic, July, 2-4 2008.

32 E. Massa, G. Lima, P. Regnier, G. Levin, and S. Brandt. Quasi-partitioned scheduling: optimality and adaptation in multiprocessor real-time systems. *Real-Time Systems*, 52(5):566–597, 2016.

33 G. Nelissen, V. Berten, V. Nelis, J. Goossens, and D. Milojevic. U-EDF: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks. In *24th Euromicro Conference on Real-Time Systems (ECRTS 2012)*, Pisa, Italy, July 11-13 2012.

34 W. Nie, S. Zhou, K. J. Lin, and S. D. Kim. An on-line capacity-based admission control for real-time service processes. *IEEE Transactions on Computers*, 63(9):2134–2145, Sept. 2014.

35 V. Nélis, J. Marinho, B. Andersson, and S. M. Petters. Global-EDF scheduling of multimode real-time systems considering mode independent tasks. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS 2011)*, Porto, Portugal, July 6-8 2011.

36 J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26(2):161–197, March 2004.

37 P. Regnier, G. Lima, E. Massa, G. Levin, , and S. Brandt. Multiprocessor scheduling by reduction to uniprocessor: an original optimal approach. *Real-Time Systems*, 49(4):436–474, 2013.

38 L. Santinelli, G. Buttazzo, and E. Bini. Multi-moded resource reservations. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium*, Chicago, Illinois, USA, April, 11-13 2011.

39 I. Shin and I. Lee. Compositional real-time scheduling framework with periodic model. *Journal ACM Transactions on Embedded Computing Systems*, 7(3):1–39, April 2008.

40 P. Souto, P. B. Sousa, R. I. Davis, K. Bletsas, and E. Tovar. Overhead-aware schedulability evaluation of semi-partitioned real-time schedulers. In *21st International Conference on Embedded and Real-Time Computing Systems and Applications*, Hong Kong, China, August 19-21 2015.

41 N. Stoimenov, L. Thiele, L. Santinelli, and G. Buttazzo. Resource adaptations with servers for hard real-time systems. In *10th International Conference on Embedded Software (EMSOFT 2010)*, Scottsdale, Arizona, USA, October, 24-29 2010.

42 F. Zhang and A. Burns. Schedulability analysis for real-time systems with EDF scheduling. *IEEE Trans. Computers*, 58(9):1250–1258, 2009.

Cache-Conscious Offline Real-Time Task Scheduling for Multi-Core Processors*

Viet Anh Nguyen[1], Damien Hardy[2], and Isabelle Puaut[3]

1 **University of Rennes 1/IRISA, Rennes, France**
anh.nguyen@irisa.fr
2 **University of Rennes 1/IRISA, Rennes, France**
damien.hardy@irisa.fr
3 **University of Rennes 1/IRISA, Rennes, France**
isabelle.puaut@irisa.fr

Abstract

Most schedulability analysis techniques for multi-core architectures assume a single Worst-Case Execution Time (WCET) per task, which is valid in all execution conditions. This assumption is too pessimistic for parallel applications running on multi-core architectures with local instruction or data caches, for which the WCET of a task depends on the cache contents at the beginning of its execution, itself depending on the task that was executed before the task under study.

In this paper, we propose two scheduling techniques for multi-core architectures equipped with local instruction and data caches. The two techniques schedule a parallel application modeled as a task graph, and generate a static partitioned non-preemptive schedule. We propose an optimal method, using an Integer Linear Programming (ILP) formulation, as well as a heuristic method based on list scheduling. Experimental results show that by taking into account the effect of private caches on tasks' WCETs, the length of generated schedules is significantly reduced as compared to schedules generated by cache-unaware scheduling methods. The observed schedule length reduction on streaming applications is 11% on average for the optimal method and 9% on average for the heuristic method.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

Keywords and phrases real-time scheduling, cache-conscious scheduling, many-core architectures, ILP, static list scheduling

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.14

1 Introduction

Many-core platforms are increasingly used to execute high performance hard real-time applications since they provide enough computing power to satisfy the applications' demand while reducing their size, weight, and power requirements. However, guaranteeing the real-time constraints of safety-critical parallel applications on such platforms is quite challenging.

One important challenge is to precisely estimate the Worst-Case Execution Times (WCETs) of codes executing on multi-cores. Many WCET estimation methods have been designed in the past for single-core architectures [34]. Such techniques take into account both the program paths and the core micro-architecture. Extending them to multi-core architectures is challenging, because some hardware resources, such as caches or buses are

* This work was supported by PIA project CAPACITES (Calcul Parallèle pour Applications Critiques en Temps et Sûreté), reference P3425-146781.

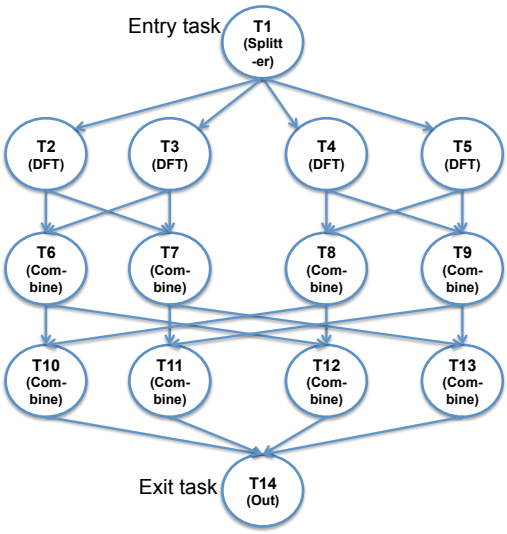


Figure 1 Task graph of a parallel version of a 8-input Fast Fourier Transform (FFT) application [2].

shared between cores, which makes the WCET of a task dependent on the tasks executing on the other cores [20, 10]. Additionally, on architectures with local caches, the WCET of a task depends on the cache contents when the task starts executing, which depends on the scheduling strategy. The WCET of one task is thus no longer unique. It depends on the execution context of the task (tasks executed before it, concurrent tasks), the execution context being defined by the scheduling strategy. We thus believe that scheduling strategies that are aware of the multi-core hardware (in particular local caches) have to be defined.

Many multi-core scheduling strategies assume a single context-insensitive WCET per task. In this paper, we propose instead *cache-aware* scheduling strategies, that take benefit of cache reuse between tasks. Each task has distinct WCET values depending on which other task has been executed before it on the same core (WCETs are context-sensitive). The proposed scheduling strategies map tasks on cores and schedules tasks on cores; the objective is to account for cache reuse to obtain the shortest schedules. For the scope of this paper, we focus on a single parallel application, modeled as a task graph, in which nodes represent tasks and edges represent dependence relations between them.

To further motivate our work, let us consider, as an example, a 8-input Fast Fourier Transform application. Its task graph is shown in Figure 1. This application contains both code and data reuse between tasks. For instance, T2 and T3 feature code reuse since they call the same function, and T2 and T6 feature data reuse since the output of T2 is the input of T6. On that example, we observe a WCET reduction of 10.7% on average when considering the cache affinity between pairs of tasks that may execute consecutively on the same core. The schedule length for that parallel application was reduced by 8% by using the Integer Linear Programming (ILP) technique presented in Section 4.1 as compared to its cache-agnostic equivalent.

In this paper, we propose two different methods to determine a static partitioned non-preemptive schedule aiming at minimizing the schedule length, for a parallel application by taking into account the variation of tasks' WCETs due to reuse of code and data between tasks. The first method is based on an Integer Linear Programming (ILP) formulation and produces optimal schedules. The second method is a heuristic technique that produces schedules very fast with schedule lengths close to the optimal ones.

The main contributions of our work are as follows:

- We argue and experimentally validate the importance of addressing the effect of private caches on tasks' WCETs in scheduling.
- We propose an ILP-based scheduling method and a heuristic scheduling method to statically find a partitioned non-preemptive schedule of a parallel application modeled as a directed acyclic graph.
- We provide experimental results showing, among others, that the proposed scheduling techniques result in shorter schedules than their cache-agnostic equivalent.

The rest of this paper is organized as follows. Section 2 surveys related work. Section 3 describes our system model and formulates the scheduling problem. Section 4 introduces the proposed ILP formulation and the proposed heuristic scheduling method. We experimentally evaluate our proposed scheduling methods in Section 5. Finally, we summarize the contents of the paper and provide directions for future work in Section 6.

2 Related work

Schedulability analysis techniques rely on the knowledge of the Worst-Case Execution Times of tasks. Originally designed for single-core processors, static WCET estimation techniques were extended recently to cope with multi-core architectures. Most research have focused on modeling shared resources (e.g., shared cache, shared bus, shared memory) in order to capture interferences between tasks which execute concurrently on different cores [15, 18, 6, 28, 13, 1]. Most extensions of WCET estimation techniques for multi-cores produce a WCET for a single task in the presence of concurrent executions on the other cores. By construction, those extensions do not account for caches effects between tasks as our scheduling techniques do. The scheduling techniques we propose have to rely on WCET estimation techniques to estimate the effect of local caches on tasks' WCETs.

Some WCET estimation techniques pay attention to the effect of private caches on WCETs. In [21], when analyzing the timing behavior of a task, Nemer et al. take into account the set of memory blocks that has been stored in the instruction cache (by the execution of previous tasks on the same core) at the beginning of its execution. Similarly, Potop-Butucaru and Puaut [26], assuming task mapping on cores known, jointly perform cache analysis and timing analysis of parallel applications. These two WCET estimation techniques assume task mapping on core and task schedule on each core known. In this paper, in contrast, task mapping and scheduling are selected to take benefit of cache reuse to have the shortest possible schedule length.

Much research effort has been spent on scheduling for multi-core platforms. Research on real-time scheduling for independent tasks is surveyed in [7]. This survey gives a taxonomy of multi-core scheduling strategies: global vs. partitioned vs. semi-partitioned, preemptive vs. non preemptive, time-driven vs. event-driven. The scheduling techniques we propose in this paper generate offline time-driven non-preemptive schedules. Most of the scheduling strategies surveyed in [7] are unaware of the hardware effects and consider a fixed upper bound on tasks' execution times. In contrast, the scheduling techniques we propose in this paper address the effect of private caches on tasks' WCETs. Our work integrates this effect in the scheduling and mapping problem by considering multiple WCETs for each task depending on their execution contexts (i.e. cache contents at the beginning of their execution).

Some scheduling techniques that are aware of hardware effects were proposed in the past. They include techniques that simultaneously schedule tasks and the messages exchanged between them [5, 30, 27]; such techniques take into consideration the Network-On-Chip

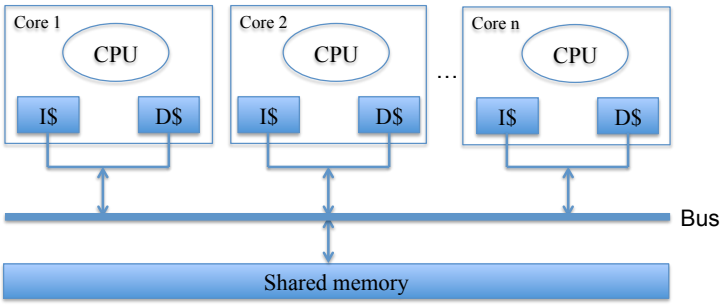


Figure 2 Considered multi-core architecture.

(NoC) topology in the scheduling process. Some other techniques aim at scheduling tasks in a way that minimizes the contentions when accessing shared resources (e.g., shared bus, shared caches) [4, 11, 9]. Besides, some approaches [35, 3, 23, 22, 24] schedule tasks according to execution models that guarantee temporal isolation between co-running tasks. In that way, scheduled tasks are guaranteed to be free from the contentions when accessing shared resources. In [29], Suhendra et al. consider data reuse between tasks to perform task scheduling for multi-core systems equipped with scratchpad memory (SPM); in their work, the most frequently accessed data are allocated in SPM to reduce the accesses latency to an off-chip memory. Our scheduling solutions in this paper differ from the above-mentioned previous works because we pay attention to the effect of private caches on tasks' WCETs. In our proposed scheduling methods, tasks are scheduled to get benefit from the effect of private caches.

Related studies also address the effect of private caches when scheduling tasks on multi-core architectures [33, 25, 31]. However, they are based on global and preemptive scheduling techniques, in which the cost of cache reload after being preempted or migrated has to be accounted for. Compared to these works, our technique is partitioned and non preemptive. We believe such a scheduling method allows to have better control on cache reuse during scheduling. Furthermore, [25] and [31] focus on single core architectures while our work target multi-core architectures.

3 System model and problem formulation

3.1 Hardware model

The class of architectures addressed in this work is the class of identical multi-core architectures, in which each core is equipped with a private instruction cache and a private data cache, as illustrated in Figure 2.

Clocks of all cores are assumed to be synchronized. Furthermore, it is assumed that the architecture is either (preferably) free from contention to access shared resources (shared bus, shared memory) or provides a way to bound the cost of interference, and in that latter case the cost of interference is included in tasks' WCETs. The issue of hardware resource sharing is thus considered outside the scope of this paper.

3.2 Task and execution model

Each application is modeled as a Directed Acyclic Graph (DAG) [16], as previously illustrated in Figure 1. A node in the DAG represents a task, denoted τ_i. An edge in the DAG represents

a precedence relation between the source and target tasks, as well as possibly a transfer of information between them. A task can start executing only when all its direct predecessors have finished their execution, and after all data transmitted from its direct predecessors are available. A task with no direct predecessor is an *entry* task, whereas a task with no direct successor is an *exit* task. Without loss of generality it is assumed that there is a single entry task and a single exit task per application.

The structure of the DAG is static, with no conditional execution of nodes. The volume of data transmitted along edges (possibly null) is known offline. Each task in the DAG is assigned a distinct integer identifier.

A communication for a given edge is implemented using transfers to and from a dedicated buffer located in shared memory. To simplify the description of the scheduling algorithms, the worst-case communication cost for a given edge is assumed constant, and is integrated in the WCETs of the sending and receiving tasks. However, we believe that non constant communication costs (e.g. depending on task mapping), could be easily integrated in the proposed scheduling algorithms.

Due to the effect of caches, each task τ_j is not characterized by a single WCET value but instead by a set of WCET values. The most pessimistic WCET value for a task, noted $WCET_{\tau_j}$, is observed when there is no reuse of cache contents loaded by a task executed immediately before τ_j. A set of WCET values noted $WCET_{\tau_i \rightarrow \tau_j}$ represent the WCETs of task τ_j when τ_j reuses some information, loaded in the instruction and/or data cache by a task τ_i that is executed immediately before τ_j on the same core.

3.3 Problem formulation

Our proposed scheduling methods take as inputs the number of cores of the architecture and the DAG of a parallel application decorated with WCET information for each task, and produce an offline time-driven partitioned non-preemptive schedule of the application. More precisely, the produced schedule for each core determines the start and finish times of all tasks assigned to the core.

4 Cache-conscious task scheduling methods

For solving the formulated scheduling problem, we propose two methods:

- An ILP formulation that allows to reach the optimal solution, i.e. the one that minimizes the application schedule length) (see Section 4.1);
- A heuristic method, based on list scheduling, that allows to find a valid schedule very fast and generally very close to the optimal one (see Section 4.2).

The notations used in the description of the scheduling methods are summarized in Table 1. The first block defines frequently used notations to manage the task graph. The second block defines integer constants, using upper case letters, used throughout the paper. Finally, the third block defines the variables, using lower case letters, used in the ILP formulation.

4.1 Cache-conscious ILP formulation (CILP)

In this section, we present the cache-conscious ILP formulation, noted CILP (for Cache-conscious ILP) hereafter. As mentioned before, one output of the scheduling methods is the mapping of tasks on cores. Since cores are identical, the *exact* core onto which a task is mapped does not matter. Based on that observation, CILP focuses on finding the sets

Table 1 Notations used in the proposed scheduling methods.

Symbol	Description	Data type
τ	The set of tasks of the parallel application	set
$dPred(\tau_j)$	The set of direct predecessors of τ_j	set
$dSucc(\tau_j)$	The set of direct successors of τ_j	set
$nPred(\tau_j)$	The set of tasks that are neither direct nor indirect predecessors of τ_j (τ_j excluded)	set
$nSucc(\tau_j)$	The set of tasks that are neither direct nor indirect successors of τ_j (τ_j excluded)	set
K	The number of cores of the processor	integer
$WCET_{\tau_j}$	The worst-case execution time of τ_j when not reusing cache contents	integer
$WCET_{\tau_i \to \tau_j}$	The worst-case execution time of τ_j when executing right after τ_i	integer
sl	The length of the generated schedule	integer
$wcet_{\tau_j}$	The worst-case execution time of τ_j	integer
st_{τ_j}	The start time of τ_j	integer
ft_{τ_j}	The finish time of τ_j	integer
f_{τ_j}	Indicates if τ_j is the first task running on a core or not	binary
$o_{\tau_i \to \tau_j}$	Indicates if τ_j is a co-located task of τ_i and executes right after τ_i or not	binary

of co-located tasks with their running orders. The assignment of sets of co-located tasks to cores is then straightforward (i.e. one set per core).

The objective function of CILP is to minimize the schedule length sl of the parallel application which is expressed as follows:

$$\text{minimize} \quad sl \tag{1}$$

Since the schedule length for the parallel application has to be larger than or equal to the finish time ft_{τ_j} of any task τ_j, the following constraint is introduced:

$$\begin{aligned} &\forall \tau_j \in \tau, \\ &\quad sl \geq ft_{\tau_j} \end{aligned} \tag{2}$$

The finish time ft_{τ_j} of a task τ_j is equal to the sum of its start time st_{τ_j} and its worst case execution time $wcet_{\tau_j}$:

$$\begin{aligned} &\forall \tau_j \in \tau, \\ &ft_{\tau_j} = st_{\tau_j} + wcet_{\tau_j} \end{aligned} \tag{3}$$

In the above equation, variable $wcet_{\tau_j}$ is introduced to model the variations of tasks' WCETs due to the effect of private caches and is computed as follows:

$$\begin{aligned} &\forall \tau_j \in \tau, \\ &wcet_{\tau_j} = f_{\tau_j} * WCET_{\tau_j} + \sum_{\tau_i \in \ nSucc(\tau_j)} o_{\tau_i \to \tau_j} * WCET_{\tau_i \to \tau_j} \end{aligned} \tag{4}$$

The left part corresponds to the case where task τ_j is the first task running on a core ($f_{\tau_j} = 1$). The sum in the right part corresponds to the case where the task τ_j is scheduled just after another co-located task τ_i ($o_{\tau_i \to \tau_j} = 1$). As shown later, only one binary variable among f_{τ_j} and variables $o_{\tau_i \to \tau_j}$ will be set by the ILP solver, thus assigning one and only one of the WCET values to τ_j depending on which other task is executed before it.

Constraints on tasks' start times

A task can be executed only when all of its direct predecessors have finished their execution. In other words, its start time has to be larger than or equal to the finish times of all its direct predecessors.

$$\begin{aligned} &\forall \tau_j \in \tau, \forall \tau_i \in dPred(\tau_j), \\ &st_{\tau_j} \geq ft_{\tau_i} \; if \; dPred(\tau_j) \neq \emptyset \\ &st_{\tau_j} \geq 0 \; otherwise \end{aligned} \tag{5}$$

In the above equation, when the task has no predecessor, its start time has to be larger than or equal to zero.

Furthermore, in case there is a co-located task τ_i scheduled right before τ_j, τ_j cannot start before the end of τ_i. In other words, the start time of τ_j has to be larger than or equal to the finish time of τ_i. Note that, τ_j can be scheduled only after a task τ_i that is neither its direct nor indirect successor.

$$\begin{aligned} &\forall \tau_j \in \tau, \forall \tau_i \in nSucc(\tau_j), \\ &st_{\tau_j} \geq o_{\tau_i \to \tau_j} * ft_{\tau_i} \end{aligned} \tag{6}$$

For linearizing equation (6), we use the classical big-M notation which is expressed as:

$$\begin{aligned} &\forall \tau_j \in \tau, \forall \tau_i \in nSucc(\tau_j), \\ &st_{\tau_j} \geq ft_{\tau_i} + (o_{\tau_i \to \tau_j} - 1) * M \end{aligned} \tag{7}$$

where M, is a constant[1] higher than any possible ft_{τ_j}.

Constraints on tasks' ordering

A task has at most one co-located task scheduled right after it, which is expressed as follows:

$$\begin{aligned} &\forall \tau_j \in \tau, \; if \; nPred(\tau_j) \neq \emptyset \\ &\sum_{\tau_i \in nPred(\tau_j)} o_{\tau_j \to \tau_i} \leq 1 \,. \end{aligned} \tag{8}$$

Note that, task τ_j can be only scheduled before task τ_i which is neither its direct nor indirect predecessor.

Furthermore, a task has one co-located task scheduled right before it which is neither its direct nor indirect successor or it is the first scheduled task, thus:

$$\begin{aligned} &\forall \tau_j \in \tau, \\ &\sum_{\tau_i \in nSucc(\tau_j)} o_{\tau_i \to \tau_j} + f_{\tau_j} = 1 \,. \end{aligned} \tag{9}$$

Finally, since the number of cores is K, the number of tasks that can be the first to be scheduled on cores is at most K:

$$\sum_{\tau_j \in \tau} f_{\tau_j} \leq K \,. \tag{10}$$

[1] For the experiments, M is the sum of all tasks' WCETs when not reusing cache contents, to ensure that M is higher than the finish time of any task.

The result of the mapping/scheduling problem after being solved by an ILP solver is then defined by two sets of variables. Task mapping is defined by variables f_{τ_j} and $o_{\tau_i \to \tau_j}$ that altogether define the set of co-located tasks and their execution order. The static schedule on every core is defined by variables st_{τ_j} and ft_{τ_j}, that define the start and finish time of the tasks assigned to that core.

4.2 Cache-conscious list scheduling method (CLS)

Finding an optimal schedule for a partitioned non-preemptive scheduling problem is NP-hard [14]. Therefore, we developed a heuristic scheduling method that efficiently produces schedules that are close to the optimal ones. The proposed heuristic method is based on list scheduling (see [17] for a survey of list scheduling methods).

The proposed heuristic method (CLS, for Cache-conscious List Scheduling) first constructs a list of tasks to be scheduled. Then, the list of tasks is scanned sequentially, and each task is scheduled without backtracking. When scheduling a task, all cores are considered for hosting the task and a schedule that respects precedence constraints is constructed for each. The core which allows the earliest finish time of the task is selected and the corresponding schedule is kept.

The ordering of the tasks in the list has to follow topological ordering such that precedence constraints are respected. Here, we select a topological order that also takes into account the WCETs of tasks. Since a task may have different WCETs according to the other task executed before it, we associate to each task a *weight* that approximates its WCET variation. The weight of a task τ_j, noted tw_{τ_j}, is defined as follows:

$$tw_{\tau_j} = \frac{1}{K} * min_{\tau_i \in nSucc(\tau_j)}(WCET_{\tau_i \to \tau_j}) + (1 - \frac{1}{K}) * WCET_{\tau_j} \,. \tag{11}$$

This formula integrates the likeliness that the WCET of task τ_j is reduced, which decreases when the number of cores increases. Different definitions of tasks weights were tested to take into account the WCET variation of tasks. Since we did not observe major difference between them, we selected this simple definition.

Given tasks' weights, the order of tasks in the list is determined based on two classical metrics, both respecting topological order. The first metric is called in the following *bottom level*. It defines for task τ_j the longest path from τ_j to the exit task (τ_j included), cumulating tasks' weights along the path:

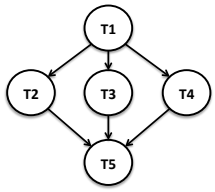
$$bottom\_level_{exit} = tw_{exit}$$
$$bottom\_level_{\tau_j} = max(bottom\_level_{\tau_i} + tw_{\tau_j}), \forall \tau_i \in dSucc(\tau_j) \tag{12}$$

The second metric is called *top level*. Symmetrically, it defines for task τ_j the longest path from the entry task to τ_j (τ_j excluded):

$$top\_level_{entry} = 0$$
$$top\_level_{\tau_j} = max(top\_level_{\tau_i} + tw_{\tau_i}), \forall \tau_i \in dPred(\tau_j) \tag{13}$$

As it will be shown in Section 5.2 none of the two metrics was shown to outperform the other for all task graphs, we thus kept both variations. In the following:

- CLS_BL refers to a sorting of tasks according to their bottom levels; in case of equality, their top levels is used to break ties; if a tie still exists, the task identifier is used to sort tasks.
- CLS_TL refers to a sorting of tasks according to their top levels, with bottom level and task identifier as tie breaking rules.
- CLS refers to the method, among CLS_BL and CLS_TL, resulting in the shortest schedule length for a given task graph.



τ_j \ τ_i	T1	T2	T3	T4	T5	Weight	Bottom Level
T1	10	–	–	–	–	10	42.5
T2	15	20	10	20	–	15	27.5
T3	20	15	25	25	–	20	32.5
T4	15	20	20	20	–	17.5	30
T5	15	10	10	10	15	12.5	12.5

Figure 3 Illustrative example for CLS_BL.

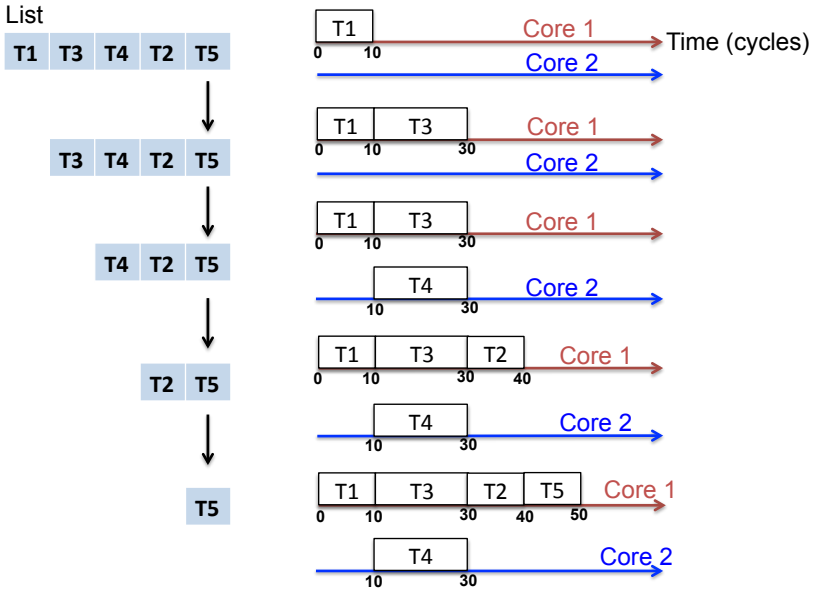


Figure 4 Illustration of CLS_BL.

We illustrate the execution of CLS_BL on a very simple task graph whose characteristics are given in Figure 3. The left part of Figure 3 gives the task graph. The right part gives the WCETs of tasks when not reusing cache contents ($WCET_{\tau_j}$, diagonal of the left part of the table) and values of $WCET_{\tau_i \to \tau_j}$; the next two columns give the weights and the bottom levels of tasks.

The execution of CLS_BL is illustrated in Figure 4 for a dual-core architecture. First, the tasks are ordered in a list according to their bottom levels. Then, the task at the head of the list (T1) is scheduled and T1 is removed from the list. Here, T1 is assigned to the first core. Next task in the list (here, T3) is then scheduled and removed from the list; T3 is mapped to the first core, which meets the precedence constraint between T1 and T3 and minimizes the finish time of T3 (date 30 on core 1 as opposed to date 35 on core 2). This process is repeated until the list is empty.

5 Experimental evaluation

In this section we evaluate the quality of generated schedules and the required time for generating them for the two proposed cache-conscious scheduling methods. We also evaluate the impact of several parameters, such as the number of cores, on the generated schedules.

Experimental conditions are described in Section 5.1. Experimental results are then detailed in Section 5.2.

5.1 Experimental conditions

5.1.1 Benchmarks

In our experiments, we use 26 benchmarks of the StreamIt benchmark suite [32]. StreamIt is a programming environment that facilitate the programming of streaming applications. We use that infrastructure to generate a sequential version of each benchmark in C++ and to get a representation of its Synchronous Data Flow graph (SDF). In the SDF graph, nodes represent filters or split-join structure and edges represent data dependencies between nodes. Each filter in the SDF consumes and produces a known amount of data. Each filter then has to be executed a certain number of times to balance the amount of data produced and consumed.

Each benchmark in the StreamIt benchmark suite starts with an initialization of the data for the initial execution of filters, followed by a steady state where the execution of filters is repeated. In our experiments, we focus on the execution of one iteration of the steady state. To obtain a task graph corresponding to our task model, we transformed manually the sequential version generated by the compiler to expose parallelism. We validated our transformations by systematically comparing the outputs of the sequential and parallel versions.

The characteristics of the obtained task graphs are summarized in Table 2. In the table, the maximum width of a task graph is defined as the maximum number of tasks with the same rank[2]. The maximum width defines the maximum parallelism in the benchmark. The average width is an average of the number of tasks for all ranks. The average width defines the average parallelism of the application. The higher the average width, the better the potential to benefit from a high number of cores. The depth of a task graph is defined as the longest path from the entry task to the exit task.

Additional information on the benchmarks is reported in Table 3. Reported information is the code size for the entire application, the average and standard deviation of code size per task, and the average amount of data communicated between tasks.

5.1.2 Hardware and WCET estimation

Our target architecture is the Kalray MPPA-256 machine [8], more precisely its first generation, named *Andey*. The Kalray MPPA-256 is a many-core platform with 256 compute cores organized in 16 compute clusters of 16 cores each. Each compute cluster has 2MB of shared memory. Each compute core is equipped with an instruction cache and a data cache of 8KB each, both set-associative with a Least Recently Used (LRU) replacement policy. An access to the shared memory, in case no contention occurs takes 9 cycles with 8 bytes fetched on each consecutive cycle [3].

Many techniques exist for WCET estimation [34] and could be used in our study to estimate WCETs and gains resulting from cache reuse. Since WCET estimation is not at the core of our scheduling methods, WCET values were obtained using measurements on the platform. Measurements were performed on one compute cluster, with no activity on the

[2] The rank of a task is defined as the longest path in term of the number of nodes to reach that task from the entry task.

Table 2 Summary of the characteristics of StreamIt benchmarks in our case studies.

Benchmark	No. of tasks	No. of Edges	Maximum graph width	Average graph width	Graph Depth
AudioBeam	20	33	15	3.3	6
Autocor	12	18	8	2.4	5
Beamformer	42	50	16	4.2	10
BitonicSort	50	66	4	2.1	24
Cfar	67	129	64	16.8	4
ChannelVocoder	264	512	201	33	8
Cholesky	95	148	11	2.3	41
ComparisonCounting	37	67	32	6.2	6
DCT	13	15	3	1.3	10
DCT_2D	10	11	2	1.3	8
DCT_2D_reference_fine	148	280	64	18.5	8
Des	247	468	48	9.9	25
FFT_coarse	192	254	64	12.8	15
FFT_fine_2	115	150	16	3.7	31
FFT_medium	131	204	16	4.7	28
FilterBank	34	45	8	2.4	14
FmRadio	67	85	20	5.6	12
IDCT	16	19	3	1.3	12
IDCT_2D	10	11	2	1.3	8
IDCT_2D_reference_fine	548	1072	256	68.5	8
Lattice	45	53	2	1.3	36
MergeSort	31	37	8	2.6	12
Oversampler	36	61	16	3.6	10
RateConverter	6	6	2	1.2	5
VectorAdd	5	4	2	1.3	4
Vocoder	71	94	7	2.2	32

other cores, providing fixed inputs for each task. The execution time of a task is retrieved using the machine's 64-bit timestamp counter counting cycles from boot time [8]. The effect of the timestamp counter on the execution time of a task turned out to be negligible. Since data caches are not coherent, they have to be flushed after each inter-core communication. We further observed that thanks to the determinism of the architecture, when running a task several times, in the same execution context, the execution time is constant (the same behavior was observed in [19]). For each task, we record its execution time when not reusing cache contents, as well as when executed after any possible other task.

Table 4 summarizes the obtained execution times. This table shows the average and standard deviation of tasks' WCET without cache reuse. It also shows the weighted average WCET reduction for each benchmark, computed as follows. For each task τ_j we calculate its average WCET reduction in percent:

$$r_{\tau_j} = 100 * \frac{\sum_{\tau_i \in nSucc(\tau_j)} \frac{WCET_{\tau_j} - WCET_{\tau_i \rightarrow \tau_j}}{WCET_{\tau_j}}}{|nSucc(\tau_j)|} \tag{14}$$

Since tasks with low WCETs tend to have high WCET reductions although they have low impact on schedule length, we weighted each value by its WCET, yielding to the following

Table 3 The size of code and communicated data for each benchmark (average μ and standard deviation σ).

Benchmark	Code size (Bytes)		Communicated data (Bytes)
	Entire application	μ / σ of tasks	μ
AudioBeam	38076	1458 / 1897	6
Autocor	12348	1014 / 538	66
Beamformer	333424	1879 / 718	10
BitonicSort	57952	1154 / 503	9
Cfar	181808	1906 / 5513	6
ChannelVocoder	302012	881 / 159	6
Cholesky	87336	916 / 667	22
ComparisonCounting	33564	893 / 840	20
DCT	23180	1188 / 831	8
DCT_2D	17248	1704 / 1101	9
DCT_2D_reference_fine	120392	724 / 145	12
Des	212808	783 / 185	12
FFT_coarse	418576	2161 / 467	52
FFT_fine2	122428	1060 / 574	9
FFT_medium	178660	1358 / 408	27
FilterBank	101096	834 / 192	4
FmRadio	374812	1072 / 679	4
IDCT	24336	1507 / 1239	7
IDCT_2D	17608	1740 / 1063	9
IDCT_2D_reference_fine	452924	802 / 154	7
Lattice	37812	817 / 274	5
MergeSort	34208	1088 / 366	16
Oversampler	56824	777 / 115	4
RateConverter	12348	683 / 247	11
VectorAdd	3080	593 / 148	4
Vocoder	125272	1064 / 1319	6

definition of weighted average reduction:

$$wr = \frac{\sum_{\tau_j \in \tau}(r_{\tau_j} * WCET_{\tau_j})}{\sum_{\tau_j \in \tau} WCET_{\tau_j}} \tag{15}$$

5.1.3 Experimental environment

We use *Gurobi optimizer* version 6.5 [12] for solving our proposed ILP formulation. The solving time of the solver is limited to 20 hours. The ILP solver and heuristic scheduling algorithms are executed on 3.6 GHz Intel Core i7 CPU with 16GB of RAM.

Table 4 Tasks' WCETs (average μ / standard deviation σ) and weighted average WCET reduction.

Benchmark	WCET in cycles (μ/σ)	Weighted average WCET reduction
AudioBeam	1479.0 / 2869.6	13.3
Autocor	3163.0 / 1855.1	5.5
Beamformer	4896.9 / 2950.2	4.5
BitonicSort	678.0 / 391.6	22.8
Cfar	2767.0 / 11612.7	13.0
ChannelVocoder	8084.5 / 26265.9	3.8
Cholesky	1512.5 / 3152.3	10.7
ComparisonCounting	1249.6 / 1477.5	14.4
DCT	718.3 / 685.0	19.1
DCT_2D	812.7 / 741.4	18.6
DCT_2D_reference_fine	1072.6 / 1519.2	17.1
Des	893.2 / 1236.2	23.4
FFT_coarse	3465.9 / 3062.3	9.8
FFT_fine_2	745.5 / 469.6	19.5
FFT_medium	1470.7 / 1456.3	11.6
FilterBank	3634.0 / 3701.0	4.6
FmRadio	2802.5 / 2652.1	5.5
IDCT	687.7 / 632.9	21.2
IDCT_2D	805.6 / 743.5	18.7
IDCT_2D_reference_fine	1538.5 / 3864.9	14.9
Lattice	515.6 / 381.8	28.6
MergeSort	1010.4 / 662.1	17.4
Oversampler	4195.3 / 684.5	6.5
RateConverter	19779.0 / 34471.5	0.9
VectorAdd	923.8 / 979.6	20.1
Vocoder	804.1 / 1227.8	15.8

5.2 Experimental results

5.2.1 Benefits of cache-conscious scheduling

In this sub-section, we show that cache-conscious scheduling, should it be implemented using an ILP formulation (CILP) or a heuristic method (CLS), yields to shorter schedules than equivalent cache-agnostic methods. This is shown by comparing how much is gained by CILP as compared to NCILP, the same ILP formulation as CILP except that cache effects are not taken into account (variable $wcet_{\tau_j}$ is systematically set to the cache-agnostic WCET, $WCET_{\tau_j}$). The gain is evaluated by the following equation, in which sl stands for the schedule length:

$$gain = \frac{sl_{NCILP} - sl_{CILP}}{sl_{NCILP}} * 100. \tag{16}$$

The gain is also evaluated using a similar formula for the heuristic method CLS (shorter schedule among CLS_BL and CLS_TL) as compared to its cache-agnostic equivalent NCLS.

Results are reported on Figures 5 and 6 for a 16 cores architecture. In Figure 5, only the benchmarks for which the optimal solution was found in a time budget of 20 hours are depicted. These figures show that both CILP and CLS reduce the length of schedules, and

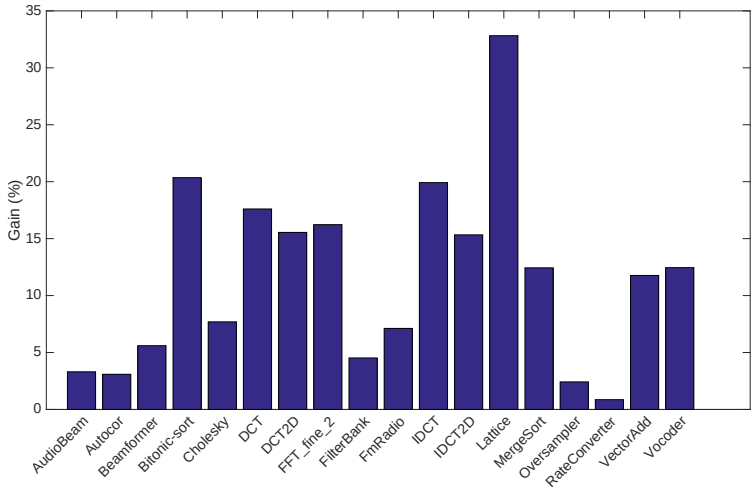


Figure 5 Gain of CILP as compared to NCILP ($gain = \frac{sl_{NCILP} - sl_{CILP}}{sl_{NCILP}} * 100$) on a 16 cores system.

this for all benchmarks. The gain is 11% on average for CILP and 9% on average for CLS. The higher reductions are obtained for the benchmarks with the higher weighted WCET reduction as defined in Table 2.

5.2.2 Comparison of optimal (CILP) and heuristic (CLS) scheduling techniques

In this sub-section, we compare CILP and CLS according to two metrics: the quality of the generated schedules, estimated through their lengths (the shorter the better) and the time required to generate the schedules. All results are obtained on a 16 cores system.

Table 5 gives the lengths of generated schedules (sl_{CILP} and sl_{CLS}), the run time of schedule generation and the gap (in percent) between the schedule lengths, computed by the following formula:

$$gap = \frac{sl_{CLS} - sl_{CILP}}{sl_{CILP}} * 100. \tag{17}$$

The shorter the gap, the closer CLS is from CILP. The gap between CLS and CILP is given only when CILP finds the optimal solution in a time budget of 20 hours.

The table shows that CLS offers a good trade-off between the efficiency and the quality of its generated schedules. CLS generates schedules very fast as compared to CILP (i.e., about 1 second for the biggest task graph *IDCT_2D_reference_fine* which contains 548 tasks). When scheduling big task graphs, such as *DES*, *ChannelVocoder* and *IDCT_2D_reference_fine*, CILP is unable to find the optimal solution in 20 hours. When CILP finds the optimal solution, the gap between CILP and CLS is very small (0.7% on average).

The highest gap (7.3%) is observed for the *Lattice* benchmark. The reason is that *Lattice* contains a reuse pattern (illustrated in Figure 7) where reuse is higher between indirect predecessors than between direct predecessors. For example, the reduction of the WCET of T6 when executing directly after T1 (37.3%) is higher than when executing directly after

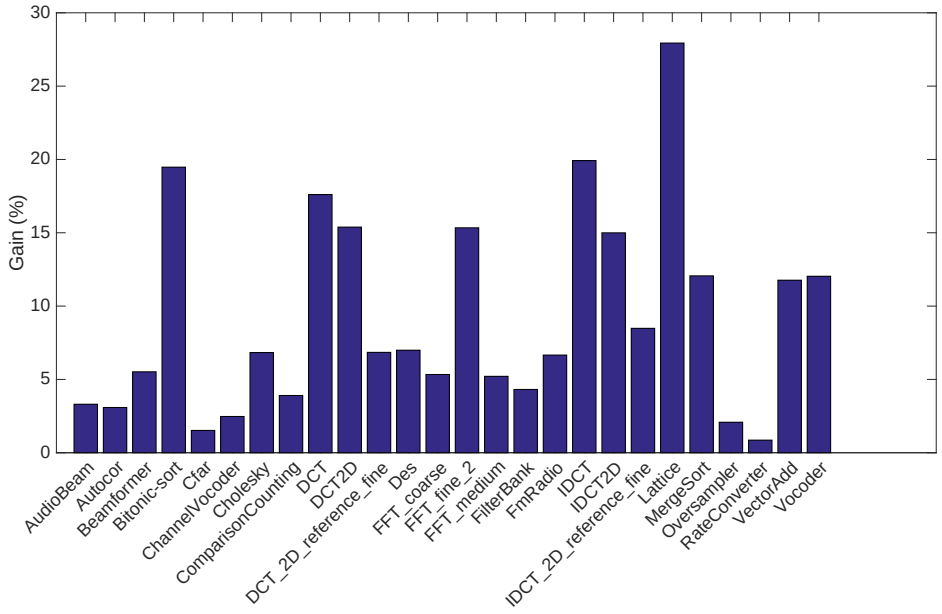


Figure 6 Gain of CLS as compared to NCLS ($gain = \frac{sl_{NCLS} - sl_{CLS}}{sl_{NCLS}} * 100$) on a 16 cores system.

T5 (22.6%). Similarly, the reduction of the WCET of T9 when executing directly after T4 (50.1%) is higher than when executing directly after T7 (31.3%). For such an application, the static sorting of CLS never places indirect precedence-related tasks (for which the higher reuse occurs) contiguously in the list, and then does not fully exploit the cache reuse present in the application.

5.2.3 Impact of the number of cores on the gain of CLS against NCLS

In this section, we evaluate the gain in term of schedule length of CLS against its cache-agnostic equivalent when varying the number of cores. The results are depicted in Figure 8 for a number of cores from 2 to 64.

In the figure, we can observe that whatever the number of cores, CLS always outperforms NCLS, meaning that our proposed method is always able to take advantage of the WCET reduction due to cache reuse to reduce the schedule length. Another observation is that the gain decreases when the number of cores increases, up to a given number of cores. This behavior is explained by the fact that when increasing the number of cores, the tasks are spread among cores which provides less opportunity to exploit cache reuse since exploiting the parallelism of the application is more profitable. However, even in that situation, the reduction of the schedule length achieved by CLS against NCLS is most of the time significant.

5.2.4 Impact of the number of cores on schedule length

In this section, we study the impact of the number of cores on schedule length for the CLS scheduling technique. This is expressed by depicting the ratio of the schedule length on one

Table 5 Comparison of CILP and CLS (schedule length and run time of schedule generation).

Benchmarks	sl_CILP	sl_CLS	time_CILP (s)	time_CLS (s)	gap (%)
AudioBeam	20746^o	20746	< 1	< 1	**0.00**
AutoCor	17455^o	17455	< 1	< 1	**0.00**
Beamformer	29778^o	29803	2	< 1	**0.08**
BitonicSort	15445^o	15616	78	< 1	**1.11**
Cfar	120370^f	120476	72000	< 1	
ChannelVocoder	x	302933	72000	< 1	
Cholesky	113474^o	114539	< 1	< 1	**0.94**
ComparisonCounting	19618^f	19640	72000	< 1	
DCT	6613^o	6613	< 1	< 1	**0.00**
DCT2D	5856^o	5867	< 1	< 1	**0.19**
DCT_2D_reference_fine	33337^f	32572	72000	< 1	
Des	100632^f	98596	72000	< 1	
FFT_coarse	x	134873	72000	< 1	
FFT_fine_2	30007^o	30326	66984	< 1	**1.06**
FFT_medium	89782^f	87144	72000	< 1	
FilterBank	47083^o	47185	15	< 1	**0.22**
FmRadio	29969^o	30125	4376	< 1	**0.52**
IDCT	7268^o	7268	< 1	< 1	**0.00**
IDCT2D	5803^o	5826	< 1	< 1	**0.40**
IDCT_2D_reference_fine	x	101970	72000	1	
Lattice	13253^o	14217	< 1	< 1	**7.27**
MergeSort	14501^o	14563	1	< 1	**0.43**
Oversampler	39143^o	39279	8	< 1	**0.35**
RateConverter	117278^o	117278	< 1	< 1	**0.00**
VectorAdd	3704^o	3704	< 1	< 1	**0.00**
Vocoder	32759^o	32916	9	< 1	**0.48**
Average					**0.72**

–) x: no solution is found in 20 hours
–) f: feasible solution is found
–) o: optimal solution is found

core sl_{1Cores} to the schedule length on n cores sl_{nCores}: $slRatio_{nCores} = \frac{sl_{1Core}}{sl_{nCores}}$. Results are given in Figure 9 for a number of cores $n = 2, 4, 8, 16, 32$ and 64. The higher the ratio, the better CLS is able to exploit the multi-core architecture for a given benchmark.

The figure shows that for all benchmarks the ratio increases up to a certain number of cores and then reaches a plateau. The plateau is reached when the benchmark does not have sufficient parallelism to be exploited by the scheduling algorithm, which is correlated to the width of its task graph as presented in Table 2.

It can be noticed that for some benchmarks (*ChannelVocoder*, *DCT_2D_reference_fine*, *FFT_coarse* and *IDCT_2D_reference_fine*) the plateau is never reached because these benchmark have too much parallelism for the number of cores. Even if the average width is below 64, we observe for these benchmarks that the maximal width is above 64 and up to 256 for *IDCT_2D_reference_fine* which explains why the plateau is not reached for these benchmarks.

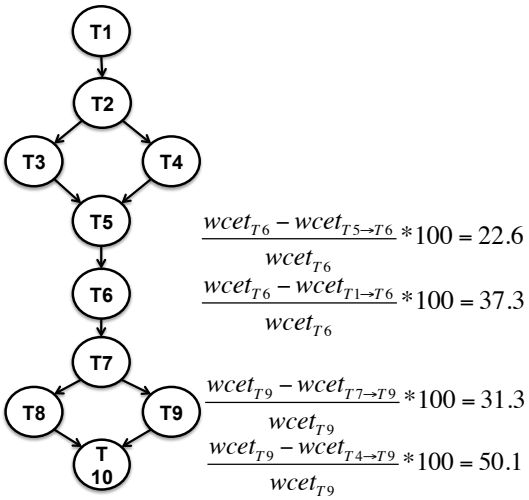


Figure 7 The reuse pattern found in the *Lattice* benchmark.

An exception is observed for *RateConverter* where there is absolutely no improvement. The graph of this benchmark is an almost linear chain of tasks with only a pair of tasks that may execute in parallel. However, there is cache reuse between these two tasks and thus the best schedule, whatever the number of available cores, is obtained when assigning all tasks to the same core.

Finally, for most benchmarks, the ratio does not increase linearly with a slope of 1, because task graphs contain precedence relations.

5.2.5 Comparison of schedule lengths for CLS_TL and CLS_BL

In this sub-section, we study the impact of the sorting technique of the list scheduling technique on the quality of schedules. For each benchmark, Figure 10 depicts the ratio of the length of the schedules generated by CLS_TL to that of CLS_BL as $slRatio_{CLS\_TL/CLS\_BL} = \frac{sl_{CLS\_TL}}{sl_{CLS\_BL}}$. A ratio of 1 indicates that the two techniques generate schedules with identical length. Results are given for different numbers of cores (4, 8, 16, 32 and 64).

The figure shows that there is no method which dominates the other for all benchmarks. Furthermore, the lengths of schedules generated by CLS_TL and CLS_BL are most of the time very close to each other.

There is a significant difference between CLS_TL and CLS_BL only in two cases, *ChannelVocoder* on 4 cores and *FmRadio* on 8 cores. The distances between the lengths of the schedules generated by CLS_TL and CLS_BL in these cases are then 3% and 8% respectively. It shows that in some special cases, the change in the order of tasks in the list significantly affects the mapping of tasks, hence the quality of generated schedules. Since both CS_TL and CLS_BL generate schedules very fast, we have throughout this paper always used both and selected the best result obtained.

5.2.6 Cost of estimating cache reuse

The information given in Table 6 allows to evaluate the cost of estimating cache reuse (estimation of values of $WCET_{\tau_i \to \tau_j}$) for the StreamIt benchmarks. The table reports for each benchmark its number of tasks, the number of task pairs that may be executed one after

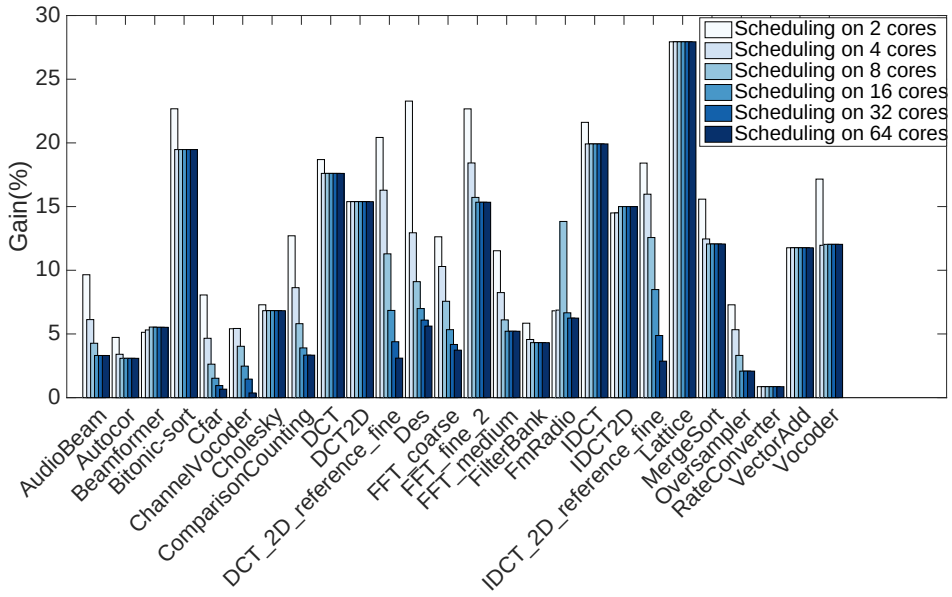


Figure 8 Impact of the number of cores on the gain of CLS against NCLS.

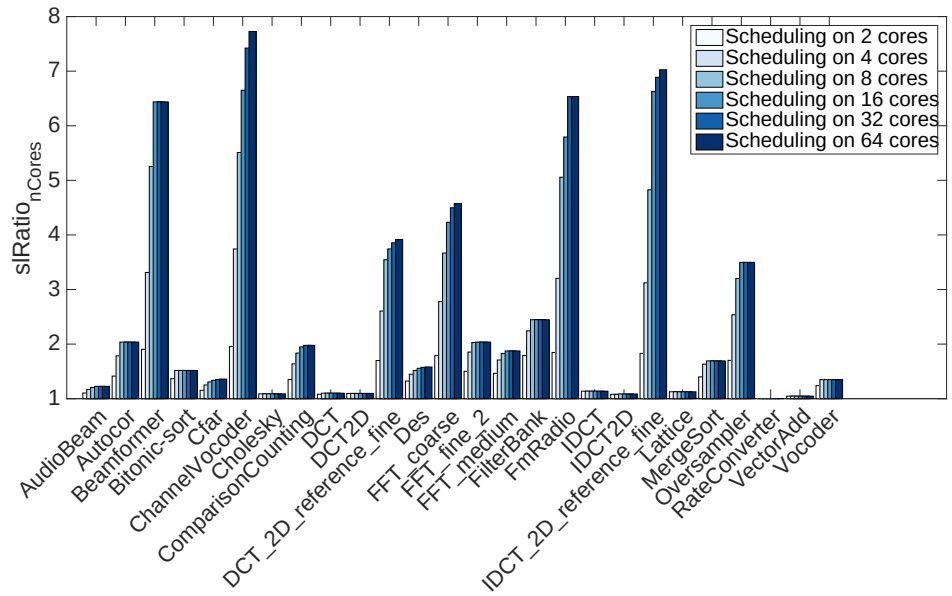


Figure 9 Impact of the number of cores on schedule length (CLS method).

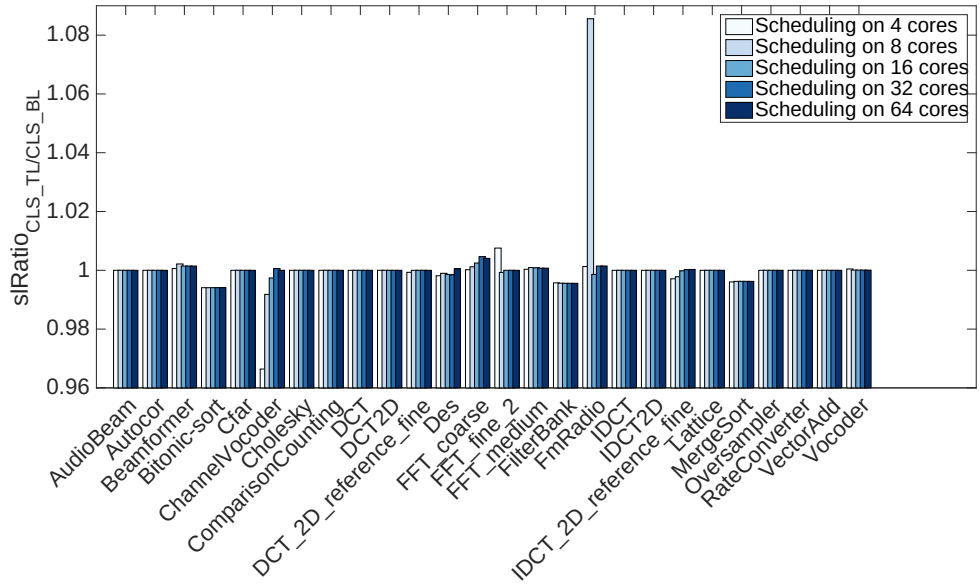


Figure 10 Comparison of schedule lengths for CLS_TL and CLS_BL.

Table 6 Cost of estimating cache reuse.

Benchmark	No. of tasks	No. of possible pairs	Profiling time (s)
AudioBeam	20	295	5
AutoCor	12	94	5
Beamformer	42	1326	7
BitonicSort	50	1341	7
Cfar	67	4227	11
ChannelVocoder	264	57481	170
Cholesky	95	7108	18
ComparisonCounting	37	1162	7
DCT	13	83	5
DCT_2D	10	47	5
DCT_2D_reference_fine	148	15414	49
Des	247	38185	135
FFT_coarse	192	34428	97
FFT_fine_2	115	7799	23
FFT_medium	131	10043	37
FilterBank	34	774	6
FmRadio	67	3841	11
IDCT	16	126	5
IDCT_2D	10	47	5
IDCT_2D_reference_fine	548	219238	625
Lattice	45	999	7
MergeSort	31	688	6
Oversampler	36	785	6
RateConverter	6	16	5
VectorAdd	5	11	5
Vocoder	71	2961	11

the other due to precedence constraints, and the time taken to evaluate all WCET values using measurements. The number of task pairs to be considered depends on the structure of the task graph. The worse observed profiling time is 10 minutes for the most complex benchmark structure *IDCT_2D_reference_fine*.

6 Conclusion

In this paper, we proposed two cache-aware scheduling techniques for applications modeled as task graphs, that generate static time-driven partitioned non-preemptive schedules for multi-core platforms. We proposed an ILP formulation as well as a heuristic scheduling method. Experimental results show that by taking into account the effect of private caches on tasks' WCETs our proposed scheduling methods produce better schedules (in term of schedule length reduction) than their cache-agnostic equivalent. The proposed heuristic scheduling method shows a good trade-off between efficiency and the quality of generated schedules. In the future, a direct extension would be to test if exploiting reuse between more than two consecutive tasks is worth the extra complexity. We will also extend our work to deal with contentions on shared hardware resources.

Acknowledgments. The authors would like to thank Dumitru Potop-Butucaru, Benjamin Rouxel, and Biswabandan Panda for comments on earlier versions of this paper.

References

1 S. Altmeyer, R. I. Davis, L. Indrusiak, C. Maiza, V. Nelis, and J. Reineke. A generic and compositional framework for multicore response time analysis. In *International Conference on Real Time and Networks Systems*, RTNS '15, pages 129–138, 2015.

2 J. H. Bahn, J. Yang, and N. Bagherzadeh. Parallel FFT algorithms on network-on-chips. In *Fifth International Conference on Information Technology: New Generations (ITNG 2008)*, pages 1087–1093, 2008.

3 M. Becker, D. Dasari, B. Nikolic, B. Akesson, V. Nélis, and T. Nolte. Contention-free execution of automotive applications on a clustered many-core platform. In *28th Euromicro Conference on Real-Time Systems, ECRTS*, pages 14–24, 2016.

4 J. M. Calandrino and J. H. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. In *21st Euromicro Conference on Real-Time Systems*, pages 194–204, 2009.

5 T. Carle, M. Djemal, D. Potop-Butucaru, R. de Simone, and Z. Zhang. Static mapping of real-time applications onto massively parallel processor arrays. In *Proceedings of the 2014 14th International Conference on Application of Concurrency to System Design*, ACSD '14, pages 112–121, 2014.

6 S. Chattopadhyay, A. Roychoudhury, and T. Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems*, SCOPES '10, pages 6:1–6:10, 2010.

7 R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):35:1–35:44, 2011.

8 B. D. de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager. Time-critical computing on a single-chip massively parallel processor. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 97:1–97:6, 2014.

9 H. Ding, Y. Liang, and T. Mitra. Shared cache aware task mapping for WCRT minimization. In *8th Asia and South Pacific Design Automation Conference, ASP-DAC*, pages 735–740, 2013.

10 G. Fernandez, J. Abella, E. Quiñones, C. Rochange, T. Vardanega, and F. J. Cazorla. Contention in multicore hardware shared resources: Understanding of the state of the art. In *14th International Workshop on Worst-Case Execution Time Analysis*, OpenAccess Series in Informatics (OASIcs), pages 31–42, 2014.

11 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*, EMSOFT '09, pages 245–254, 2009.

12 Inc. Gurobi Optimization. Gurobi optimizer reference manual, 2015.

13 D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS*, pages 68–77, 2009.

14 H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Trans. Comput.*, 33(11):1023–1029, 1984.

15 T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. Static analysis of multi-core tdma resource arbitration delays. *Real-Time Syst.*, 50(2):185–229, 2014.

16 Y.-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. In *Journal of Parallel and Distributed Computing*, volume 59, pages 381–422, 1999.

17 Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

18 Y. Liang, H. Ding, T. Mitra, A. Roychoudhury, Y. Li, and V. Suhendra. Timing analysis of concurrent programs running on shared cache multi-cores. *Real-time Systems*, 48(6):638–680, 2012.

19 V. Nélis, P. M. Yomsi, and L. M. Pinho. The variability of application execution times on a multi-core platform. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, OpenAccess Series in Informatics (OASIcs), pages 1–11, 2016.

20 V. Nélis, P. M. Yomsi, L. M. Pinho, J. C. Fonseca, M. Bertogna, E. Quiñones, R. Vargas, and A. Marongiu. The challenge of time-predictability in modern many-core architectures. In *14th International Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OpenAccess Series in Informatics (OASIcs)*, pages 63–72, 2014.

21 F. Nemer, H. Cassé, P. Sainrat, and A. Awada. Improving the worst-case execution time accuracy by inter-task instruction cache analysis. In *IEEE Second International Symposium on Industrial Embedded Systems, SIES*, pages 25–32, 2007.

22 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 *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '11, pages 269–279, 2011.

23 Q. Perret, P. Maurère, E. Noulard, C. Pagetti, P. Sainrat, and B. Triquet. Mapping hard real-time applications on many-core processors. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, RTNS '16, pages 235–244. ACM, 2016.

24 Q. Perret, P. Maurère, E. Noulard, C. Pagetti, P. Sainrat, and B. Triquet. Temporal isolation of hard real-time applications on many-core processors. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 37–47, 2016.

25 G. Phavorin, P. Richard, J. Goossens, T. Chapeaux, and C. Maiza. Scheduling with preemption delays: Anomalies and issues. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS '15, pages 109–118, 2015.

26 D. Potop-Butucaru and I. Puaut. Integrated worst-case execution time estimation of multicore applications. In *13th International Workshop on Worst-Case Execution Time Analysis*, volume 30, pages 21–31, 2013.

27 W. Puffitsch, E. Noulard, and C. Pagetti. Off-line mapping of multi-rate dependent task sets to many-core platforms. *Real-Time Systems*, 51(5):526–565, 2015.

28 H. Rihani, M. Moy, C. Maiza, R. I. Davis, and S. 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, 2016.

29 V. Suhendra, C. Raghavan, and T. Mitra. Integrated scratchpad memory optimization and task scheduling for mpsoc architectures. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 401–410, 2006.

30 P. Tendulkar, P. Poplavko, I. Galanommatis, and O. Maler. Many-core scheduling of data parallel applications using SMT solvers. In *17th Euromicro Conference on Digital System Design, DSD*, pages 615–622, 2014.

31 C. Tessler and N. Fisher. BUNDLE: real-time multi-threaded scheduling to reduce cache contention. In *IEEE Real-Time Systems Symposium, RTSS*, pages 279–290, 2016.

32 W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 365–376, 2010.

33 B. C. Ward, A. Thekkilakattil, and J. H. Anderson. Optimizing preemption-overhead accounting in multiprocessor real-time systems. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 235:235–235:243, 2014.

34 R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem: Overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008.

35 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, 2012.

Optimal Dataflow Scheduling on a Heterogeneous Multiprocessor With Reduced Response Time Bounds

Zheng Dong[1], Cong Liu[2], Alan Gatherer[3], Lee McFearin[4], Peter Yan[5], and James H. Anderson[6]

1 **The University of Texas at Dallas, Dallas, TX, USA**
zheng@utdallas.edu
2 **The University of Texas at Dallas, Dallas, TX, USA**
cong@utdallas.edu
3 **America Wireless Access Laboratory, Huawei Technologies Co. Ltd, USA**
alan.gatherer@huawei.com
4 **America Wireless Access Laboratory, Huawei Technologies Co. Ltd, USA**
lee.mcfearin@huawei.com
5 **America Wireless Access Laboratory, Huawei Technologies Co. Ltd, USA**
peter.yifey.yan@huawei.com
6 **University of North Carolina at Chapel Hill, Chapell Hill, NC, USA**
anderson@cs.unc.edu

Abstract

Heterogeneous computing platforms with multiple types of computing resources have been widely used in many industrial systems to process dataflow tasks with pre-defined affinity of tasks to subgroups of resources. For many dataflow workloads with soft real-time requirements, guaranteeing fast and bounded response times is often the objective. This paper presents a new set of analysis techniques showing that a classical real-time scheduler, namely earliest-deadline-first (EDF), is able to support dataflow tasks scheduled on such heterogeneous platforms with provably bounded response times while incurring no resource capacity loss, thus proving EDF to be an optimal solution for this scheduling problem. Experiments using synthetic workloads with widely varied parameters also demonstrate that the magnitude of the response time bounds yielded under the proposed analysis is reasonably small under all scenarios. Compared to the state-of-the-art soft real-time analysis techniques, our test yields a 68% reduction on response time bounds on average. This work demonstrates the potential of applying EDF into practical industrial systems containing dataflow-based workloads that desire guaranteed bounded response times.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

Keywords and phrases real-time scheduling, schedulability, heterogeneous multiprocessor

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.15

1 Introduction

In many applications, such as traffic monitoring [12], trajectory tracking [14], and modem signal processing, dataflows periodically arrive in the form of data streams (e.g., continuous image frames). A stream processing system is required to handle such data streams often in a soft real-time fashion, e.g., results must be processed within a short and bounded time period. For example, in traffic monitoring systems and voice data processing systems in

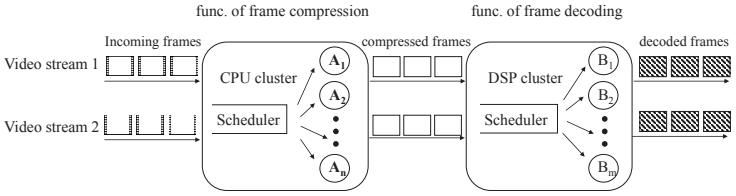


Figure 1 An example video stream processing system.

cellular networks, results become meaningless if they cannot be processed and returned to the end user within a bounded time window.

To handle continuously arrived dataflows with fast and bounded response times, heavily heterogeneous computing systems [2] containing various types of accelerators (e.g., digital signal processors–DSP) are used in many stream processing systems, such as Apache Storm and its successor Twitter Heron [13], StreamCloud [10] and Apache Spark Streaming [26]. In these systems, multiple types of resources are used to process different functionalities using dataflow tasks as the data source with pre-defined affinity of tasks to subgroups of resources (i.e., each functionality is designated to be processed on a specific type of computing resource, such as CPU, DSP, or one of the many types of hardware-accelerators). One significant challenge faced by such systems is the need to develop a multi-resource, real-time scheduler, which can correctly support a maximum load of dataflows that may fully utilize all system resources, along with the accompanying analytical schedulability test algorithms, which validate at design time whether a set of dataflows can feasibly run with bounded response times.

Fig. 1 shows a simplified example video stream application processed in a heterogeneous computing system. There are two types of computing resources: a cluster of CPUs and a cluster of DSPs [9]. Each video stream is independent and has two subtasks, where the first subtask is to compress incoming video frames and is dedicated to be executed on CPUs, while the second subtask is frame decoding and is dedicated to be executed on DSPs. In this example, each video stream has a period of $\frac{1}{24}s$, i.e., each stream periodically generates frames at a rate of 24 frame per second (FPS). For each frame, the output of the first subtask processed by CPUs is then fed to DSPs as data input to execute the second subtask. The multi-resource scheduler determines how to schedule and allocate resources to various subtasks belonging to different dataflow tasks.

Much recent work has been conducted on scheduling soft real-time (i.e., task response times must be provably bounded) tasks on a homogeneous multiprocessor [5, 7]. However, analysis is lacking for supporting soft real-time applications developed using the dataflow formalism in a heterogeneous computing system with pre-defined task affinity to resources. In this paper, we address this lack of support by presenting new schedulability analysis techniques for a classical real-time scheduler, namely earliest-deadline-first (EDF). The resulting scheulability test proves EDF to be an optimal solution for this heterogeneous dataflow scheduling problem in terms of system utilization. *That is, EDF can correctly support any set of dataflow tasks with probably bounded response times that may even require all types of computing resources to be fully utilized, thus incurring no capacity loss on any resource.*

Specifically, we present new schedulability analysis techniques showing that any dataflow task system is schedulable under EDF with bounded response times if $U^k_{sum} \leq M_k, 1 \leq k \leq m$, where m denotes the number of resource types, M_k denotes the number of processors of the k^{th} resource type, and U^k_{sum} denotes the total resource utilization required by all dataflow

tasks on processors of the k^{th} resource type. This schedulability test implies EDF's optimality due to no capacity loss on any type of resource. We have also conducted extensive experiments involving synthetic workloads with widely varied parameters, which demonstrate that the magnitude of the response time bound yielded under our schedulability test is reasonably small under all scenarios. Compared to the state-of-the-art soft real-time analysis techniques, our test yields a 68% reduction on response time bounds on average.

2 Related Work

Task scheduling has been a well-studied problem in many parallel and distributed systems (e.g., [21, 4, 24, 27]). A plethora of schedulers have been developed and used in practice, e.g., the Fair Scheduler, the Capacity Schedular, and delay scheduling. These scheduling strategies, however, cannot be applied to solve our problem of scheduling dataflow tasks with provably bounded response times on a heterogeneous computing platform, because they target at different application models and different performance objectives. On the other hand, the problem of scheduling task systems on multiprocessors with bounded response times [19, 3, 16] (e.g., several recent dissertations are produced focusing on this topic [6, 15]) or hard deadlines [7, 8, 23, 22, 11, 1] have received much recent attention. Such works mostly focus on supporting on sporadic task systems on homogeneous multiprocessors, where the traditional sporadic task model is a basic recurrent task model that is much simpler than the dataflow task model studied in this paper (both models will be specifically defined in later sections).

In recent works [25, 17], a release-enforcer technique has been proposed to schedule DAG-based tasks on both homogeneous [17] and heterogeneous multiprocessors [25], showing that no utilization loss can be achieved under GEDF scheduling. Specifically, the release-enforcer technique forces a DAG-based task to release jobs in a sporadic manner through arbitrarily delaying jobs' releases, thus eliminating the need of directly handling DAG-induced precedence constraints. A DAG-based task set can thus be transformed to an ordinary sporadic task set. Unfortunately, this technique forces any job's release to be delayed to the worst-case completion time of any of its predecessor jobs, thus causing a rather pessimisitic response time bound. Fundamentally differeant from this technique, this paper presents a new set of analysis techniques that *directly* analyze jobs' response time on its original GEDF schedule, without any artificial delay of jobs' releases. The resulting response time bounds are thus much improved (Sec. 7 shows the advantage of our proposed techniques over applying the existing technique [25] in terms of respones time bounds).

3 The Dataflow Task Model

In this section, we introduce the studied dataflow task system model, which is directly motivated by the workloads seen in a couple of industrial cellular network systems.

We consider the problem of scheduling n dataflow tasks on a heterogeneous computing platform consisting of $m > 1$ types of processors, where each type of processor is designated to execute a specific functionality. Let M^k denote the number of processors of type k ($1 \leq k \leq m$). Each dataflow task is specified as a 2-tuple (G_i, P_i), where G_i is a chain of m subtasks, and P_i is a positive real number. We discuss these parameters below.

- Since a dataflow task may continuously generate data streams that need to be processed (i.e., "jobs"), the period P_i denotes the minimum amount of time that must elapse between the releases of successive jobs of τ_i. That is, if a job is released at time t, then the next

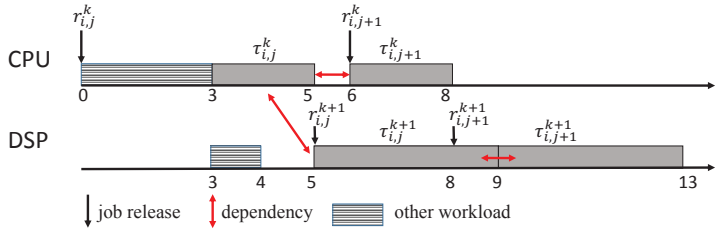


Figure 2 Two kinds of dependencies.

job of τ_i may not be released prior to the time instance $t + P_i$. Let $\tau_{i,j}$ denote the j^{th} job released by τ_i and $r_{i,j}$ denote the release time of $\tau_{i,j}$.

- The chain of subtasks G_i is specified as $\tau_i^1, ..., \tau_i^m$, where τ_i^k denotes the k^{th} subtask of dataflow task τ_i. Subtask τ_i^k is designated to be executed on processors of type k. Any j^{th} job released by τ_i is thus composed by a chain of m subjobs, each of which belongs to the corresponding subtask. Any subjob $\tau_{i,j}^k$ is characterized by its corresponding subtask τ_i^k's worst-case execution time (WCET) e_i^k [20]. The edge connecting any two subtasks represents dependencies between the released subjobs. That is, any subjob $\tau_{i,j}^k$ (i.e., the j^{th} subjob belonging to subtask τ_i^k) must complete execution before subjob $\tau_{i,j}^{k+1}$ can begin execution. Moreover, any subjob $\tau_{i,j}^k$ must complete execution before subjob $\tau_{i,j+1}^k$ can begin execution, which is the next released subjob belonging to the same subtask. These two types of dependencies among subjobs are illustrated in Fig. 2 and by the following example,

▶ **Example 1.** The above dataflow task model studied in this paper is mainly motivated by many industrial applications seen in practice. Consider the video stream application introduced in Fig. 1 as an example to illustrate the Dataflow Task Model. A video stream task τ_i periodically generates frames at a rate of 24 FPS, thus $P_i = \frac{1}{24}s$. Any j^{th} job (i.e., j^{th} frame in this example) $\tau_{i,j}$ needs to be processed using two functionalities, frame compression and frame decoding, which corresponds to the two subtasks of τ_i that are designated to be processed on two types of computing resources, CPUs and DSPs respectively. Thus $G_i = (\tau_i^1, \tau_i^2)$. As illustrated in Fig. 2, there are two types of dependencies among subjobs: (i) There is dependency between any two subjobs $\tau_{i,j}^k$ and $\tau_{i,j}^{k+1}$. This is because for each frame, the functionality of frame decoding can only happen after the same frame has been compressed. (ii) There is dependency between any two subjobs $\tau_{i,j}^k$ and $\tau_{i,j+1}^k$ (or $\tau_{i,j}^{k+1}$ and $\tau_{i,j+1}^{k+1}$). This is because in terms of each functionality, the video frames must be processed in sequential order since they are captured sequentially.

In order to apply the EDF[1] scheduling algorithm, we assign a deadline parameter D_i to each dataflow task τ_i, where $D_i = P_i$. Thus, any job $\tau_{i,j}$ has a deadline at time $r_{i,j} + D_i$. Under EDF, jobs are prioritized by deadlines, where jobs with shorter deadlines have higher priorities. Note that we apply EDF scheduling on all types of processors. Thus, all subjobs of any job $\tau_{i,j}$ inherit the priority of $\tau_{i,j}$, regardless of the type of processor on which they are executed. We allow job preemption and migration (but only migrating to a processor of the correct type), i.e., a higher-priority job that requires type-k processors may preempt

[1] We have verified that both preemption and the dynamic job-level priority-based EDF scheduler can possibly be implemented in a couple of industrial dataflow-based cellular network systems seen in practice.

the execution of a lower-priority job on a type-k processor and this lower-priority job may migrate to another type-k processor if such a processor is available.[2]

The utilization of τ_i on type-k processors is denoted u_i^k (i.e., the utilization of subtask τ_i^k), which is given by $\frac{e_i^k}{P_i}$, and we require

$$u_i^k \leq 1; \tag{1}$$

for otherwise the response time of τ_i^k may grow unboundedly. Since a task periodically releases jobs, the concept of task utilization is important as it characterizes the percentage of a single processor's capacity a task (or a subtask) requires in the long term.

The total utilization of a task system τ on type-k processors is defined as $U_{sum}^k(\tau) = \sum_{i=1}^{n} u_i^k$. We place no constraint on total utilization except that

$$U_{sum}^k(T) \leq M^k. \tag{2}$$

A scheduling algorithm is considered to be optimal if it can schedule any task system τ that satisfies Eq. (2) with bounded response times guaranteed for all tasks in τ.

The goal of this paper is to derive response time bounds for a dataflow task system scheduled under EDF. The response time of a task is defined to be the maximum response time of any of its released jobs. A job's response time is defined to be $f_{i,j} - r_{i,j}$, where $f_{i,j}$ denotes the time at which $\tau_{i,j}$ completes and $r_{i,j}$ is the release time of $\tau_{i,j}$. Instead of deriving the response time bound directly, we derive a tardiness bound for any dataflow task scheduled under EDF. The tardiness of a job $\tau_{i,j}$ is defined to be $\max\{0, f_{i,j} - d_{i,j}\}$, where $d_{i,j}$ is the deadline of $\tau_{i,j}$. The tardiness of a task is defined to be the maximum tardiness of any of its released jobs. Since $d_{i,j} = r_{i,j} + P_i$, the response time bound of any task τ_i can be obtained by simply using the derived tardiness bound of the task plus a P_i value.

4 Analysis Overview

Our goal now is to derive a tardiness bound for each dataflow task τ_i scheduled under EDF. Since a job released by any dataflow task consists of a chain of sub-tasks, instead of directly bounding the tardiness of each task, our analysis technique seeks to (i) bound the tardiness of any first subtask τ_i^1 (Sec. 4.1), (ii) bound the tardiness of any second subtask τ_i^2 (Sec. 5), and (iii) finally bound the tardiness of any k^{th} ($k > 2$) subtask τ_i^k based on the tardiness bound derived for τ_i^{k-1} (Sec. 6). In the following, we first bound the tardiness of any τ_i^1 as it is quite straightforward, and then describe the challenges in bounding the tardiness for other subtasks.

4.1 Tardiness Bound on Type 1 Processors

According to our dataflow task model, every dataflow task τ_i releases its jobs periodically with a minimum job inter-arrival gap of P_i, and any subjob $\tau_{i,j+1}^1$ of job $\tau_{i,j+1}$ can start execution at time t if $r_{i,j+1} \leq t$ and subjob $\tau_{i,j}^1$ completes by t. Thus, the first subtask τ_i^1 of each task τ_i scheduled on type-1 processors can be viewed as a sporadic task[3] scheduled on a

[2] Note that, although in many hardware accelerators frequent job preemptions are allowed but discouraged due to overhead consideration, we allow preemptions in this paper as this is the first attempt resolving this scheduling problem. We leave the further issue of enforcing non-preemptive executions on certain types of processors as future work.

[3] Under the sporadic task model, each sporadic task continuously releases jobs and can be specified by (e_i, P_i), where e_i denotes the WCET of any released job and p_i specified the minimum amount of time that must elapse between the releases of successive jobs of this task.

homogeneous multiprocessor with M^1 processors: the period and WCET of each task τ_i^1 are P_i and e_i^1, respectively.

As reviewed earlier, Devi and Anderson [6] have shown that EDF scheduling is capable of scheduling sporadic task sets with probably bounded tardiness while incurring no capacity loss, and the tardiness bound can be calculated using the following closed-form expression:

$$Tardiness(\tau_i^1) = \frac{\sum_{\tau_i^1 \in \varepsilon_{max}(\tau^1, M^1-1)} e_i^1 - e_{min}^1}{M^1 - \sum_{\tau_i^1 \in \mathcal{U}_{max}(\tau^1, M^1-1)} u_i^1} + e_i^1, \tag{3}$$

where $\mathcal{U}_{max}(\tau^1, M^1-1)$ ($\varepsilon_{max}(\tau^1, M^1-1)$, respectively) denotes a subset of (M^1-1) tasks with the highest utilization (largest WCET, respectively) in τ^1 where τ^1 denotes the set of the first subtasks of all dataflow tasks. To ease our description, in the rest of this paper, we use TB_i^k to denote tardiness bound of subtask τ_i^k which is scheduled on type-k processors.

4.2 Challenges on Deriving Tardiness Bound for Any Subtask τ_i^k ($k > 1$)

It is straightforward to derive a tardiness bound for any subtask τ_i^1 because such subtasks can be naturally modeled as sporadic tasks, thus allowing existing analysis to be directly applied. However, since any subjob $\tau_{i,j}^k$ ($k > 1$) cannot start execution until $\tau_{i,j}^{k-1}$ completes, the releasing pattern of any such subtask τ_i^k is hard to characterize. Note that due to dependencies, subjob $\tau_{i,j}^k$ ($k > 1$) is said to be released when $\tau_{i,j}^{k+1}$ completes execution. That is, $r_{i,j}^k = f_{i,j}^{k+1}$, where $f_{i,j}^{k+1}$ denotes the completion time of subjob $\tau_{i,j}^{k+1}$. A straightforward approach is to force such subtasks to be modeled as sporadic tasks by calculating the minimum separation between any two consecutive subjob releases by τ_i^k. However, we use an example illustrating that such an approach would result in significant pessimism and could be invalid in many cases.

▶ **Example 2.** Fig. 2 shows an example to illustrate the pessimism induced by ordinary sporadic model. $\tau_{i,j}^k$ is released at 0 and completes execution at 5, when $\tau_{i,j}^{k+1}$ is released. $\tau_{i,j+1}^k$ is released at 6 and completes execution at 8, when $\tau_{i,j+1}^{k+1}$ is released. Thus, the time interval between the release time of these two consecutive subjobs of τ_i^{k+1} is 3 time units. The minimum time interval between any two consecutive subjob releases of τ_i^{k+1} is thus 3 time units. However, since $e_i^{k+1} = 4$ in this example, the utilization of τ_i^{k+1} would be forced to become $4/3 > 1$, which implies that τ_i^{k+1} is unschedulable and thus the whole task system is unschedulable.

As seen in the above example, intuitively, the approach of forcing any subtask τ_i^2 does not work due to dependencies. The minimum time separation between any two consecutive subjob releases by τ_i^2 could be too small due to the facts that any subjob $\tau_{i,j}^2$ is released at the time when $\tau_{i,j}^1$ completes and $\tau_{i,j}^1$ may complete later than its deadline due to tardiness. This may dramatically increase τ_i^2's utilization in the analysis and thus artificially create more workloads due to τ_i^2 than it actually contributes, which is too pessimistic and may even make the resulting schedulability test invalid. To resolve this challenge, we next present a set of novel analysis techniques to bound any subtask τ_i^2's tardiness by analyzing the actual workload contributed by each such subtask.

5 Tardiness Bound on Type 2 Processors

In this section, we derive the tardiness bound for any subtask τ_i^2 which is scheduled on type-2 processors. Our analysis techniques involve comparing the resource allocation to τ^2, which

denotes the set of all subtasks τ_i^2 in the system, in a processor sharing (PS) schedule (defined below) and an actual EDF schedule of interest for τ^2, both on M^2 type-2 processors, and quantifying the difference between the two. We analyze allocations on a per-subtask basis.

The time interval $[t_1, t_2)$, where $t_2 > t_1$, consists of all time instances t, where $t_1 \leq t < t_2$, and is of length $t_2 - t_1$. For any time $t > 0$, the notation τ^- denotes the time $t - \epsilon$ in the limit $\epsilon- > 0+$, and the notation t^+ is used to denote the time $t + \epsilon$ in the limit $\epsilon- > 0+$.

▶ **Definition 3** (active subjob)**.** A subtask τ_i^2 is active at time t in a schedule $\mathcal{S}$ if there exists a job $\tau_{i,j}^2$ (called τ_i^2's active job at t) such that $f_{i,j}^1 \leq t < d_{i,j}$ (as defined earlier, $f_{i,j}^1$ denotes the completion time of subjob $\tau_{i,j}^1$ as well as the time $\tau_{i,j}^2$ is released). By our task model, every subtask has at most one active subjob at any time.

▶ **Definition 4** (pending subjob)**.** A subjob $\tau_{i,j}^2$ is pending at time t in a schedule $\mathcal{S}$ if $f_{i,j}^1 \leq t$, and $\tau_{i,j}^2$ has not completed execution by t in $\mathcal{S}$. A subtask is pending at time t if any of its subjobs are pending at time t.

▶ **Definition 5** (ready subjob)**.** A pending subjob $T_{i,j}^2$ is ready at time t in a schedule $\mathcal{S}$ if $f_{i,j}^1 \leq t$, $T_{i,j}^2$ has not yet completed at t, and all its predecessor subjobs (i.e., $\tau_{i,j}^1$ and $\tau_{i,j-1}^2$) have completed execution by t in $\mathcal{S}$.

Let $A(\tau_{i,j}^2, t_1, t_2, \mathcal{S})$ denote the total processing time allocated to $\tau_{i,j}^2$ in an arbitrary schedule $\mathcal{S}$ in $[t_1, t_2)$. Then, the total time allocated to all jobs of τ_i^2 in S is given by

$$A(\tau_i^2, t_1, t_2, \mathcal{S}) = \sum_{j \geq 1} A(\tau_{i,j}^2, t_1, t_2, \mathcal{S}) \tag{4}$$

Now consider a PS schedule PS defined below.

▶ **Definition 6** (processor sharing)**.** PS is a processor-sharing (PS) schedule on M^2 type-2 processors for all subtasks in τ^2. In such a schedule, τ_i^2 executes with the rate u_i^2 when it is active.

Thus, if τ_i^2 is active throughout $[t_1, t_2)$, then

$$A(\tau_i^2, t_1, t_2, PS) = (t_2 - t_1) \cdot u_i^2. \tag{5}$$

The difference between the allocation to a subjob $\tau_{i,j}^2$ up to time t in a PS schedule and an arbitrary schedule S, denoted *the lag of subjob* $\tau_{i,j}^2$ *in schedule* S, is defined by

$$lag(\tau_{i,j}^2, t, S) = A(\tau_i^2, 0, t, PS) - A(\tau_i^2, 0, t, S). \tag{6}$$

The concept of lag is important because, if lags remain bounded, then tardiness is bounded as well. In order to bound lags for subjobs of subtasks $\tau_i^2 \in \tau^2$ on type-2 processors, PS plays a pivotal role. Intuitively, we can imagine that PS represent the schedule of a system of n type-2 processors, where the i^{th} processor has a capacity of $\frac{e_i^2}{P_i}$ and is dedicated to executing subjobs of τ_i^2. Thus, we can see that in a PS schedule subjobs released by different subtaskτ_i^2 will not interfere with each other and subjobs released by the same subtask execute at a constant rate sequentially.

The concept of PS is originally introduced in [6] for sporadic task systems. For the sporadic case, PS represents an "ideal" schedule since each job released at time t is expected to complete at its deadline, which is $t + P_i$. Thus, bounding the lag can be achieved through comparing the allocation to an arbitrary schedule S and the allocation in PS. However, a key difference between the sporadic task model and our dataflow task model is that each

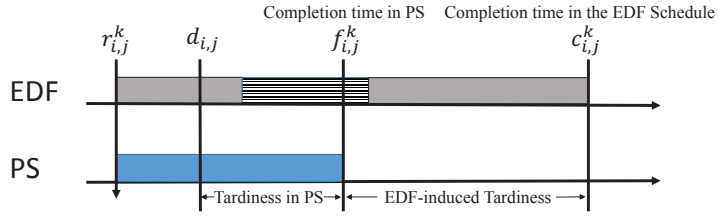


Figure 3 $\tau_{i,j}^k$'s tardiness is comprised of its tardiness in PS and the EDF-induced tardiness.

subjob has two types of dependencies. Due to dependencies, a subjob may not be able to ideally complete by its deadline in PS. Specifically, due to the facts that the release time $r_{i,j}^2$ of any subjob $\tau_{i,j}^2$ is the completion time $f_{i,j}^1$ of the subtask $\tau_{i,j}^1$ executed on type-1 processors ($f_{i,j}^1 > d_{i,j}$ is possible due to tardiness according to Eq. 3), and $\tau_{i,j}^1$ and $\tau_{i,j}^2$ have the same deadline $d_{i,j}$, $\tau_{i,j}^2$ thus may not complete by its deadline at $d_{i,j}$ in PS.

The above discussions highlight a key point that under the dataflow task model, the tardiness of τ_i^2 is due to two sources: (i) the tardiness due to using an arbitrary scheduler which can be less ideal than the PS scheduler (e.g., EDF in this paper), which can be bounded by comparing the allocation to τ_i^2 in the corresponding arbitrary schedule against the allocation in PS, and (ii) the tardiness of τ_i^2 seen in PS, which exists due to the potential late completion of subjobs of τ_i^1 (i.e., its subjobs complete after the corresponding deadlines) executed on type-1 processors (note again that this PS-induced tardiness does not exist for the ordinary sporadic task model.) This is also illustrated in Fig. 3.

Motivated by this key observation, we seek to bound the tardiness of τ_i^2 by bound two types of tardiness: PS-induced tardiness (Sec. 5.1) and EDF-induced tardiness (Sec. 5.2). Finally we obtain a tardiness bound for any τ_i^2 by combining these two types of tardiness (Sec. 5.3).

5.1 *PS*-induced Tardiness

In this section, we derive a bound on the PS-induced tardiness for τ_i^2; in particular, the following theorem shows that the tardiness of any subjob of τ_i^2 in PS can be bounded by a constant that depends only on the dataflow task τ_i's period and and the tardiness bound of τ_i^1 executed on type 1 processors.

▶ **Theorem 7.** *τ_i^2's tardiness in PS is bounded by $TB_i^1 + P_i$.*

Proof. For the first subjob $\tau_{i,1}^2$ of τ_i^2, its release time $r_{i,1}^2$ is no later than $r_{i,1}^1 + P_i + TB_i^1$. This is because the completion time of $\tau_{i,1}^1$ executed on type-1 processor is no later than $r_{i,1}^1 + P_i + TB_i^1$ according to Eq. 3. According to the definition of PS, we know that $\tau_{i,1}^2$ begins execution no later than $r_{i,1}^1 + P_i + TB_i^1$ and will complete no later than $r_{i,1}^1 + P_i + TB_i^1 + P_i$ in PS. Since the deadline of $\tau_{i,1}^2$ is $r_{i,1}^1 + P_i$, $\tau_{i,1}^2$'s tardiness is no greater than $TB_i^1 + P_i$ in PS.

We now prove this theorem's correctness for all the other jobs $\tau_{i,j}^2$ ($j > 1$) by contradiction. Let us assume that the tardiness of some job of τ_i^2 is larger than $TB_i^1 + P_i$ in PS. Let $\tau_{i,x}^2$ denote the first such job.

According to the definition of PS, all subjobs of τ_i^2 execute sequentially on type-2 processors at a constant rate $\frac{e_i^2}{P_i}$. Since the tardiness of $\tau_{i,x}^2$ in PS is greater than $TB_i^1 + P_i$, $\tau_{i,x}^2$ must begin execution after $r_{i,x}^1 + P_i + TB_i^1$. Note that $r_{i,x}^1$, which is one predecessor job of $\tau_{i,x}^2$, must complete by $r_{i,x}^1 + P_i + TB_i^1$ according to Eq. 3. Thus, this implies that $\tau_{i,x-1}^2$

does not complete at $r^1_{i,x} + P_i + TB^1_i$. Since the deadline of $\tau^2_{i,x-1}$ is $r^1_{i,x-1} + P_i$, the tardiness of $\tau^2_{i,x-1}$ in PS is greater than $(r^1_{i,x} + P_i + TB^1_i) - (r^1_{i,x-1} + P_i) = r^1_{i,x} + TB^1_i - r^1_{i,x-1}$. Since $r^1_{i,x} - r^1_{i,x-1} \geq P_i$, the tardiness of $\tau^2_{i,x-1}$ in PS is greater than $TB^1_i) + P_i$. This implies that $\tau^2_{i,x}$ is not the first subjob of τ^2_i that has a tardiness greater than $TB^1_i) + P_i$. A contradiction is reached and the theorem is thus proved. ◀

5.2 EDF-induced Tardiness

In this section, we bound EDF-induced tardiness for subtask τ^2_i by comparing any subjob $\tau^2_{i,j}$'s completion time under EDF and its completion time under PS. Since we only focus on subtasks and subjobs that are scheduled on type-2 processors in this section, to make the description easier, we call τ^2_i a task and $\tau^2_{i,j}$ a job in this section.

First we quantify the allocation to job $\tau^2_{i,j}$ in an interval $[t_1, t_2)$ in PS. For this, we have defined $A(S, \tau^2_{i,j}, t_1, t_2)$ to denote the total processing time allocated to $\tau^2_{i,j}$ in an arbitrary schedule $\mathcal{S}$ in $[t_1, t_2)$ in Eq. 4 and we have $A(\tau^2_{i,j}, t_1, t_2, PS) \leq (t_2 - t_1) \cdot u^2_i$ according to Eq. 5.

We relate the allocation to a job $\tau^2_{i,j}$ under PS to its allocation under EDF using the notation of lag, which is defined as

$$lag(\tau^2_{i,j}, t, EDF) = A(\tau^2_{i,j}, 0, t, PS) - A(\tau^2_{i,j}, 0, t, EDF). \tag{7}$$

Then, task lags can be defined in a similar way:

$$lag(\tau^2_i, t, EDF) = \sum_{j \geq 1} (A(\tau^2_{i,j}, 0, t, PS) - A(\tau^2_{i,j}, 0, t, EDF)). \tag{8}$$

The lag of task τ^2_i at t in EDF schedule indicates the difference between the allocation to τ^2_i in EDF and PS in interval $[0, t)$. If $lag(\tau^2_i, t, EDF)$ is positive, then schedule EDF has performed less work on the jobs of τ^2_i until t than PS, and more work if $lag(\tau^2_i, t, EDF)$ is negative.

The total lag for a finite job set Θ at t, denoted $LAG(\Theta, t, EDF)$, is given by the sum of the lags of all jobs in Θ. That is,

$$\begin{aligned} LAG(\Theta, t, EDF) &= \sum_{\tau^2_{i,j} \in \Theta} lag(\tau^2_{i,j}, t, EDF) \\ &= \sum_{\tau^2_{i,j} \in \Theta} (A(\tau^2_{i,j}, 0, t, PS) - A(\tau^2_{i,j}, 0, t, EDF)). \end{aligned} \tag{9}$$

Since both $A(\tau^2_{i,j}, 0, 0, PS)$ and $A(\tau^2_{i,j}, 0, 0, EDF)$ are zero, $\forall\, i$ and $\forall\, j$, $LAG(\Theta, 0, EDF)$ is zero. By Eq. 7 and Eq. 9, we have the following for $t_2 > t_1$:

$$\begin{aligned} LAG(\Theta, t_2, EDF) = LAG(\Theta, t_1, EDF) \\ + A(\Theta, t_1, t_2, PS) - A(\Theta, t_1, t_2, EDF). \end{aligned} \tag{10}$$

▶ **Definition 8** (busy/none-busy interval)**.** A time interval $[t_1, t_2)$ is busy for a job set Θ, if, at each time-instant $t \in [t_1, t_2)$, all processors executes jobs from Θ, and is non-busy for Θ otherwise. An interval $[t_1, t_2)$ is maximally non-busy for Θ if it is non-busy for Θ at every time instant within it and either $t_1 = 0$ or t_1^- is a busy instant for Θ.

If $[t_1, t_2)$ is a busy interval in an EDF schedule for Θ, then the tasks in Θ receive a total allocation of $M^2(t_2 - t_1)$ time in that interval in an EDF schedule. By Eq. 8, the total allocation to Θ cannot exceed $M^2(t_2 - t_1)$ in PS. Thus, by Eq. 9, the LAG of Θ at t_2 is no larger than that at t_1, and the following lemma holds.

▶ **Lemma 9.** *For any time interval* $[t_1, t_2)$ *that is busy for* Θ, $LAG(\Theta, t_2, EDF) \leq LAG(\Theta, t_1, EDF)$.

Lemma 9 implies that the total lag for a job set can only increase across non-busy intervals, which causes tardiness of jobs. Next, we bound EDF-induced tardiness on type 2 processors using lags. For the rest of this section, let τ^2 denote the task system scheduled on type 2 processors. τ^2 has a PS schedule and satisfies Eq. 1 and Eq. 2. All jobs $\tau^2_{i,j} \in \tau^2$ have the following information:

- release time $r^2_{i,j}$
- execution time e^2_i
- completion time $f^2_{i,j}$ for $\tau^2_{i,j}$ in PS

▶ **Definition 10.** $\hat{f}^2_{i,j} = \max\{d_{i,j}, f^2_{i,j}\}$ denotes the earliest time at or after $d_{i,j}$ by which $\tau^2_{i,j}$ has completed in PS.
Note that $\hat{f}^2_{i,j} \geq f^2_{i,j}$ and job $\tau^2_{i,j}$'s tardiness in PS equals $\hat{f}^2_{i,j} - d_{i,j}$.

▶ **Definition 11.** Let $c^2_{i,j}$ denotes the completion time of $\tau^2_{i,j}$ in the EDF schedule. **EDF-induced tardiness** equals $\max\{c^2_{i,j} - \hat{f}^2_{i,j}, 0\}$, as illustrated earlier in Fig. 3.

In the rest of this section, we bound EDF-induced tardiness by leveraging and extending the general analysis framework first developed in [6]. Let

$$\rho = \max_{\tau^1_i \in \tau^1} \{TB^1_i\}. \tag{11}$$

ρ denotes the largest tardiness bound among all subtasks τ^1_i that ae scheduled on type-1 processors. We assume that the EDF schedule has the following property.

▶ **Property 12.** *The EDF-induced tardiness of every job of every task* τ^2_k *in* τ^2 *with deadline less than* $d_{i,j}$ *is at most* $x + e^2_k$, *where* $x \geq \rho$.

Our goal is to determine the smallest $x \geq \rho$ such that the tardiness of $\tau^2_{i,j}$ remains at most $x + e^2_i$. Such a result would by induction imply a tardiness of at most $x + e^2_k$ for all jobs of every task $\tau^2_k \in \tau^2$. Because τ^2 is arbitrary, the tardiness bound will hold for every concrete instantiation of τ^2. The objective is easily met if $\tau^2_{i,j}$ completes by its deadline, $d_{i,j}$, so assume otherwise. The completion time of $\tau^2_{i,j}$ then depends on the amount of work that can compete with $\tau^2_{i,j}$ after $d_{i,j}$. Hence, a value for x can be determined via the following steps.

1. Compute an upper bound on pending work for tasks in τ^2 (including $\tau^2_{i,j}$) that can compete with $\tau^2_{i,j}$ after $d_{i,j}$.
2. Determine the amount of such work necessary for the tardiness of $\tau^2_{i,j}$ to exceed $x + e^2_i$.
3. Determine the smallest $x \geq \rho$ such that the tardiness of $\tau^2_{i,j}$ is at most $x + e^2_i$ using the upper bound in Step 1 and the necessary condition in Step 2.

To reason about the tardiness of $\tau^2_{i,j}$ we need to determine how other jobs preempt/delay its execution. We classify such jobs based on the relation between their deadlines and completion time on PS and those of $\tau^2_{i,j}$, as follows.

- $\mathbf{fd} = \{\tau^2_{l,k} : d_{l,k} \leq d_{i,j} \wedge f^2_{l,k} \leq \hat{f}^2_{i,j}\}$
- $\mathbf{fD} = \{\tau^2_{l,k} : d_{l,k} > d_{i,j} \wedge f^2_{l,k} \leq \hat{f}^2_{i,j}\}$
- $\mathbf{Fd} = \{\tau^2_{l,k} : d_{l,k} \leq d_{i,j} \wedge f^2_{l,k} > \hat{f}^2_{i,j}\}$
- $\mathbf{FD} = \{\tau^2_{l,k} : d_{l,k} > d_{i,j} \wedge f^2_{l,k} > \hat{f}^2_{i,j}\}$

In this notation, f and F denote jobs' completion time in PS at most and greater than $\hat{f}_{i,j}$, respectively. d denotes that $\tau^2_{l,k}$'s deadline is no later than that of $\tau^2_{i,j}$, and D denotes that $\tau^2_{l,k}$'s deadline is later than that of $\tau^2_{i,j}$. Note that $\tau^2_{i,j} \in fd$.

The set of jobs with completion time in PS at most $\hat{f}^2_{i,j}$ is further referred to as $\Theta = fd \cup fD$. This set of jobs do not execute beyond $\hat{f}^2_{i,j}$ in the PS schedule. Note that because the jobs in $fd \cup Fd$ might have the priority at least that of $\tau^2_{i,j}$ (at some time instant), the execution of $\tau^2_{i,j}$ might be postponed (in the worst case) until there are at most m ready jobs in $fd \cup Fd$ including $\tau^2_{i,j}$. Based on this observation, we derive the EDF-induced tardiness bound in three steps.

5.2.1 Step 1: An upper bound on competing work

In this section, we determine an upper bound on competing work for $\tau^2_{i,j}$, which is denoted by $UB(fd \cup Fd, \hat{f}_{i,j}, EDF)$.

Because jobs in $fd \cup Fd$ can have priorities at least that of $\tau^2_{i,j}$, the competing work due to $fd \cup Fd$ for $\tau^2_{i,j}$ beyond $\hat{f}^2_{i,j}$, $UB(fd \cup Fd, \hat{f}_{i,j}, EDF)$, is bounded by the sum of (i) the amount of work pending at $\hat{f}^2_{i,j}$ for jobs in fd, and (ii) the amount of work $W(Fd, \hat{f}^2_{i,j}, EDF)$ required by jobs in Fd that can compete with $\tau^2_{i,j}$ after $\hat{f}^2_{i,j}$. For the pending work mentioned in (i), because jobs from Θ have completion time in PS at most $\hat{f}^2_{i,j}$, they do not execute in the PS schedule beyond $\hat{f}^2_{i,j}$. Thus, the work pending for jobs in fd is given by $LAG(fd, \hat{f}^2_{i,j}, EDF)$, which must be positive in order for $\tau^2_{i,j}$ to exceed its completion time in PS at $\hat{f}_{i,j}$.

Instead of bounding $LAG(fd, \hat{f}^2_{i,j}, EDF)$, we try to bound $LAG(\Theta, \hat{f}^2_{i,j}, EDF)$. This is because $LAG(fd, \hat{f}^2_{i,j}, EDF) \leq LAG(\Theta, \hat{f}^2_{i,j}, EDF)$. This inequality holds because $\Theta = fd \cup fD$, and $LAG(fD, \hat{f}^2_{i,j}, EDF)$ is non-negative because according to the definition of fD, the jobs in fD cannot perform more work by time $\hat{f}^2_{i,j}$ in EDF than they have performed in the PS schedule. Thus, $W(fd \cup Fd, \hat{f}^2_{i,j}, EDF) \leq LAG(\Theta, \hat{f}^2_{i,j}, EDF) + W(Fd, \hat{f}^2_{i,j}, EDF)$. Therefore, $W(fd \cup Fd, \hat{f}^2_{i,j}, EDF)$ can be obtained by determining upper bounds for $LAG(\Theta, \hat{f}^2_{i,j}, EDF)$ and $W(Fd, \hat{f}^2_{i,j}, EDF)$ individually.

Upper bound on $LAG(\Theta, \hat{f}^2_{i,j}, EDF)$. Since we are deriving this bound w.r.t. Θ, we assume that all busy and non-busy intervals considered are w.r.t. Θ and the EDF schedule.

By Lemma 9, if no non-busy interval for Θ exists in $[0, \hat{f}^2_{i,j})$, then $LAG(\Theta, \hat{f}^2_{i,j}, EDF) \leq LAG(\Theta, 0, EDF) = 0$. Thus, we consider the more interesting case where some non-busy interval exists in $[0, \hat{f}^2_{i,j})$. An interval for jobs in Θ could be non-busy for two reasons:

- Simply not enough ready jobs in Θ exists that can occupy all available processors. Thus, it does not matter whether jobs from Fd or FD execute during the interval. Such an interval is called non-busy non-occupation.
- There exists ready jobs in Θ that cannot execute within some sub-intervals in $[0, \hat{f}^2_{i,j})$, since jobs in Fd occupy one or more processors and they have higher priorities. Such an interval is called non-busy occupation.

Let the carry-in job $\tau^2_{c,h}$ of a task τ^2_c be defined as the job, if any, for which $r^2_{c,h} \leq \hat{f}^2_{i,j} < f^2_{c,h}$ holds. Note that at most one such job could exist for each task τ^2_c and only such carry-in jobs can prevent the execution of jobs in Θ before time $\hat{f}^2_{i,j}$, thus increasing the LAG for Θ.

▶ **Definition 13.** Let τ^2_H be the set of tasks that have carry in jobs in Fd.

▶ **Definition 14.** Let δ_c be the amount of work performed by a carry-in job $\tau^2_{c,h}$ in EDF by time $\hat{f}^2_{i,j}$.

As discussed earlier, LAG for Θ can increase only across non-busy intervals. In much of the rest of the analysis, we focus on a time t_2 defined as follows: if there exists a non-busy non-occupation interval before $\hat{f}^2_{i,j}$, across which LAG for Θ increases, then let $[t_1, t_2]$ denote the latest such interval; otherwise, $t_1 = t_2 = 0$.

The following two lemmas have been proved previously in [6, 18] for ordinary sporadic task systems (note again that no PS-induced tardiness exists for sporadic task systems). Note that when we calculate EDF-induced tardiness, the value of $LAG(\Theta, \hat{f}^2_{i,j}, EDF) + W(FD, \hat{f}^2_{i,j}, EDF)$ depends only on allocations in the PS schedule and allocations to jobs in $\Theta \cup FD$ by time $\hat{f}^2_{i,j}$ in the EDF schedule, which can compete processing time with $\tau^2_{i,j}$ after $\hat{f}^2_{i,j}$. Thus, the tardiness in PS does not affect the derivation of the EDF-induced tardiness. Also, Property 12 alone is sufficient for determining how much work any job in $\Theta \cup FD$ other than $\tau^2_{i,j}$ completes before $\hat{f}^2_{i,j}$. For these reasons, Lemmas 15 and Lemma 16 continue to hold for τ^2 task systems.

▶ **Lemma 15.** $LAG(\Theta, \hat{f}^2_{i,j}, EDF) \leq LAG(\Theta, t_2, EDF) + \sum_{\tau^2_c \in \tau^2_H} \delta_c(1 - u^2_c)$.

Proof. By Eq. (9) and Eq. (10),

$$\begin{aligned} &LAG(\Theta, \hat{f}_{l,j}, EDF) \\ &\leq LAG(\Theta, t_2, EDF) + A(\Theta, t_2, \hat{f}_{l,j}, PS) \\ &- A(\Theta, t_2, \hat{f}_{l,j}, EDF). \end{aligned} \tag{12}$$

We split $[t_2, \hat{f}_{i,j}]$ into b non-overlaping intervals $[t_{p_i}, t_{q_i})$, where $1 \leq i \leq b$, such that $t_2 = t_{p_1}$, $t_{q_{(i-1)}} = t_{p_i}$ and $t_{q_b} = \hat{f}_{i,j}$. Each interval $[t_{p_i}, t_{q_i})$ is either busy, non-busy occupation or non-busy non-occupation. We assume that any occupation interval $[t_{p_i}, t_{q_i})$ is defined so that if a task $\tau^2_c \in \tau^2_H$ executes at some point in the interval, then it executes continuously throughout the interval. Note that such a task τ^2_c does not necessarily execute continuously throughout $[t_2, \hat{f}_{i,j})$. For each occupation interval $[t_{p_i}, t_{q_i})$, we define a subset of tasks $\alpha^2_c \subseteq \tau^2_H$ that execute continuously throughout $[t_{p_i}, t_{q_i})$. The allocation difference for Θ throughout the interval $[t_2, \hat{f}^2_{i,j})$ is thus:

$$\begin{aligned} &A(\Theta, t_2, \hat{f}^2_{l,j}, PS) - A(\Theta, t_2, \hat{f}^2_{l,j}, EDF) \\ &= \sum_{i=1}^{b} (A(\Theta, t_{p_i}, t_{q_i}, PS) - A(\Theta, t_{p_i}, t_{q_i}, EDF)). \end{aligned} \tag{13}$$

We now bound the difference between the work performed in the PS schedule and the schedule EDF across each of these intervals $[t_{p_i}, t_{q_i})$. The sum of these bounds gives us a bound on the total allocation difference throughout $[t_2, \hat{f}^2_{i,j})$. Depending on the nature of the interval $[t_{p_i}, t_{q_i})$, three cases are possible.

Case 1. $[t_{p_i}, t_{q_i})$ **is busy.** Because in EDF all processors are occupied by jobs in Θ, $A(\Theta, t_{p_i}, t_{q_i}, EDF) = M^2 \times (t_{q_1} - t_{p_i})$. In PS, $A(\Theta, t_{p_i}, t_{q_i}, PS) \leq U^2_{sum}(t_{q_1} - t_{p_i})$. Since $U^2_{sum} \leq M^2$, we have

$$A(\Theta, t_{p_i}, t_{q_i}, PS) - A(\Theta, t_{p_i}, t_{q_i}, EDF) \leq 0. \tag{14}$$

Case 2. $[t_{p_i}, t_{q_i})$ **is non-busy non-occupation.** By the selection of $[t_1, t_2]$, LAG does not increase for Θ across $[t_{p_i}, t_{q_i})$. Therefore, from Eq. (10), we have

$$A(\Theta, t_{p_i}, t_{q_i}, PS) - A(\Theta, t_{p_i}, t_{q_i}, EDF) \leq 0. \tag{15}$$

Case 3. $[t_{p_i}, t_{q_i})$ **is non-busy occupation.** The cumulative utilization of all tasks $\tau^2_c \in \alpha^2_c$, which execute continuously throughout $[t_{p_i}, t_{q_i})$, is $\sum_{\tau^2_c \in \alpha^2_c} u_c$. The carry-in jobs of these tasks do not belong to Θ, by the definition of Θ. Therefore, the allocation of jobs

in Θ during $[t_{p_i}, t_{q_i})$ in PS is at most $A(\Theta, t_{p_i}, t_{q_i}, PS) = (t_{q_i} - t_{p_i})(M^2 - \sum_{\tau_c^2 \in \alpha_c^2} u_c)$. All processors are occupied at every time instant in the interval $[t_{p_i}, t_{q_i})$, because it is occupation. Thus, $A(\Theta, t_{p_i}, t_{q_i}, EDF) = (t_{q_i} - t_{p_i})(M^2 - |\alpha_c^2|)$. Therefore, the allocation difference for jobs in Θ throughout the interval is

$$\begin{aligned} & A(\Theta, t_{p_i}, t_{q_i}, PS) - A(\Theta, t_{p_i}, t_{q_i}, EDF) \\ & \leq (t_{q_i} - t_{p_i})((M^2 - \sum_{\tau_c^2 \in \alpha_c^2} u_c^2) - (M^2 - \mid \alpha_c^2 \mid)) \\ & = (t_{q_i} - t_{p_i}) \sum_{\tau_c^2 \in \alpha_c^2} (1 - u_c^2)). \end{aligned} \tag{16}$$

To complete our proof, define $\alpha_c^2 =$ null for all intervals $[t_{p_i}, t_{q_i})$ that are either busy or non-busy non-occupation. Then, summing the allocation differences for all the intervals $[t_{p_i}, t_{q_i})$ given by Eq. (14), Eq. (15), and Eq. (16), we have

$$\begin{aligned} & A(\Theta, t_{p_i}, t_{q_i}, PS) - A(\Theta, t_{p_i}, t_{q_i}, EDF) \\ & \leq \sum_{i=1}^{b} \sum_{\tau_c^2 \in \alpha_c^2} (t_{q_i} - t_{p_i})(1 - u_c^2)). \end{aligned} \tag{17}$$

For each task $\tau_c^2 \in \tau_H^2$, the sum of the lengths of the intervals $[t_{p_i}, t_{q_i})$, in which the carry-in job of τ_c^2 executes continuously is at most δ_c^2. $A(\Theta, t_{p_i}, t_{q_i}, PS) - A(\Theta, t_{p_i}, t_{q_i}, EDF) \leq \sum_{\tau_c^2 \in \tau_H^2} \delta_c^2(1-u_c^2)$. Setting this value into Eq. (12), we get $LAG(\Theta, \hat{f}_{i,j}, EDF) \leq LAG(\Theta, t_2, EDF) + \sum_{\tau_c^2 \in \tau_H^2} \delta_c^2(1 - u_c^2)$. ◀

▶ **Lemma 16.** *$lag(\tau_k^2, t, EDF) \leq x \times u_k^2 + e_k^2$ for any task τ_k^2 and $t \in [0, \hat{f}_{i,j}^2]$.*

Proof. Let $\hat{f}_{k,j}^2$ be the completion time of the earliest pending job of τ_k^2, $\tau_{k,j}^2$, in the PS schedule at time t. Let γ_k be the amount of work $\tau_{k,j}^2$ performs before t in the EDF schedule. We first prove the lemma for the case $\hat{f}_{k,j}^2 < t$. By Eq. (7) and the selection of $\tau_{k,j}^2$,

$$\begin{aligned} & lag(\tau_k^2, t, EDF) \\ & = \sum_{h \geq j} lag(\tau_{k,h}^2, t, EDF) \\ & = \sum_{h \geq j} (A(\tau_{k,h}^2, 0, t, PS) - A(\tau_{k,h}^2, 0, t, EDF)) \end{aligned} \tag{18}$$

$A(\tau_{k,h}^2, 0, t, EDF) = A(\tau_{k,h}^2, r_{k,h}^2, t, EDF)$, because no job executes before its release time. Thus,

$$\begin{aligned} & lag(\tau_k^2, t, EDF) \\ & = \sum_{h > j} (A(\tau_{k,h}^2, r_{k,h}^2, t, PS) - A(\tau_{k,h}^2, r_{k,h}^2, t, EDF)) \\ & + A(\tau_{k,j}^2, r_{k,j}^2, t, PS) - A(\tau_{k,j}^2, r_{k,j}^2, t, EDF). \end{aligned} \tag{19}$$

$A(\tau_{k,j}^2, r_{k,j}^2, t, PS) = e_k^2$ and $\sum_{h>j} A(\tau_{k,h}^2, r_{k,h}^2, t, PS) \leq u_k^2(t - \hat{f}_{k,j}^2)$, by the definition of PS. $A(\tau_{k,j}^2, r_{k,j}^2, t, EDF) = \gamma_k$ and $\sum_{h>j} A(\tau_{k,h}^2, r_{k,h}^2, t, EDF) = 0$, by the selection of $\tau_{k,j}^2$. Setting these values into Eq. (19), we have

$$lag(\tau_k^2, t, EDF) \leq u_k^2(t - \hat{f}_{k,j}^2) + e_k^2 - \gamma_k \tag{20}$$

By Property (P), $\tau^2_{k,j}$ has EDF-induced tardiness at most $x+e^2_k$, so $t+e^2_k-\gamma_k \le \hat{f}^2_{k,j}+x+e^2_k$. Thus, $t-\hat{f}^2_{k,j} \le x+e^2_k+\gamma_k-e^2_k$. From 20, we have

$$lag(\tau^2_k,t,EDF) \le u^2_k(t-\hat{f}^2_{k,j})+e^2_k-\gamma_k = u^2_k(x+e^2_k+\gamma_k-e^2_k)+e^2_k-\gamma_k$$
$$\le x\times u^2_k+e^2_k.$$

Then, we prove the lemma for the case $\hat{f}^2_{k,j} \ge t$. By Eq. (7) and the selection of $\tau^2_{k,j}$,

$$lag(\tau^2_k,t,EDF) = \sum_{h\le j} lag(\tau^2_{k,h},t,EDF) = \sum_{h\le j}(A(\tau^2_{k,h},0,t,PS)-A(\tau^2_{k,h},0,t,EDF))$$
$$= \sum_{h<j}(A(\tau^2_{k,h},r^2_{k,h},t,PS)-A(\tau^2_{k,h},r^2_{k,h},t,EDF))$$
$$+A(\tau^2_{k,j},r^2_{k,j},t,PS)-A(\tau^2_{k,j},r^2_{k,j},t,EDF). \quad (21)$$

By the definition of PS, $\sum_{h<j} A(\tau^2_{k,h},r^2_{k,h},t,PS) \le \sum_{h<j} e^2_k$; since $\tau_{k,j}$ is the earliest pending job of τ^2_k in the schedule EDF at time t, $\sum_{h<j} A(\tau^2_{k,h},r_{k,h},t,EDF) = \sum_{h<j} e^2_k$. Also, $A(\tau^2_{k,j},r^2_{k,j},t,PS) \le e^2_k$ and $A(\tau^2_{k,j},r^2_{k,j},t,EDF) = \gamma_k \ge 0$, and setting these values into Eq. (21):

$$lag(\tau^2_k,t,EDF) \le (\sum_{h<j} e^2_k - \sum_{h<j} e^2_k)+e^2_k-\gamma_k \le e^2_k$$

Therefore, $lag(\tau^2_k,t,EDF) \le x\times u^2_k+e^2_k$ for any task τ^2_k and $t\in[0,\hat{f}_{i,j}]$ is proved. ◀

We now prove that there is an upper bound on Θ's LAG at time t_2. Let $k'=\max\{k : \tau_{i,k}\in\Theta, r_{i,k}\le\hat{f}_{l,j}\}$. Define

▶ **Definition 17.** Let $U(\tau^2,y)$ ($E(\tau^2,y)$) be the set of at most $\min\{|\tau^2|,y\}$ tasks from τ^2 of highest utilization (execution cost), where $|\tau^2|$ is the number of tasks in τ^2, and let

$$E^2_L = \sum_{\tau^2_i\in E(\tau^2,M^2-1)} e^2_i \quad (22)$$

and

$$U^2_L = \sum_{\tau^2_i\in U(\tau^2,M^2-1)} u^2_i \quad (23)$$

▶ **Lemma 18.** $LAG(\Theta,t_2,EDF) \le x\times U^2_L+E^2_L$.

Proof. To bound $LAG(\Theta,t_2,EDF)$, we sum individual task lags at t_2. If $t_2=0$, then $LAG(\Theta,t_2,EDF)=0$ and the lemma holds trivially. So assume that $t_2>0$. Consider the set of tasks $\chi=\{\tau^2_i : \exists\tau^2_{i,j}\in\Theta$ such that $\tau^2_{i,j}$ is pending at $t_2\}$. Because the instant t_2 is non-busy non-occupation, $|\chi|\le(M^2-1)$. If a task has no pending jobs at t_2, then $lag(\tau^2_i,t_2,S)\le 0$. Therefore, by Eq. (7) and Lemma 16, we have

$$LAG(\Theta,t_2,EDF)$$
$$= \sum_{\tau^2_i:\tau^2_{i,j}\in\Theta} lag(\tau^2_i,t_2,EDF) \le \sum_{\tau^2_i\in\chi} lag(\tau^2_i,t_2,EDF)$$
$$\le \sum_{\tau^2_i\in\chi}(u^2_i\times x+e^2_i) \le E^2_L+x\times U^2_L \quad (24)$$

◀

Thus, based on Lemma 15 and Lemma 18, we have the desired upper bound $LAG(\Theta,\hat{f}_{i,j},S) \le x\times U^2_L+E^2_L+\sum_{\tau^2_c\in\tau^2_H}\delta_c(1-u^2_c)$.

Upper bound on $W(Fd, \hat{f}^2_{i,j}, EDF)$. To compute a bound on the requirement of jobs that can compete with $\tau^2_{i,j}$ after $\hat{f}^2_{i,j}$, $W(Fd, \hat{f}^2_{i,j}, EDF)$, we first find the latest release time of such a job. Intuitively, if a job is released at a time instant which is far behind $\hat{f}^2_{i,j}$, the job's priority may not be higher than $\tau^2_{l,j}$'s priority due to the constrained range of its tardiness bound on the type 1 processors.

▶ **Lemma 19.** *If $\tau^2_{l,k} \in Fd \cup fd$, then $r^2_{l,k} \leq \hat{f}^2_{i,j} + TB^1_l$.*

Proof. Given that $\hat{f}^2_{i,j} \geq d_{i,j}$ holds for any job $\tau^2_{i,j}$, if $\tau^2_{l,k} \in Fd \cup fd$, then $\hat{f}^2_{i,j} \geq d_{i,j} \geq d_{l,k}$ holds. According to Eq. (3), we have $r^2_{l,k} \leq d_{l,k} + TB^1_l$. Thus, $r^2_{l,k} \leq \hat{f}^2_{i,j} + TB^1_l$ is proved. ◀

▶ **Corollary 20.** *All jobs in $Fd \cup fd$ are released at or before $\hat{f}^2_{i,j} + \rho$, where $\rho = \max_{\tau^1_i \in \tau^1}\{TB^1_i\}$ according to Eq. 11.*

According to the dataflow task model, a subjob's release time on type 2 processors is the completion time of the corresponding subjob released by the same dataflow job on type 1 processors. Then, the minimum inter-arrival time of any two consecutive subjobs released by the same dataflow task on type 2 processors is the dataflow task's execution time on type 1 processors. Thus, we have the following corollary.

▶ **Corollary 21.** *The minimum inter-arrival time of any two consecutive jobs, i.e. $\tau^2_{l,k}$ and $\tau^2_{l,k+1}$, released by the same task τ^2_l on type 2 processors equals to the corresponding dataflow task's execution time on type 1 processors, i.e. e^1_l.*

▶ **Lemma 22.** *The amount of work $W(Fd, \hat{f}^2_{i,j}, EDF)$ required by jobs in Fd that can compete with $\tau^2_{i,j}$ after $\hat{f}^2_{i,j}$ can be bounded by $\sum_{\tau^2_c \in \tau^2_H}(e^2_c - \delta_c) + \sum_{\tau^2_l \in \tau^2 \setminus \tau^2_i}(\lceil \frac{TB^1_l}{e^1_l} \rceil) e^2_l$.*

Proof. Each job $\tau_{l,k}$ in Fd is either a carry-in job or is released after $\hat{f}_{i,j}$. In the latter case, by Lemma 19, $\tau_{l,k}$ is released in the interval $(\hat{f}^2_{i,j}, \hat{f}^2_{i,j} + TB^1_l]$. Thus, each task τ^2_l may have one carry-in job in Fd and up to $\lceil \frac{TB^1_l}{e^1_l} \rceil$ jobs in Fd released after $\hat{f}^2_{i,j}$ according to corollary. 20 and 21. If τ^2_l has a carry-in job, then τ^2_l is in τ^2_H and the work due to its carry-in job after $\hat{f}^2_{i,j}$ is at most $e^2_l - \delta_i$. The work generated by any job of τ^2_l in Fd released after $\hat{f}^2_{i,j}$ is at most e^2_l. From these facts, the lemma follows. (Note that τ^2_i is excluded from the second summation because it does not have jobs in Fd.) ◀

Upper bound on $W(fd \cup Fd, \hat{f}^2_{i,j}, EDF)$. Since $W(fd \cup Fd, \hat{f}^2_{i,j}, EDF) \leq LAG(\Theta, \hat{f}^2_{i,j}, EDF) + W(Fd, \hat{f}^2_{i,j}, EDF)$, by Lemmas 15, 18, and 22, we have

$$\begin{aligned} W(fd \cup Fd, \hat{f}^2_{i,j}, EDF) &\leq x \times U^2_L + E^2_L + \sum_{\tau^2_c \in \tau^2_H}(\delta_c(1 - u^2_c) + (e^2_c - \delta_c)) \\ &\quad + \sum_{\tau^2_l \in \tau^2 \setminus \tau^2_i}(\lceil \frac{TB^1_l}{e^1_l} \rceil) e^2_l \\ &\leq x \times U^2_L + E^2_L + \sum_{\tau^2_l \in \tau^2 \setminus \tau^2_i}(\lceil \frac{TB^1_l}{e^1_l} \rceil + 1) e^2_l. \end{aligned} \tag{25}$$

5.2.2 Step 2: Determining necessary condition for tardiness to exceed $x + e^2_i$

We now find a lower bound on the amount of competing work that is necessary for $\tau^2_{i,j}$ to miss its deadline by more than $x + e^2_i$.

▶ **Lemma 23.** *If the tardiness of $\tau_{i,j}^2$ exceeds $x + e_i^2$, where $x \geq \rho$ (recall that ρ is defined in Eq. 11), then $W(fd \cup Fd, \hat{f}_{i,j}^2, EDF) > \rho + m \times (x - \rho) + e_i^2$.*

Proof. We prove it by contraposition: we assume that $W(fd \cup Fd, \hat{f}_{i,j}^2, EDF) \leq \rho + m \times (x - \rho) + e_i^2$ holds and show that the tardiness of $\tau_{i,j}^2$ can not exceed $x + e_i^2$. All jobs in $fD \cup FD$ are ignored for this proof, because they can not preempt $\tau_{i,j}^2$ and thus they can not delay $\tau_{i,j}^2$'s completion time. According to Corollary 20, all jobs in $fd \cup Fd$ are released at or before $\hat{f}_{i,j}^2 + \rho$. Thus, the number of tasks with pending jobs in $fd \cup Fd$ definitely decreases after $\hat{f}_{i,j}^2 + \rho$.

Consider the point in time $b_{i,j} = \max\{c_{i,j-1}^2, v_{i,j}^2\}$, where $v_{i,j}^2 = \min\{t \geq \hat{f}_{i,j}^2 : [t, \infty)$ is a non-busy interval$\}$ and $c_{i,j-1}^2$ is the completion time of $\tau_{i,j-1}^2$ in EDF schedule.

At $b_{i,j}$, $\tau_{i,j}^2$ must have begun executing in EDF, because $b_{i,j} \geq v_{i,j}^2 \geq \hat{f}_{i,j}^2 \geq r_{i,j}^2$. Its predecessor has completed (since $b_{i,j} \geq c_{i,j-1}^2$), and there is at least one idle processor after $b_{i,j}$. Therefore, $\tau_{i,j}^2$ will complete by $b_{i,j} + e_i^2$. In other words,

$$c_{i,j}^2 \leq \max\{c_{i,j-1}^2 + e_i^2, v_{i,j}^2 + e_i^2\} \leq \max\{\hat{f}_{i,j-1}^2 + x + e_i^2 + e_i^2, v_{i,j}^2 + e_i^2\} \tag{26}$$

We consider two cases depending on the relationship between $\hat{f}_{i,j-1}^2 + x + e_i^2 + e_i^2$ and $v_{i,j}^2 + e_i^2$.

Case 1. $\boldsymbol{\hat{f}_{i,j-1}^2 + x + e_i^2 + e_i^2 \geq v_{i,j}^2 + e_i^2}$**.** Because jobs of τ_i^2 execute sequentially at a rate of u_i^2 in PS, and do not begin until their predecessors complete, thus,

$$f_{i,j}^2 \geq f_{i,j-1}^2 + e_i^2/u_i^2 \geq f_{i,j-1}^2 + e_i^2. \tag{27}$$

Also, jobs execute in PS after their release times, thus,

$$f_{i,j}^2 \geq r_{i,j}^2 + e_i^2/u_i^2 \geq r_{i,j}^2 + e_i^2 \geq d_{i,j-1}^2 + e_i^2. \tag{28}$$

Combine Eq. (27) with Eq. (28), we get $f_{i,j}^2 \geq \max\{f_{i,j-1}, d_{i,j-1}\} + e_i^2 = \hat{f}_{i,j-1}^2 + e_i^2$. Thus, Eq. (26) becomes

$$\begin{aligned} c_{i,j}^2 &\leq \max\{\hat{f}_{i,j-1} + x + e_i^2 + e_i^2, v_{i,j}^2 + e_i^2\} = \hat{f}_{i,j-1} + x + e_i^2 + e_i^2 \leq f_{i,j} + x + e_i^2 \\ &\leq \hat{f}_{i,j}^2 + x + e_i^2, \end{aligned} \tag{29}$$

which implies that the tardiness of $\tau_{i,j}^2$ does not exceed $x + e_i^2$.

Case 2. $\boldsymbol{\hat{f}_{i,j-1}^2 + x + e_i^2 + e_i^2 < v_{i,j}^2 + e_i^2}$**.** If $v_{i,j}^2 \leq \hat{f}_{i,j}^2 + x$, then by the definition of $v_{i,j}^2$, the tardiness of $\tau_{i,j}^2$ can not exceed $x + e_i^2$. Thus, we assume otherwise, i.e., $v_{i,j}^2 > \hat{f}_{i,j}^2 + x$. In this case, $[\hat{f}_{l,j} + \rho, \hat{f}_{l,j} + x]$ must be a busy interval. This is because according to Corollary 20, all jobs in $fd \cup Fd$ are released at or before $\hat{f}_{i,j}^2 + \rho$. Thus, the number of tasks with pending jobs in $fd \cup Fd$ definitely decreases after $\hat{f}_{i,j}^2 + \rho$. When $v_{i,j}^2 > \hat{f}_{i,j}^2 + x$, $v_{i,j}^2$ must be the earliest non-busy time instance after $\hat{f}_{i,j}^2 + \rho$. The reason is explained in Fig. 4. Since no jobs are released after $\hat{f}_{i,j}^2 + \rho$, once a processor is idle at a non-busy time instant $t' \geq \hat{f}_{i,j}^2 + \rho$, no jobs can be scheduled to it. Then, the time interval after t' becomes non-busy. Thus, by the definition of $v_{i,j}^2$, $v_{i,j}^2$ must equal t'. Thus, the workload in EDF during $[\hat{f}_{i,j}^2 + \rho, \hat{f}_{i,j}^2 + x]$ is at least $(x - \rho) \times m$. Since $W(fd \cup Fd, \hat{f}_{i,j}^2, EDF) \leq \rho + m \times (x - \rho) + e_i^2$, the workload in EDF during $[\hat{f}_{i,j}^2, \hat{f}_{i,j}^2 + \rho]$ and $[\hat{f}_{i,j}^2 + x, \hat{f}_{i,j}^2 + x + e_i^2]$ is at most $\rho + e_i^2$. Even if all this work executes sequentially, it will complete by $\hat{f}_{i,j}^2 + x + e_i^2$ because $x \geq \rho$. Therefore, $c_{i,j}^2 \leq \hat{f}_{i,j}^2 + x + e_i^2$ and hence the contraposition holds. This lemma is proved. ◀

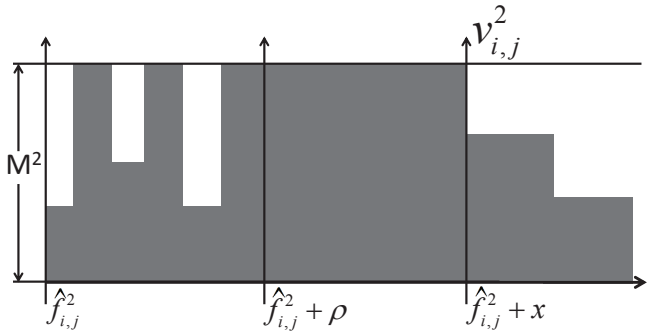


Figure 4 The structure of workload in Lemma 23.

5.2.3 Step 3: Deriving EDF-induced tardiness bound

By Lemma. 23, setting the upper bound on Eq. (25) to be at most $\rho + m \times (x - \rho) + e_i^2$ will ensure that the EDF-induced tardiness of $\tau_{i,j}^2$ is at most $x + e_i^2$. The resulting inequality is as follows.

$$x \times U_L^2 + E_L^2 + \sum_{\tau_l^2 \in \tau^2 \setminus \tau_i^2} (\lceil \frac{TB_l^1}{e_l^1} \rceil + 1) e_l^2 \leq \rho + m \times (x - \rho) + e_i^2 \tag{30}$$

Thus,

$$x \geq \frac{E_L^2 + D^*}{m - U_L^2} \tag{31}$$

where

$$D^* = (m-1)\rho - e_i^2 + \sum_{\tau_l^2 \in \tau^2 \setminus \tau_i^2} (\lceil \frac{TB_l^1}{e_l^1} \rceil + 1) e_l^2. \tag{32}$$

If x equals the greater of ρ and the right-hand side of Eq. (31) (recall that $x \geq \rho$ is required), then the EDF-induced tardiness of $\tau_{i,j}^2$ will not exceed $x + e_i^2$.

▶ **Theorem 24.** *With x as defined above, the EDF-induced tardiness for a job $\tau_{i,j}^2$ on type 2 processors in EDF schedule is at most $x + e_i^2$.*

5.3 Tardiness Bound of τ_i^2

We have bounded PS-induced tardiness and EDF-induced tardiness of any τ_i^2 in Secs. 5.1 and 5.2, respectively. As shown earlier in Fig. 3, the total tardiness of τ_i^2 scheduled under EDF on type-2 processors can be bounded by combining these two types of tardiness. The following theorem immediately follows.

▶ **Theorem 25.** *The tardiness of any task $\tau_i^2 \in \tau$ scheduled under EDF is at most*

$$TB_i^2 = TB_i^1 + P_i + \frac{E_L^2 + D^*}{m - U_L^2} + e_i^2, \tag{33}$$

where $D^* = (m-1)\rho - e_i^2 + \sum_{\tau_l^2 \in \tau^2 \setminus \tau_i^2} (\lceil \frac{TB_l^1}{e_l^1} \rceil + 1) e_l^2$.

6 Tardiness Bound of τ_i^k ($k > 2$)

In this section, we bound any τ_i^k's ($k > 2$) tardiness based on τ_i^{k-1}'s tardiness on type-$(k-1)$ processors. The proof procedure is exactly the same as how we bound τ_i^2's tardiness based on τ_i^1's tardiness in Sec. 5.

6.1 *PS*-induced Tardiness of τ_k^i

▶ **Theorem 26.** *Tardiness Bound of τ_i^k on PS Schedule is $TB_i^{k-1} + P_i$.*

Proof. We can prove this Theorem by mathematical induction. (i) When $k = 2$, the tardiness Bound of τ_i^2 in PS is $TB_i^1 + P_i$ according to Theorem 7. (ii) When $k > 2$, the tardiness Bound of τ_i^k in PS is $TB_i^{k-1} + P_i$, which can be proved using the same reasoning for proving Theorem 7 (by simply replacing subtask indexes 1 and 2 by indexes $k-1$ and k respectively). Thus, this theorem is proved. ◀

6.2 EDF-induced Tardiness of τ_i^k

In this section, we bound EDF-induced tardiness of τ_i^k exactly as how we bound such tardiness for τ_i^2 described in Sec. 5.2 in the following three steps.

Step 1: We derive an **upper bound** on competing work at $\hat{f}_{i,j}^k$ following the same methods described in Sec. 5.2.1.

$$W(fd \cup Fd, \hat{f}_{i,j}^k, EDF) \leq x \times U_L^k + E_L^k + \sum_{\tau_c^k \in \tau_H^k} (\delta_c(1 - u_c^k) + (e_c^k - \delta_c))$$

$$+ \sum_{\tau_l^k \in \tau^k \setminus \tau_i^k} (\lceil \frac{TB_l^{k-1}}{e_l^{k-1}} \rceil) e_l^k \leq x \times U_L^k + E_L^k + \sum_{\tau_l^k \in \tau^k \setminus \tau_i^k} (\lceil \frac{TB_l^{k-1}}{e_l^{k-1}} \rceil + 1) e_l^k. \tag{34}$$

Step 2: Determining **necessary condition** for tardiness to exceed $x + e_i^k$ following the same methods described in Sec. 5.2.2.

$$W(fd \cup Fd, \hat{f}_{i,j}^k, EDF) > \rho + m \times (x - \rho) + e_i^k, \tag{35}$$

where $\rho = \max_{\tau_i^{k-1} \in \tau^{k-1}} \{TB_i^{k-1}\}$.

Step 3: Deriving **EDF-induced tardiness bound** on type k processors following the same methods described in Sec. 5.2.3. Based on Eq. 34 and Eq. 35, we have

$$x \geq \frac{E_L^k + D^*}{M^k - U_L^k} \tag{36}$$

where

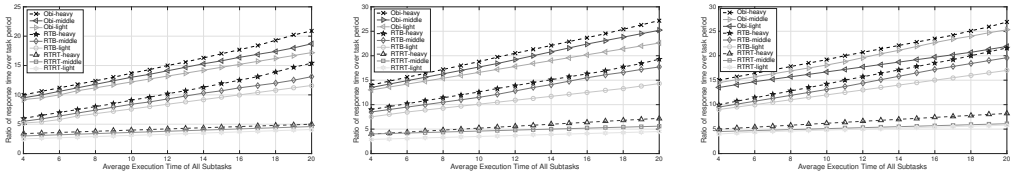
$$D^* = (M^k - 1)\rho - e_i^k + \sum_{\tau_l^k \in \tau^k \setminus \tau_i^k} (\lceil \frac{TB_l^{k-1}}{e_l^{k-1}} \rceil + 1) e_l^k. \tag{37}$$

Thus, the following corollary immediately follows.

▶ **Corollary 27.** *The tardiness of any subtask $\tau_i^k \in \tau$ scheduled under EDF is at most*

$$TB_i^k = TB_i^{k-1} + P_i + \frac{E_L^k + D^*}{M^k - U_L^k} + e_i^k$$

$$= TB_i^1 + (k-1)P_i + \sum_{j=2}^{k} \frac{E_L^j + D^*}{M^j - U_L^j} + \sum_{j=2}^{k} e_i^j, \tag{38}$$



(a) 4 types of processors **(b)** 6 types of processors **(c)** 8 types of processors

Figure 5 Magnitude of our derived analytical response time bounds.

where $D^* = (M^j - 1)\rho - e_i^j + \sum_{\tau_l^j \in \tau^j \setminus \tau_i^j} (\lceil \frac{TB_l^{j-1}}{e_l^{j-1}} \rceil + 1) e_l^j$.

Since the response time bound of any task τ_i can be obtained by simply using the derived tardiness bound of the task plus a P_i value, the response time bound can be easily calculated by the following Theorem 28.

▶ **Theorem 28.** *The response time bound for any dataflow task $\tau_i \in \tau$ with m subtasks scheduled under EDF, denoted RB_i, is*

$$\begin{aligned} RB_i &= TB_i^m + P_i \\ &= TB_i^1 + mP_i + \sum_{j=2}^{m} \frac{E_L^j + D^*}{M^j - U_L^j} + \sum_{j=2}^{m} e_i^j, \end{aligned} \tag{39}$$

where $D^* = (M^j - 1)\rho - e_i^j + \sum_{\tau_l^j \in \tau^j \setminus \tau_i^j} (\lceil \frac{TB_l^{j-1}}{e_l^{j-1}} \rceil + 1) e_l^j$ *and* TB_i^1 *is given by Eq. 3.*

7 Experiments

So far we have shown that EDF is optimal to support dataflow tasks with bounded response times on a heterogeneous computing platform. The magnitude of the analytical response time bound yielded under our analysis is of importance. To assess this, we have conducted experiments using randomly-generated task sets with widely varied parameters to examine how large the magnitude of the analytical response time bound is. Although we can only use synthesized task sets for assessment, we have verified that the generated range of parameters is wide enough to cover a couple of industrial dataflow-based cellular network systems seen in practice.

7.1 Experiment setup

The experiments compare the analytical response time bounds (RTB) given by Corollary 28 and the actual response time (RTRT) observed at runtime for randomly generated dataflow tasks scheduled under EDF in a heterogeneous multiprocessor system with m types of resources. For each resource type, there are 8 identical processors. When we generate subtasks, new subtasks were added until the total utilization of subtasks on each type of resources equals 8. Subtask utilizations were distributed differently for each experiment using three uniform distributions. The ranges for the uniform distributions were [0.005,0.1] (light), [0.1,0.3] (medium), and [0.3,0.8] (heavy). Task execution time were uniformly distributed over $(0ms, 20ms]$. Task periods were calculated from execution time and utilizations. For each task utilization distribution, $1,000,000$ task sets were generated for systems with $m = 4$,

6, or 8. We compare RTB, Obi, and RTRT under three utilization settings: per-subtask utilization is heavy, medium, and light.

7.2 Results

The obtained results are shown in Figure 5. Each curve plots the ratio of response time over task period averaged among all tasks generated in each experiment, as a function of the average execution time of all subtasks.

As seen in the figures, in all scenarios, RTB yields a much reduced response time bounds compared to Obi. For example, as shown in Fig. 5(a), when the average execution time of all subtasks is 12, RTB (Obi) yields a ratio of response time over task period of 8.6 (13.7), 9.3 (14.5), 10.1 (15.3), under light, medium, and heavy per-subtask utilizations, respectively. On average, RTB yields a 68% reduction on response time bounds compared to Obi. Moreover, in most scenarios, RTB yields reasonably small response time bounds compared to RTRT. For example, as shown in Fig. 5(a), when the average execution time of all subtasks is at most 12, the ratio of response time over task period under RTB ranges from 5 to 10, where the ratio under RTRT ranges from 4 to 6. Even for the worst-case scenario where the average execution time of all subtasks is large, the response time bound yielded under RTB is at most 3 times greater than RTRT. Given that RTRT represents the runtime response time observed in an actual GEDF schedule, we believe that the analytical bounded yielded under RTB is not only safe, but also reasonably tight in most cases. Another observation seen in all tested scenarios is that the analytical response time bounds under RTB become larger when per-subtask utilizations become heavier. This is because the task sets with heavy utilization may have the largest values of U_L^k (defined in Eq. 23), which results the largest response time bound according to Corollary 28.

8 Conclusion

In this paper, we investigate the problem of scheduling dataflow tasks on a heterogeneous computing platform with multiple types of resources with pre-defined affinity of tasks to subgroups of resources, which is motivated by many industrial applications seen in practice. We present a new set of analysis techniques that demonstrate that the classical and simple EDF scheduler can guarantee probably bounded response times for tasks with no capacity loss, thus proving EDF to be an optimal solution for this dataflow scheduling problem. Despite EDF's optimality in terms of schedulability, experiments also demonstrate that the magnitude of the response time bounds calculated under our analysis is reasonably small under all scenarios. This paper demonstrates the potential of applying EDF into practical industrial systems to schedule dataflow workloads with guaranteed bounded response times.

References

1 Ahmed Alhammad and Rodolfo Pellizzoni. Schedulability analysis of global memory-predictable scheduling. In *2014 International Conference on Embedded Software, EMSOFT 2014, New Delhi, India, October 12-17, 2014*, pages 20:1–20:10, 2014. doi:10.1145/2656045.2656070.

2 Björn Andersson, Gurulingesh Raravi, and Konstantinos Bletsas. Assigning real-time tasks on heterogeneous multiprocessors with two unrelated types of processors. In *Proceedings of the 31st IEEE Real-Time Systems Symposium, RTSS 2010, San Diego, California, USA, November 30 – December 3, 2010*, pages 239–248, 2010. doi:10.1109/RTSS.2010.32.

3 Bach Duy Bui, Rodolfo Pellizzoni, Marco Caccamo, Chin F. Cheah, and Andrew Tzakis. Soft real-time chains for multi-hop wireless ad-hoc networks. In *Proceedings of the 13th*

IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2007, April 3-6, 2007, Bellevue, Washington, USA, pages 69–80, 2007. doi:10.1109/RTAS.2007.34.

4 Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150, 2004. URL: http://www.usenix.org/events/osdi04/tech/dean.html.

5 UmaMaheswari C. Devi and James H. Anderson. Fair integrated scheduling of soft real-time tardiness classes on multiprocessors. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004), 25-28 May 2004, Toronto, Canada*, pages 554–561, 2004. doi:10.1109/RTTAS.2004.1317303.

6 UmaMaheswari C. Devi and James H. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. In *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS 2005), 6-8 December 2005, Miami, FL, USA*, pages 330–341, 2005. doi:10.1109/RTSS.2005.39.

7 Zheng Dong and Cong Liu. Closing the loop for the selective conversion approach: A utilization-based test for hard real-time suspending task systems. In *2016 IEEE Real-Time Systems Symposium, RTSS 2016, Porto, Portugal, November 29 – December 2, 2016*, pages 339–350, 2016. doi:10.1109/RTSS.2016.040.

8 Piotr Dziurzanski, Amit Kumar Singh, Leandro Soares Indrusiak, and Björn Saballus. Hard real-time guarantee of automotive applications during mode changes. In *Proceedings of the 23rd International Conference on Real Time Networks and Systems, RTNS 2015, Lille, France, November 4-6, 2015*, pages 161–170, 2015. doi:10.1145/2834848.2834859.

9 Tom Z. J. Fu, Jianbing Ding, Richard T. B. Ma, Marianne Winslett, Yin Yang, Zhenjie Zhang, Yong Pei, and Bingbing Ni. Livetraj: Real-time trajectory tracking over live video streams. In *Proceedings of the 23rd Annual ACM Conference on Multimedia Conference, MM'15, Brisbane, Australia, October 26-30, 2015*, pages 777–780, 2015. doi:10.1145/2733373.2807401.

10 Vincenzo Gulisano, Ricardo Jiménez-Peris, Marta Patiño-Martínez, Claudio Soriente, and Patrick Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Trans. Parallel Distrib. Syst.*, 23(12):2351–2365, 2012. doi:10.1109/TPDS.2012.24.

11 Ralf Jahr, Mike Gerdes, Theo Ungerer, Haluk Ozaktas, Christine Rochange, and Pavel G. Zaykov. Effects of structured parallelism by parallel design patterns on embedded hard real-time systems. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications, Chongqing, China, August 20-22, 2014*, pages 1–10, 2014. doi:10.1109/RTCSA.2014.6910546.

12 Shunsuke Kamijo, Yasuyuki Matsushita, Katsushi Ikeuchi, and Masao Sakauchi. Traffic monitoring and accident detection at intersections. *IEEE Trans. Intelligent Transportation Systems*, 1(2):108–118, 2000. doi:10.1109/6979.880968.

13 Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 – June 4, 2015*, pages 239–250, 2015. doi:10.1145/2723372.2742788.

14 Kam-yiu Lam, Jiantao Wang, Joseph Kee-Yin Ng, Song Han, Limei Zheng, Calvin Ho Chuen Kam, and Chun Jiang Zhu. Smartmood: Toward pervasive mood tracking and analysis for manic episode detection. *IEEE Trans. Human-Machine Systems*, 45(1):126–131, 2015. doi:10.1109/THMS.2014.2360469.

15 Hennadiy Leontyev and James H. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. In *Proceedings of the 28th IEEE Real-Time Systems Sym-*

posium (RTSS 2007), 3-6 December 2007, Tucson, Arizona, USA, pages 413–422, 2007. doi:10.1109/RTSS.2007.33.

16 Yang Li, Linh Thi Xuan Phan, and Boon Thau Loo. Network functions virtualization with soft real-time guarantees. In *35th Annual IEEE International Conference on Computer Communications, INFOCOM 2016, San Francisco, CA, USA, April 10-14, 2016*, pages 1–9, 2016. doi:10.1109/INFOCOM.2016.7524563.

17 Cong Liu and James H. Anderson. Supporting Soft Real-Time DAG-Based Systems on Multiprocessors with No Utilization Loss. In *Proceedings of the 31st IEEE Real-Time Systems Symposium, RTSS 2010, San Diego, California, USA, November 30 – December 3, 2010*, pages 3–13, 2010. doi:10.1109/RTSS.2010.38.

18 Cong Liu and James H. Anderson. An o(m) analysis technique for supporting real-time self-suspending task systems. In *Proceedings of the 33rd IEEE Real-Time Systems Symposium, RTSS 2012, San Juan, PR, USA, December 4-7, 2012*, pages 373–382, 2012. doi:10.1109/RTSS.2012.87.

19 José Marinho, Vincent Nélis, Stefan M. Petters, Marko Bertogna, and Robert I. Davis. Limited pre-emptive global fixed task priority. In *Proceedings of the IEEE 34th Real-Time Systems Symposium, RTSS 2013, Vancouver, BC, Canada, December 3-6, 2013*, pages 182–191, 2013. doi:10.1109/RTSS.2013.26.

20 André Oliveira Maroneze, Sandrine Blazy, David Pichardie, and Isabelle Puaut. A formally verified WCET estimation tool. In *14th International Workshop on Worst-Case Execution Time Analysis, WCET 2014, July 8, 2014, Ulm, Germany*, pages 11–20, 2014. doi:10.4230/OASIcs.WCET.2014.11.

21 Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio C. Buttazzo. Response-time analysis of conditional DAG tasks in multiprocessor systems. In *27th Euromicro Conference on Real-Time Systems, ECRTS 2015, Lund, Sweden, July 8-10, 2015*, pages 211–221, 2015. doi:10.1109/ECRTS.2015.26.

22 Jan Nowotsch and Michael Paulitsch. Quality of service capabilities for hard real-time applications on multi-core processors. In *21st International Conference on Real-Time Networks and Systems, RTNS 2013, Sophia Antipolis, France, October 17-18, 2013*, pages 151–160, 2013. doi:10.1145/2516821.2516826.

23 Alessandro Vittorio Papadopoulos, Martina Maggio, Alberto Leva, and Enrico Bini. Hard real-time guarantees in feedback-based resource reservations. *Real-Time Systems*, 51(3):221–246, 2015. doi:10.1007/s11241-015-9224-1.

24 Keivan Ronasi, Vincent W. S. Wong, and Sathish Gopalakrishnan. Distributed scheduling in multihop wireless networks with maxmin fairness provisioning. *IEEE Trans. Wireless Communications*, 11(5):1753–1763, 2012. doi:10.1109/TWC.2012.030812.110493.

25 Kecheng Yang, Ming Yang, and James H. Anderson. Reducing response-time bounds for dag-based task systems on heterogeneous multicore platforms. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS 2016, Brest, France, October 19-21, 2016*, pages 349–358, 2016. doi:10.1145/2997465.2997486.

26 Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *4th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'12, Boston, MA, USA, June 12-13, 2012*, 2012. URL: https://www.usenix.org/conference/hotcloud12/workshop-program/presentation/zaharia.

27 Haibo Zeng and Marco Di Natale. Outstanding paper award: Using max-plus algebra to improve the analysis of non-cyclic task models. In *25th Euromicro Conference on Real-Time Systems, ECRTS 2013, Paris, France, July 9-12, 2013*, pages 205–214, 2013. doi:10.1109/ECRTS.2013.30.

Design and Implementation of a Time Predictable Processor: Evaluation With a Space Case Study*

Carles Hernández[†1], Jaume Abella[‡2], Francisco J. Cazorla[3], Alen Bardizbanyan[4], Jan Andersson[5], Fabrice Cros[6], and Franck Wartel[7]

1 **Barcelona Supercomputing Center (BSC), Barcelona, Spain**
carles.hernandez@bsc.es
2 **Barcelona Supercomputing Center (BSC), Barcelona, Spain**
jaume.abella@bsc.es
3 **Barcelona Supercomputing Center (BSC) and IIIA-CSIC, Barcelona, Spain**
francisco.cazorla@bsc.es
4 **Cobham Gaisler, Gothenburg, Sweden**
alen.bardizbanyan@gaisler.com
5 **Cobham Gaisler, Gothenburg, Sweden**
jan@gaisler.com
6 **Airbus Defense and Space, Toulouse, France**
fabrice.cros@airbus.com
7 **Airbus Defense and Space, Toulouse, France**
franck.wartel@airbus.com

Abstract

Embedded real-time systems like those found in automotive, rail and aerospace, steadily require higher levels of guaranteed computing performance (and hence time predictability) motivated by the increasing number of functionalities provided by software. However, high-performance processor design is driven by the average-performance needs of mainstream market. To make things worse, changing those designs is hard since the embedded real-time market is comparatively a small market. A path to address this mismatch is designing low-complexity hardware features that favor time predictability and can be enabled/disabled not to affect average performance when performance guarantees are not required. In this line, we present the lessons learned designing and implementing LEOPARD, a four-core processor facilitating measurement-based timing analysis (widely used in most domains). LEOPARD has been designed adding low-overhead hardware mechanisms to a LEON3 processor baseline that allow capturing the impact of jittery resources (i.e. with variable latency) in the measurements performed at analysis time. In particular, at core level we handle the jitter of caches, TLBs and variable-latency floating point units; and at the chip level, we deal with contention so that time-composable timing guarantees can be obtained. The result of our applied study with a Space application shows how per-resource jitter is controlled facilitating the computation of high-quality WCET estimates.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

* The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the PROXIMA Project grant agreement no. 611085 (http://www.proxima-project.eu). This work has also been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2015-65316-P and the HiPEAC Network of Excellence.
† Carles Hernández is jointly funded by the Spanish Ministry of Economy and Competitiveness and FEDER funds through grant TIN2014-60404-JIN.
‡ Jaume Abella has been partially supported by the Ministry of Economy and Competitiveness under Ramon y Cajal postdoctoral fellowship number RYC-2013-14717.

Keywords and phrases processor design, performance guarantees, multicore, industrial case studies, application of real-time technology in realistic systems

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.16

1 Introduction

Software is becoming the main competitive advantage in embedded real-time products fuelled by the goal of achieving autonomous (i.e. software-controlled) vehicles in market sectors such as automotive, aerospace and railway. In this line, software increasingly implements more complex functionalities with relentless demands for guaranteed computing power across different domains [18, 22]. This has motivated high-performance processor chip manufacturers (e.g. Intel, NVIDIA, and ARM) to start adding time-predictable features in their processor designs [46, 16]. In the same line, processor companies already targeting the embedded real-time domain, e.g. Infineon and Cobham Gaisler, have been motivated to evolve very rapidly from simple micro-controllers to more advanced processor designs [23, 10].

The guaranteed performance requirements of real-time systems challenges the adoption of advanced performance-improving hardware features: as resources become more statefull and interact in more complex ways, deriving tight timing bounds is more difficult. Furthermore, this complicates providing timing analysis techniques with information about hardware behaviour. For static timing analysis this includes access delays of hardware resources which are increasingly hard to derive from manuals, forcing practitioners to stick to measured values [33]. For measurement-based timing analysis (MBTA) assessing whether the execution scenarios captured at analysis effectively cover those bad (worst) conditions that can arise at operation requires dealing with more and more hard-to-track low-level hardware details.

Overall, the quality of the Worst-Case Execution Time (WCET) estimates derived with MBTA (the focus on of this paper and widely adopted in the real-time domain [48, 49]) depends on the ability of the user to build test scenarios at analysis time in which program's execution conditions are close to those that can lead to the WCET during operation. This requires capturing the impact of the sources of jitter (SoJ) in the measurement observations taken at the analysis phase. This ability had by users with simple hardware, diminishes with the advent of more complex hardware: users are increasingly forced to deal with low-level hardware SoJ (e.g. requests alignment and cache mapping), while their real focus is on problems at higher levels of abstraction (e.g. algorithm and end-to-end models). Users neither have the will, (and in many cases) nor the means to exercise this level of control on low-level hardware. This is in contrast to other high-level SoJ, such as execution path coverage, for which clear metrics are defined (e.g. DC and MC/DC) and tools exist to help the user to reach a given target coverage. Hence, solutions that help increasing confidence on measurements without requiring the user to deal with processor internals are fundamental to enable the use of more complex processors in real-time embedded domains.

In this applied study, we present the lessons learned in the PROXIMA EU project [38] designing and implementing LEOPARD (LEON-based probabilistically analyzable processor design), a 4-core processor based on Cobham Gaisler's LEON processor family (deployed in the Space domain). LEOPARD's design exposes the jitter of micro-architectural resources so that the execution time measurements taken at analysis factor in the impact of those resources. LEOPARD helps the user providing evidence, as needed for safety standards, that analysis-time execution scenarios upperbound those that can arise during operation. This is achieved by introducing several low-complexity features in the baseline processor that can be

activated/deactivated to reduce impact on average performance. This helps the ultimate goal of having (from the hardware point of view) one design that fits the requirements of several domains and increasing the cost-effectiveness of MBTA since it reduces its application costs while helps achieving the level of confidence required by the domain prescriptions [2]. LEOPARD identifies and attacks the following low-level SoJ.

- The baseline floating-point unit takes variable latency for some operations depending on the particular values operated. To control this SoJ with standard MBTA, the user would need to control the particular values operated at analysis ensuring their representativeness w.r.t. those that can appear during operation. Instead, in the LEOPARD design all floating-point operations are made to work on their respective worst latency, making their impact on execution time to be captured in the analysis-time measurements.
- The use of cache-memory resources (i.e. the data and instruction cache and Translation Lookaside Buffers, TLBs) requires the end user to control memory allocation of code/data, and hence their cache layout at analysis, so that it is the same as during operation. However, even small variations in the order in which the object files are linked together and in other elements of the memory layout (e.g. environmental variables) may significantly affect memory layout – which hence must be controlled by the user. LEOPARD removes this requirement by implementing random placement (and enhancing already deployed random replacement), breaking the dependence among memory allocation of data/code and cache layout. As a result, by performing enough runs [31], the end user can probabilistically assess the impact caches have on execution time.
- At the chip level, we propose an AMBA-compatible time-composable random arbitration to handle contention. To balance time-composability and WCET tightness, we implement a credit-based random (permutation) arbitration policy that randomizes the impact of contention on request's timing behaviour while preserving fairness across cores.
- For the shared L2, we (1) implement random placement and replacement; and (2) assign different cache ways to the different cores (as supported in other architectures) to control contention, while preserving the effective management of cache coherence in the L2 cache.
- Further, we developed high-speed transparent Ethernet tracing features to simplify validation and verification and applicability of industrial timing analysis tools and methods.

Results with stressing applications and a space case-study from Airbus Defense and Space show that LEOPARD's performance guarantees are significantly better than those that could be achieved with Commercial off-the-shelf (COTS) LEON processors. LEOPARD also preserves average performance to the levels of the baseline design. Finally, implementation results show that LEOPARD incurs low area and delay overheads to achieve timing predictability and high-performance tracing capabilities.

The rest of the paper is organized as follows. Section 2 provides some background on MBTA and hardware design favoring it. Section 3 and Section 4 describe the baseline processor and the changes implemented to ease timing analyzability at core level and at chip level, respectively, along with some tracing support enhancements. Section 5 evaluates LEOPARD ability to control identified SoJ. Section 6 presents the most relevant related works. Concluding remarks are presented in Section 7.

2 Background and MBPTA requirements

2.1 Safety Standards

Criticality originally emanates from functional safety, with several existing safety-related standards in different domains: the generic IEC61508 and domain specific variants of it:

ISO26262 in automotive and EN-50126/50128/50129 in the rail domain; and others like ECSS-Q-ST-80C in Space and DO-178C/DO254 for software/hardware aeronautics.

A task overrun should never lead to an unsafe state of the system, which would mean a bad-designed safety solution. Instead, a safety process is defined (according to the corresponding standard) covering the definition of safety goals and requirements, and a safety strategy in general, to mitigate the risk that hardware or software misbehaviour causes a system failure. As the criticality of the software component under analysis increases, more mechanisms are put in place (replication, online monitoring, watchdog) to detect and react to undesired situations.

Many standards require hardware to provide means to demonstrate sufficient independence between different software units. Partitioning mechanisms and monitors are the two preferred means to reach these goals. The use of multicore processors, however, complicate this approach since, although time and space partitioning is achieved [1] (i.e. each task/partition is assigned slots in which only it can use the CPU and it cannot modify another partition's memory space and vice-versa), timing interference is not easily prevented. New requirements are imposed on the hardware and software such as controlling sources of jitter (interference channels in CAST32-A [9] for aerospace).

2.2 Timing

Predictability defines the ability of predicting (a priori) when an event or set of events will occur. While in general in real-time systems predictability is understood as determinism, it has been shown that predictability can be also achieved in probabilistic terms [8].

MBTA involves an operation phase in which the system is deployed, and a analysis (pre-deployment) phase comprising several test campaigns in which the application is run on the target hardware. The goal of MBTA is to derive WCET estimates from the execution runs of the program performed at analysis and provide evidence that those estimates hold valid during the operation of the system. This requires that the execution conditions exercised experimentally at analysis capture those worst-case conditions that can occur during operation. Interestingly, when evidence obtained is sufficient, MBTA can be used for high-integrity software, e.g. DAL-A functions in avionics [28].

With MBTA, user's ability to design stressful operation-representative test scenarios plays a fundamental role in the reliability of the derived WCET estimates. High-level SoJ such as path coverage can be tracked and controlled (as presented in the introduction). However, the control the user has on hardware SoJ diminishes with the advent of more complex features since the cost of controlling the entire design space of all hardware SoJ in such complex designs is unaffordable. For instance, i) the impact of FPU jittery operations would require the user to understand which operands result in longer latencies and software support to track operated values (since hardware support does not exist for that); and ii) capturing the execution time variability consequence of different cache layouts would require understanding the impact of cache layout on WCET estimates. However, in general, it is hard for the user to design experiments in which bad (worst) cache layouts are enforced when even small changes in their memory layout may cause significant jitter in the observed timing behaviour [30]. Fixing the memory layout is only possible during very late phases of the development process, going against the incremental software integration principle and the definition of a global (best-) worst-case memory layout for an application comprising several tasks is a generally intractable problem [30].

[1] Partitioning is no yet achieved or within under specific conditions on multicores.

Time composability is another desired property for derived WCET estimates when controlling the impact of SoJ. First, time composability across incremental software integration [30] ensures that early phase WCET estimates (ideally at the unit testing level) hold across integration reducing the risk of costly late detection of timing violations. And second, time composability at the multicore level ensures that the WCET estimate of a task does not depend on its co-runners' load on shared resources. This provides independence across tasks, that can easily be developed by different software providers in integrated systems (e.g. Integrated Modular Avionics, IMA), allowing parallel development and testing.

MBTA requires collecting execution time traces on the target platform. Transparent trace collection also requires hardware support so that time (or performance monitoring counter) readings can be collected without impact on programs execution. Furthermore, code instrumentation can be performed with hardware [14], causing no overhead on program execution time, but with high associated cost, or at software level. The latter, while it is more generic and portable, it can cause the *probe* effect: instrumentation code create discrepancies in terms of timing w.r.t. the non-instrumented code, complicating timing Validation and Verification. A recent work [13] shows that *nop* operations can be used to substitute instrumentation instructions in a way that simplifies qualification/certification and at the same time reduces the impact of instrumentation.

2.3 Requirements

LEOPARD controls the jitter of the different SoJ in two different ways. First, with deterministic bounding by forcing resources to work on their worst (deterministic) latency. And second with probabilistic upperbounding that makes resources have a randomized timing behaviour, resulting in a probabilistic distribution of execution times that hold during operation [26]. Hence, when enough runs are performed probabilistic upperbounds to execution time [31] can be derived. This principle emanates from probabilistic and statistics theory, where a random variable can be modelled based on a sample of observations with increasing confidence and accuracy as the size of the sample grows.

Probabilistic distributions are handled with a variant of MBTA, called measurement-based probabilistic timing analysis (MBPTA). MBPTA, which builds on representative execution time observations (obtained via the mechanisms to control jitter described above), deploys statistical analysis through Extreme Value Theory (EVT) [27]. EVT enables deriving the probability that bad behaviour of several hardware SoJ, whose impact has been captured in the analysis time runs, are triggered in the same run. This is a powerful solution that reduces MBTA application costs not needing the user to design experiments in which bad behaviour of all SoJ (e.g. bus, cache, FPU) are simultaneously triggered. Overall, the requirements to penetrate high-performance hardware designs while facilitating MBTA (in the form of MBPTA) are:

1. Exposing the impact of SoJ so that (i) representative operation-phase execution time measurements are collected during analysis without needing the end user to design complex experiments to control them, and hence enabling deriving high-quality WCET estimates at low cost; and (ii) derived time composable estimates hold across incremental software integration [30] and are independent of contender's load on the shared resources.
2. Incurring low-implementation overhead, specially in terms of processor complexity to minimize the cost of verification.
3. Reducing the impact on average performance by making time-predictable features to be activated/deactivated depending on the time predictability needs of the system instance.
4. Providing high-bandwidth transparent tracing with no interference on program's execution time. Enabling collecting traces from all cores simultaneously for increased observability,

Table 1 Input value examples triggering different latencies for FDIVD and FSQRTD.

Op.	Lat	Input 1		Input 2	
		hexa	decimal	hexa	decimal
FDIVD	15	0xBFF0000000000000	-1.0	0x4000000000000000	2.0
FDIVD	18	0x001ABC0000000010	$3.717(...) \cdot 10^{-308}$	0x3FF000400A07610C	1.00006107(...)
FSQRTD	23	0x4030000000000000	16.0		
FSQRTD	26	0x4008000000000000	3.0		

.

3 Core Design

In this section we focus on the main SoJ at the core level, while we cover chip-level SoJ in Section 4. Both sections first describe the baseline design and then the proposed changes.

3.1 Baseline Design

The baseline design corresponds to an enhanced implementation of a LEON3 [17] resembling the NGMP processor [10], a multicore processor candidate for the European Space Agency missions in the next years.

Pipeline. The processor implements a pipelined architecture comprising the following stages: fetch (F), decode (D), register access (RA), execution of non-memory operations (Exe), DL1 access (M), Exceptions (Exc) and write back (WB). The execution units comprises an integer and a floating-point unit (FPU).

1. The FPU takes a variable latency depending on the particular values operated for divisions (FDIVD) and square roots (FSQRTD). Table 1 provides a summary of those jittery FP operations and their associated jitter.
2. The core incorporates a static branch-always predictor that starts fetching instruction from the branch target address. On a prediction hit, 1 or 2 clock cycles are saved. Under a mispredicted branch, instruction hits in IL1 change LRU replacement history (also hits in the L2), while misses in IL1 and the L2 pollute cache contents.

DL1 and IL1. The target processor comprises first level instruction (IL1) and data (DL1) caches, with the DL1 implementing a write-through no write allocate policy. The bus propagates DL1 and IL1 misses to the L2 cache (see Figure 3) is discussed in Section 4. IL1 and DL1 are 16KB with 4-way set-associative caches with modulo placement and LRU replacement. L1 caches also support cache freezing to ensure that the execution of the interrupt handler will not evict any cache line and when control is returned to the interrupted task, the cache state is identical to what it was before the interrupt.

TLBs and cache coherence support. To support memory translation (and space partitioning) the LEON processor is provided with a Memory Management Unit (MMU) comprising TLBs of 64 entries for instructions and data deploying LRU replacement.

The LEON3 processor uses Virtually indexed, virtually tagged (VIVT) first level caches so that virtual addresses are used for both the index and tag bits. This caching scheme results in fast lookups, since the MMU does not need to be looked up first to determine the physical address for a given virtual address. Since the LEON3 is a shared memory multiprocessor

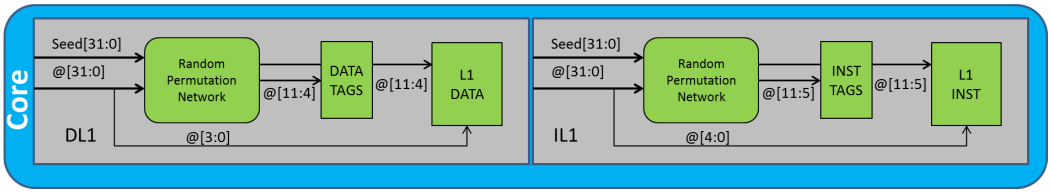


Figure 1 Sketch of the implementation of the Random Modulo technique.

(SMP), on every access to the on-chip bus, the address is snooped by all the caches to check if these data are present in the cache and, consequently, need to get invalidated. To speed up this process, the LEON3 cache also includes the physical tags in a separate SRAM structure so in every access to the on-chip memory, snoop hits are concurrently detected. In case of a snoop hit – a write operation is performed to data stored in the cache – the corresponding cache line is invalidated.

Since VIVT caches suffer from *aliasing*, the LEON3 fixes the cache way size to match (MMU) page size (4KB). With this restriction, that is imposed by hardware, synonyms are enforced to be placed in the same set with modulo placement and thus, can be safely invalidated when a snoop hit is detected. The MMU provides address translation of both instructions and data via page tables stored in memory. When needed, the MMU accesses the page tables to calculate the correct physical address. The latest translations are stored in TLBs. The MMU also provides access control, making it possible to "sandbox" unprivileged code from accessing the rest of the system.

3.2 LEOPARD design

The design presented in this section builds on the design we presented in [20] that focused on single core sources of jitter and included only the implementation of random placement [26] instead of random modulo placement in the L1 caches.

Cache resources. Random replacement has been implemented in IL1 and DL1 caches and TLBs (ITLB and DTLB) building on the *random seeds* provided by the pseudo-random number generator (PRNG) described later in this section. DL1 and IL1 also deploy random placement to release the user from controlling the placement of all programs and memory objects at analysis and during operation. For that purpose we have implemented random modulo (*RM*) [21], fitting the requirements of the specific processor implementation (e.g. number of cache sets, delay constraints).

RM placement performs a random permutation of the bits used to index the cache set (see Figure 1). By doing so, like modulo, *RM* retains spatial locality properties. Let W be the way size and A and address such that $(A \mod W) = 0$. With *RM* any pair of cache line addresses in the range $[A, A + W)$, which are said to belong to the same *segment*, are prevented from conflicting into the same cache set. For instance, if addresses A and B belong to the same cache segment (i.e. $\lfloor A/W \rfloor = \lfloor B/W \rfloor$) and with modulo are mapped to different sets (k_A and k_B respectively), RM randomizes the index bits such that (in every run) with a seed $seed_i$, A is mapped to any (random) set $l_A = set_{rm}^{seed_i}(A)$ and B to $l_B = set_{rm}^{seed_i}(B)$ and l_A and l_B are necessarily different. Hence, *RM* removes the dependence between memory mapping and cache layout by ensuring that the index permutation covers cache conflicts probabilistically during the analysis phase. During the analysis phase a different seed is employed to cover the cache conflicts that can occur during operation.

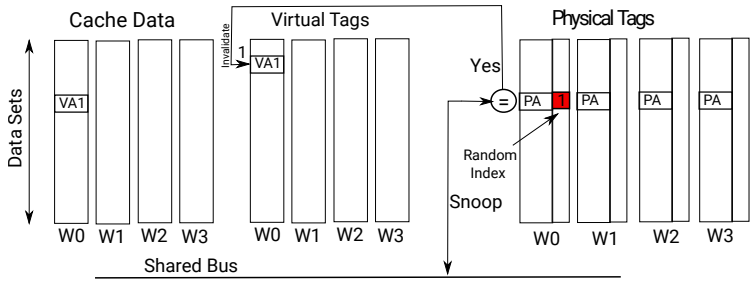


Figure 2 Randomised LEON3 cache configuration. General approach for invalidation.

In the baseline processor, we have detected a single source of timing anomalies. It arises when an instruction i that would has missed in DL1 and hit in L2, actually misses in L2 because before it accesses L2, a younger instruction j misses in both IL1 and L2 and evicts the L2 line where i would hit. Hence, delaying j would allow i to hit in L2 and execute faster. With cache randomization j can evict i line in L2 with a probability $1/S_{L2}$, where S_{L2} stands for the number of L2 cache sets. Hence, if enough runs are performed the impact of this situation would be captured in the measurements. It is part of our future work enforcing IL1 misses to wait for accessing the bus until all older instructions have been resolved in DL1 to avoid any reordering. Users stick to their current practice to handle this situation.

Cache Coherence. The support for cache coherence in a randomised cache design using a MMU introduces some complexities in the cache configuration. In a cache with random placement the index does not only depend on the modulo operation, like in a regular cache with modulo, but also on the upper bits of the address. Then, since virtual and physical addresses referring to the same data have different upper bits (e.g. PA=0x40000004 and VA=0x00000004) the index computed for the virtual address using the random placement function will not necessarily correspond to the one where the physical tag is located. This leads to a conflict for resolving the invalidation of the data affected by snoop hits. Furthermore, the mismatch between indexes makes that synonyms, i.e. two virtual addresses that are mapped to the same physical address, are placed at arbitrary locations in the cache rather than in the same cache set.

To solve this, we have designed a software/hardware solution that requires on one hand, moderate hardware changes in the cache configuration and on the other hand, forcing the OS to flush caches on every context switch. Hardware modifications of the generic solution consist of extending cache contents to keep the randomised index bits that are required to identify the cache set that needs to be evicted in case of a snoop hit. Figure 2 shows the required hardware changes. The randomised index bits are written in the same SRAM structure where the physical tag is and are updated every time new data are fetched into the cache. Finally, the flush on context switch functionality has to be implemented by the OS to ensure that only one address space is present in the cache at a time[2].

Branches. The default configuration of the processor allows issuing fetch requests to the L2 cache on a IL1 miss under branch speculation. As explained before, if the branch is mispredicted, this may pollute IL1 and L2 cache contents, and their replacement history even

[2] Flush on context switch is also a common way to solve the synonym problem in regular cache designs where page size does not match cache way size.

on hits. By using random replacement, replacement becomes stateless and hence, cache hits do not change its state. On the other hand, in order to avoid any cache state modification due to mispredicted branches, we have modified the bitstream to forbid cache misses to be served under speculated branches. This is done by programming the appropriate bit of the ASR17 configuration register. Note that jitter caused by different paths is handled by timing analysis techniques, and hence it is not covered in this hardware paper. An example of one of those techniques is Extended Path Coverage [50] that derives upper bounds of the probabilistic execution time of the complete program under analysis even when the user-provided input vectors do not exercise the worst-case path.

Worst-latency FPU. The implementation of FDIV/FSQRT operations for double precision has been modified so that during the analysis phase, they exhibit a fixed latency that matches their highest latency. In particular, those operations are non-pipelined and iterate in some internal stages of the FPU until the result is produced, thus allowing early termination of some operations. In analysis mode, the early termination signal is inhibited, thus enforcing all those operations to experience their highest latency regardless of the input values operated. At operation time, those operations are allowed to take a variable time depending on the values operated. The net result is that their jitterless timing behaviour at analysis time upper-bounds that during operation, thus releasing the end user from having to control the impact on execution time of the particular values involved. Note that the same approach of delaying execution until its worst case have been used to handle contention in shared resources [36][4].

Creating a source of randomization. A SIL-3 IEC-61508 pseudo-random number generator (PRNG) [3] has been incorporated in the design to feed appropriately the components requiring time-randomized behaviour. The PRNG is based on linear feedback shift registers [6] and consists of a pool of programmable random numbers. In general, the sequence of numbers provided by the PRNG must be long enough to ensure repetition occurs after a period long enough for any potential correlation between the outcomes of the system at different time instants to be probabilistically irrelevant. The degree of randomness of the used PRNG was validated statistically by checking the lack of meaningful patterns, repetitions, imbalance between different values, etc. for a number of bit sequences generated. This was measured with the tests provided by the US National Institute of Standards and Technology [40]. The PRNG provides randomised bits for random replacement: 2 for DL1 (4-way), 2 for IL1 (4-way), and 6 for DTLB and 6 for ITLB (64-entry fully-associative both of them), so 16 bits per core plus few extra bits for the random arbitration in the bus.

4 Chip Design

4.1 Baseline Design

At the chip level the components with the highest jitter impact are the L2 cache controller, the on-chip bus, and the memory controller.

On-chip Bus. An AMBA AHB compatible bus is included in the processor to handle concurrent requests to the different slaves included in the system. The arbiter employs a round-robin arbitration between the different masters, including the processor cores, to determine the one that gets access to the bus. In the baseline design, round-robin is implemented by rotating the priority after every bus transfer.

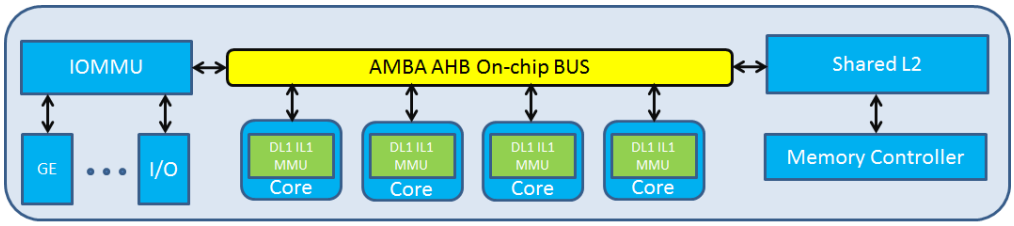


Figure 3 LEOPARD processor block diagram.

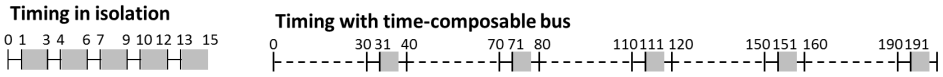


Figure 4 Example of high contention caused by the original scheme with random permutations.

Shared L2 cache. The baseline processor includes a 4-way shared L2 cache. The replacement policy can be configured as LRU (least-recently-used) or master-index (the way in which the replacement occurs is determined by the master index). The cache way is 32KB with a cache line size of 64 bytes. Requests from the cores to the shared L2 cache are arbitrated at the on-chip bus and the bus is kept locked, i.e. no further request are accepted, until the current request is processed by the L2.

Shared Memory Controller. The memory controller included in the baseline design is a DDR2 SDRAM controller with AMBA AHB back-end. The controller interfaces a 64-bit wide DDR2 memory with the L2 and acts as a slave on the AHB bus where it occupies a configurable range of the address space for DDR2 SDRAM access. The memory implements a FIFO with room for two write bursts to maximize throughput, since the second write can be written into the FIFO while the first write is being written to the DDR memory.

IOMMU. To ensure global spatial partitioning is provided efficiently, the baseline LEON3 COTS design already implements an IOMMU. This is significantly important for properly handling direct memory access (DMA) transfers and interrupts [32]. The IOMMU functionality of the core implemented in this processor provides address translation and access protection on the full 4GiB AMBA address space.

4.2 LEOPARD design

On-chip Bus. We modify the arbiter to implement random permutations [25]. Interestingly AMBA standard does not specify any arbitration policy for AHB buses, so our bus is fully AMBA compliant. Further, while the baseline platform we use builds on the AMB AHB specification [7], the modifications we have implemented in the bus can also be applied to more recent bus protocols like the AXI [7]. The random permutation arbiter defines windows with as many slots as arbitrated cores, N_c, with all slots having the same duration. Slots are allocated randomly to cores in each window so each core has *exactly* one slot per window and it can access the bus only during its assigned slots. This allows obtaining time-composable pWCET estimates since no assumption is made on the number and duration of the requests of the other contenders. With random permutations each core gets on average $\frac{1}{N_c}$ of the slots and the maximum waiting time due to contention is shorter than 2 arbitration windows: $MaxContention < L \cdot (2 \times N_c - 1)$, where L stands for the slot duration. The maximum

contention occurs when one core is allocated the first slot in one window and the last slot in the following window, since all remaining cores are arbitrated twice in between.

Random permutations is fair granting cores access to the bus with the same frequency. However, it is not fair in terms of the bus time granted to each core. This is so because to achieve composable bounds, slot duration (L) must be large enough to allow the longest potential request to be served. For instance, if the maximum duration of a request is 56 cycles, then $L \geq 56$, otherwise, the longest requests could not be granted access to the bus without using slots belonging to other cores. However, using a long, fixed slot may cause a significant impact in the performance of tasks with requests whose duration is much lower than L. This is better illustrated with the example in Figure 4 that shows how a program with 1-cycle requests runs for 15 cycles in isolation and takes 193 cycles when $N_c = 4$ and $L = 10$: when a given software Unit of Analysis (UoA) accesses the bus, it may have to wait for $(N_c - 1 = 3)$ slots assigned to the contenders. Let assume that the UoA accesses the bus in cycle 30 and releases it in cycle 31; further from 31 to 33 it performs some computations and eventually, at cycle 33 the UoA needs to access the bus again. However since it has only 7 cycles remaining in its slot and the duration of the requests is unknown a priori (e.g. it could be 1 cycle for a L2 hit and 10 for a L2 miss in this example), the request is not allowed to proceed and has to wait until the beginning of its next slot in cycle 70. The same scenario repeats for each request so that the last one is granted access in cycle 190, served in 191 and the program completes in cycle 193. As we can see, even if the UoA is granted access during 25% of the time to the bus, its slowdown is 12.9x in a 4-core setup. Note that for the sake of this example we have assumed that slots are allocated homogeneously in time to the core of the UoA. Randomly allocating slots to tasks would bring the very same results as in the example on average.

In order to mitigate the impact that dealing with long requests may have on the quality of time-composable WCET estimates we modify the arbiter to implement a credit-based arbitration (CBA) scheme [43], fitting it to the particular characteristics of the LEON3 multicore and its bus arbiter. CBA allocates each core a given credit (budget) matching $MaxL$ – the longest time a request can occupy the bus, and which can be derived either analytically or by measurements. Then, arbitration is performed across all cores with pending requests and an available budget of exactly $MaxL$ cycles. When a request is granted access to the bus, the budget of the corresponding core is decreased by the bus hold time. For instance, this is implemented by decreasing by 1 the budget of the core using the bus. In parallel, every cycle all cores get their budget increased as shown in Equation 1, where $Budget_i(t)$ stands for the credits (cycles) of core i in cycle t

$$Budget_i(t+1) = \min(Budget_i(t) + 1/N_c, MaxL)\,. \tag{1}$$

The budget assigned to each core saturates at $MaxL$ to prevent the case in which one core spends long time not using the bus and then it hogs the bus during a long time period. This approach reduces the maximum contention experienced by a given core but at the expense of wasting some bandwidth (cores cannot accumulate budget beyond $MaxL$). A similar approach that allows cores to go beyond $MaxL$ was proposed in [5]. Note also that although conceptually the budget is increased by a fraction, this can be implemented by multiplying all factors in Equation 1 by N_c. In that case, when using the bus, the budget should also be decreased by N_c every cycle instead of by 1.

CBA operation is illustrated in Figure 5 where each core always has requests ready and the random permutation generated by the arbiter are as follows: [A, B, C, D], [B, A, D, C], [D, B, C, A], [B, C, A, D], [A, C, B, D], [A, B, C, D]. Requests from cores A, B and C

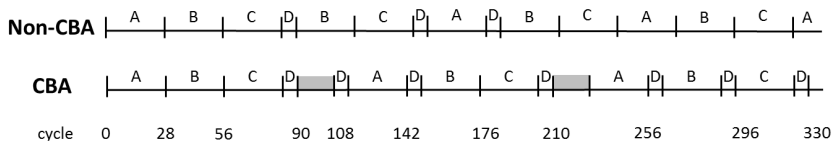


Figure 5 Chronogram showing requests arbitrated with and without CBA. The time scale at the bottom is only approximate since time intervals in the chronogram are not exactly proportional to the time they take for the sake of readability.

take 28 cycles and from D take 6 cycles. We focus in the first 336 cycles (the time needed to hypothetically send 3 28-cycles requests from each core). As shown, without CBA only 3 requests from core D are served in 336 cycles. However, with CBA, in this time frame core D gets 7 requests arbitrated. In the example, the arbiter grants access to a core if it has enough budget. Otherwise, the random list of core ides is searched until a core with enough budget is found. For instance, after the first request of D is served (cycle 90) no core has $MaxL$ budget. At that point the one recovering its budget earlier is D (in cycle 108), so in cycle 108 D is granted access and we skip B and A in the sequence (and also D since it is arbitrated).

Shared L2. The shared L2 has been configured to implement per-way partitioning (master-index replacement). With this configuration each core is provided with one out of the 4 ways available. As for L1 caches, we implement random placement to make WCET estimates hold regardless of the actual memory layout that will be at system deployment. However, unlike for L1 caches, we use hash-based random placement [26] in the L2. The reason is that random modulo [21] forces memory objects to preserve the cache way alignment. For the L1 cache the way size is equal to the OS page size (4KB) and thus, keeping such restriction is not only doable but preserved by default in the general case. However, imposing such way alignment for L2 caches is not realistic since L2 cache ways are generally much bigger than the page size. Finally, we have also implemented random replacement, but when the L2 is fully partitioned, random replacement is not required since each core is only entitled to evict lines from its (single) corresponding way.

Shared Memory Controller. The access latency to shared resources needs to be deterministic upperbounded or randomized to control jitter, relieving the end-user from the burden of controlling how requests from the different tasks align. In our design, no modification was required since the L2 included in the design does not allow any further request to be sent to memory while another request is being served. Also, response time of the memory controller needs to be made constant for each request type to remove dependencies across requests from different cores. This feature was implemented following the same principle as for the FPU unit in the core: delaying requests so that they experience the maximum latency allowed. We are currently in the process of enabling split requests in the bus to increase memory-level parallelism. Besides the functional technicalities, this feature is an important SoJ with potential side effects: at least it is required to use separated request queues for each core in the memory controller. Further, a suitable arbitration mechanism (e.g. random permutations in our case) across cores (so across queues) is also required in the memory controller. Using separate queues, prevents one core to clog the others, despite they have independent cache partitions, as it has been shown for some ARM architectures [45].

High-Speed Tracing. We extended the baseline tracing capabilities to support powerful timing analyses minimizing (or eliminating) timing interferences. In particular, all instructions can be dumped into the corresponding trace buffers in the cores, and sent immediately to a separate DRAM region through a dedicated memory controller using the Debug Support Unit (DSU) interface, thus not interfering with the AMBA bus used for L2 cache and memory accesses. Once instructions information (including instruction and data addresses) is placed in that DRAM memory region, an ad-hoc trace controller reads those traces and sends them asynchronously through the Ethernet interface to the host. In this setup, execution time can only be interfered if the DRAM region is filled in before the trace controller can send the data to the host. However, this allows plenty of room for collecting large traces without creating any interference since the DRAM region is typically large (e.g. 512MB in our experiments), thus allowing tracing full programs or large regions of them.

5 Evaluation

We present LEOPARD evaluation results with benchmarks and a representative space application from Airbus Defense and Space. Hardware overheads numbers for the baseline and LEOPARD configuration are also provided.

Note that the goal of this paper is not to provide a fair comparison of different timing analysis techniques that has been shown a complex task [1]. The the effort required to tailor a static timing analysis tool to the our target platform (e.g. including TLBs, unified caches and shared buses) and the space case study (e.g. to provide flow-facts) is significant. Our goal instead is showing that high-quality performance guarantees can be achieved with relatively small hardware overhead for our relatively complex processor with little impact on average performance and without increasing analysis cost w.r.t. simpler architectures.

5.1 Methodology

MBPTA application. We use MBPTA to derive WCET estimates [47]. The number of measurements of the unit of analysis (UoA) required to apply MBPTA (1) has to guarantee that the events with highest impact are effectively captured in the measurements and (2) has to allow the correct application of EVT [27]. For the experiments conducted in this paper we base on [31, 11].

The common practice for MBTA approaches is to collect end-to-end execution time measurements of the UoA when it is fed with a set of user defined input vectors. An inherent limitation of this approach is that the resulting WCET bounds are only valid for the set of paths for which observations are collected. To overcome this issue, timing analysis techniques, like Extended Path Coverage (EPC) [50], derive upper bounds of the probabilistic execution time of the complete program under analysis even when the user-provided input vectors do not exercise the worst-case path. However, despite that LEOPARD allows deriving pWCET estimates with EPC, in this paper we stick to single-path case results rather than on the software application of EPC since our focus is on the hardware platform.

In order to ensure that in each run the UoA is analyzed under the same *initial conditions*, which upper bound those during operation, we empty the caches right before UoA execution. To do this, we configure the scheduling to ensure that the UoA starts its execution right after a time partition switch and we configure the hypervisor to flush the caches on that time partition switch. When experiments are executed in bare-metal, i.e. without operating system support, we perform this process manually.

For multicore evaluation we use two setups. In the first one, LEOPARD is instructed not to provide time-composable contention bounds (referred to as non-TC mode), see Section 2.2. The second one does force time-composable contention bounds (referred to as TC mode).

5.2 Complexity of LEOPARD features

The different modifications required to achieve MBPTA compliance have been shown to involve little hardware overheads. In terms of complexity, for the FPGA implementation the maximum operating frequency on an Altera DE4 board (100MHz) has been preserved despite the latency overhead introduced by the random placement implementation in the access to the caches. For LEOPARD the *RM* placement [21] effectively minimizes this overhead and facilitates keeping the maximum operating frequency. However, frequency preservation is not guaranteed for different technology libraries and/or different processors implementations, and such evaluation is out of the scope of this paper. Modifications did not have any effect on the most critical path of the different components except for cache memories. First level caches implementing random modulo [21] only increased their delay by a XOR gate to combine address and random seed bits. In the case of the L2 cache, hash-based random placement had a larger impact on critical path due to the higher complexity of its design (few XOR gates and a bit rotator) [26]. Still, impact was not enough to decrease the maximum operating frequency.

In terms of hardware resources occupancy, the baseline design occupied 70% of the resources in the FPGA, whereas LEOPARD occupies 72%, thus showing that all modifications required to achieve MBPTA-compliance and high-speed tracing incur very low overheads.

5.3 Evaluating LEOPARD features

Average performance. Our results show no performance degradation when LEOPARD features are deactivated on the modified RTL based prototype w.r.t. the original unmodified RTL design not containing LEOPARD modifications. Hence LEOPARD does not affect the average performance of applications requiring no performance guarantees.

FPU. To test the effectiveness of the worst-latency FPU unit we have designed four micro-benchmarks (DIVshort, DIVlong, SQRTshort, SQRTlong) that respectively execute short and long latency divisions and square roots. They are executed on two configurations: one with variable-latency FPU operations (labelled COTS) and another with fixed-latency FPU operations (labelled LEOPARD). Execution times when running these benchmarks in isolation (non TC mode) are shown in Figure 6. As shown, under the original setup DIVshort executes faster than DIVlong. Analogously, SQRTshort executes faster than SQRTlong. Therefore, it can be concluded that input values operated impact execution time in the original setup. Conversely, DIVshort and DIVlong have exactly the same execution time on top of the LEOPARD setup. Such execution time matches the execution time of DIVlong on top of the original setup. Results for SQRTshort and SQRTlong show exactly the same behaviour. In fact, the execution time variation between DIVshort and DIVlong corresponds exactly to the number of FDIVD instructions multiplied by 3, which is the difference between short (15 cycles) and long (18 cycles) latencies. The situation for SQRTshort and SQRTlong is analogous. Overall, this experiment validates that modifications in the FPU remove jitter due to the input data operated.

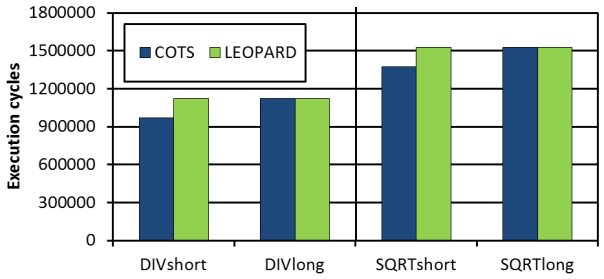


Figure 6 Tests to assess the control of the FPU jitter.

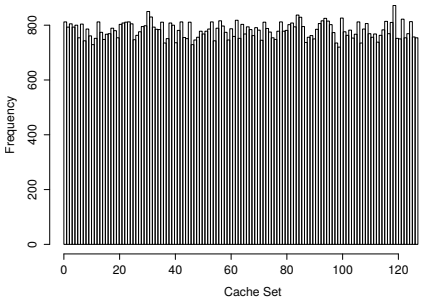


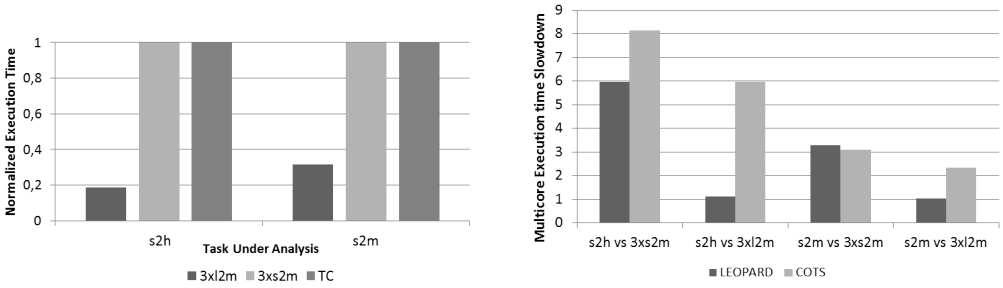
Figure 7 Randomized cache tests.

Cache. To evaluate randomized cache designs in LEOPARD, we test the ability of the cache placement function to cover different memory layouts and thus, capture different cache conflicts associated with the different arrangement of objects in memory. Figure 7 shows how an arbitrary address is mapped to different cache sets across 10,000 different runs in the LEOPARD IL1 (128 sets). Hence, the random placement function maps a given address uniformly across the different cache sets which benefits the application of MBPTA [26].

Multicore contention. To test the LEOPARD bus arbitration we assess whether (1) the measured multicore contention with LEOPARD under TC mode effectively upperbounds the highest contention scenario we can create at software level; and (2) whether the slowdown due to bus contention with LEOPARD CBA design is lower than the one of the COTS platform. To do so, we have developed several micro-benchmarks consisting in loads that always miss or hit in the L2 (respectively called *l2m* and *l2h*); and stores that always miss or hit in the l2 (respectively called *s2m* and *s2h*).

Figure 8 (a) shows the results we obtain when running benchmarks *s2h* and *s2m* (the UoA) under three scenarios. Under the former two, the UoA runs in non-TC mode, against 3 copies of *l2m* and *s2m*, respectively. In the third setup *s2h* and *s2m* run under TC mode. We observe that measurements in TC mode upperbound those of the former two scenarios. Interestingly, the second scenario is the worst case we can create in software with each request of the UoA suffering the delay of two requests due to a dirty miss eviction (56 cycles): note that *l2m* generates non-dirty misses and *s2m* dirty misses. We observe the the worst case generated by software matches the time-composable bounds generated by LEOPARD.

Figure 8 (b) shows the benefits brought by the CBA to handle variable latency requests. To that end we show the maximum slowdown due to multicore contention for different benchmarks on the baseline platform and on LEOPARD with TC-mode. For example *s2h*.vs.*3xs2m* represents the case where the UoA is the *s2h* benchmark when it runs against 3 cores executing the *s2m* benchmark. As shown, in all cases except one, LEOPARD significantly reduces the bus contention. For instance, for the $s2h$.vs.$3xl2m$ case, contention is reduced from 5.96X to 1.12X. However, for the $s2h$.vs.$3xl2m$ case, LEOPARD contention is slightly worse. The reason is that despite CBA is effective to control the interference, it also has a side effect on the UoA preventing it to access the bus when the budget is exhausted. However, we have observed that this only occurs in very extreme cases like in this one, where all instructions from the UoA perform two memory operations, one to write the current data and one to evict data stored in the L2. In this case the budget of the UoA is exhausted in every bus access and the CBA scheme provides no benefit.



(a) Contention upperbound **(b) Slowdown**

Figure 8 Results of the experiments carried out to assess LEOPARD's on-chip bus design.

Tracing. Our results show that the maximum achievable *I-point* frequency in terms of cycles between *I-point*s is 24 cycles if just 1 core is traced, 51 for 2 cores, 75 for 3 cores and 100 cycles for 4 cores. For the values in which more than one processor is traced, the value corresponds to traces from each available processor. This test has been accomplished by using an infinite loop in which *I-point*s arrive at a constant frequency. This might not be exactly representative of a real application since there might be different phases in a real application and if there is enough space between phases to empty the buffers it might be possible to trace more frequent *I-point*s for short periods of time. But if the application has loops with very high number of iterations which contain *I-point*s, then those results are representative as the *I-point*s will arrive at a constant frequency. As expected, with the increasing number of cores the rate at which *I-point*s can be traced reduces linearly. This can be explained with the fact that the amount of data that needs to be read from the external DRAM increases linearly with the increasing number of cores.

Since the tracing scheme with external DRAM depends on shared resources (Ethernet link speed, AHB bus, and the host) there is a limitation on the frequency of *I-point*s that can be traced real-time. Our evaluations showed that the main bottleneck in the trace bandwidth are GRMON reads and partly the AHB BUS architecture. Increasing the Ethernet buffer size from 4kB to 64kB allowed to increase the read speed from GRMON significantly which resulted in increased bandwidth. But since the FIFOs from the processors compete with the Ethernet reads on the same bus, the read speed during tracing is slower compared to reading the DRAM while there are no traces written to the DRAM. This reduces the potential maximum bandwidth that can be achieved with the available Ethernet link speed. For example, while it is possible to reach on average 550Mbit/s read speed while reading 64MB of data from the external DRAM through GRMON, the read speed reduces to 470Mbit/s when processor(s) create very frequent *I-point*s. Also a more dedicated software for large data reads on the host side can improve the bandwidth.

5.4 Space Case-study

The space case study we have used consists of a payload application with a high criticality application, a control loop applying deformations on mirrors, and a low criticality application, responsible of processing images coming from 3 detectors and that requires performance. To fulfill mixed-criticality isolation requirements, applications from this case study run on top of PikeOS hypervisor [44]. The goal of the application is to get better images on the detectors by applying deformations on the mirrors. The output of the image processing is used as an input to compute the displacement of the mirrors; however the control loop of the mirrors

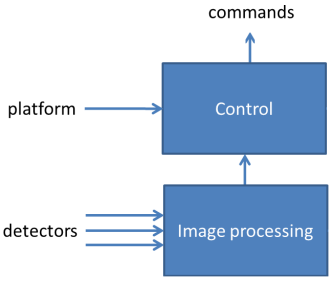


Figure 9 Sketch of the Space App.

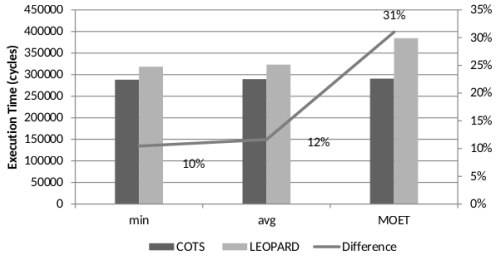


Figure 10 Single-core measured execution times.

can work for some cycles without the input value (with degraded performance). A general view of the application is given Figure 9.

- Image processing partition. The image processing application consists is 3 different tasks each of them pinned to a core. The input data of the image processing partition is preloaded in the memory of the platform to avoid I/O interference during the experiments. This simplifies the observations. The image processing application runs at a higher frequency than the control application. It generates 10 values that are merged by the control partition. The control loop needs to compute the voltage to apply to the mirror motors in order to compensate thermal effects on the main mirror. The image processing has no real-time deadline and thus does not need to be periodic.
- Control partition. The control partition is the destination of the queuing port shared with the image processing partition and corresponds to the UoA of this case study. The control partition uses 10 values to compute the new matrix of voltages to apply to the mirrors. In the case study, there is no mirror to command but these output values are logged to verify the functional behaviour of the application. In the absence of values in the queue or if less than 10 values have been computed by the image processing partition, the control partition reuses old values. Thus, it is tolerant to errors coming from the image processing partition. The control partition runs in isolation in one core.

We first compare single-core performance of the COTS platform and the LEOPARD processor that must not be confused with average performance, for which LEOPARD provides no degradation. In the measurements for the COTS platform, see Figure 10, the impact of SoJ (e.g. FPU and cache jitter) are not factored in the measurements, while the measurements obtained on top of the LEOPARD design expose this jitter to the measurements. For the space case study used in this paper, LEOPARD single-core measurements show that the impact of SoJ on the control application is 12% on average, 10% for the minimum observed execution time and 31% for the maximum observed execution time (MOET).

Figure 11 compares the execution time results of the control application (the highly critical) when is run in three different configurations: (1) isolation, (2) with the image processing application, referred as mixed criticality or MC scenario, and (3) time-composable estimates that are derived in the worst contention scenario. We observe that when the high critical control application runs with the 3-core low critical image processing application, its execution time increases w.r.t. its execution time in isolation (first column). Still measurements collected in TC mode effectively upperbound them (second column).

In Figure 12, where all values are normalized to the MOET of the COTS processor when the control application runs in isolation, we observe that the MOET for LEOPARD with the 3-core image processing application (LEOPARD MC) is 1.87. This value is smaller than that

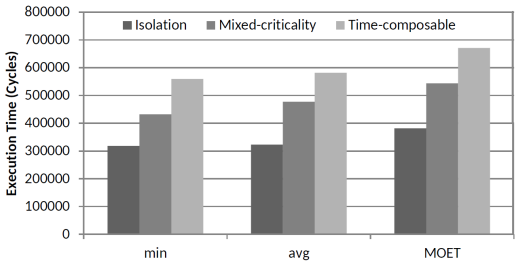


Figure 11 Multicore measured execution times.

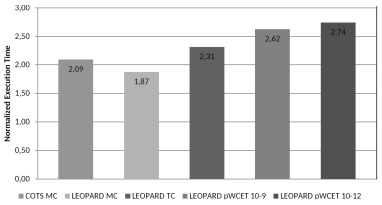


Figure 12 Mixed-criticality execution times and pWCET estimates.

for the same experiment on the COTS platform (COTS MC) that is 2.09. This confirms that the credit-based arbitration allows increasing the performance of the UoA in the presence of inter-core (contention) interference.

The MOET in the highest contention scenario (LEOPARD TC mode) is 2.31x showing that is close to the actual observed values (1.87). These TC values are used as input to MBPTA that results in tight pWCET estimates for exceedance probabilities (per activation of the image processing algorithm) at 10^{-9} and 10^{-12} (considered relevant in previous case studies [47]). Reported pWCET estimate results are in the range 2.6x-2.7x, which are reasonable bounds for a 4-core architecture.

6 Related Work

The timing requirements across different domains (or different systems in the same domain) change and so do the processor designs that have been proposed to support them. In this section we review several research efforts aiming at achieving time-predictability with different processor implementations. Note that we have opted to provide system-level descriptions of related works rather than focusing on related work for each technique, which we have presented in previous sections.

Patmos [42] is a statically scheduled, dual-issue RISC processor specifically designed to facilitate deriving tight WCET estimates with static timing analysis. As both, static timing analysis (STA) and MBTA, are used by industry to derive WCET estimates, having processor designs capturing STA requirements is of interest for both academia and industry. Unlike Patmos that favors WCET over average-performance, i.e. the latter is considered a secondary goal, LEOPARD is designed to incur minimum impact on average performance. LEOPARD also aims at injecting minimum changes on the baseline platform rather than to make a design specifically oriented to WCET.

CompSOC [19] platform offers a virtual platform (resource partition) per application so that applications cannot cause any interference on other co-running applications. LEOPARD, instead, focuses on applications domains for which having bounded interference (rather than no interference) suffices to satisfy timing isolation requirements [37]. In LEOPARD, instead of preventing any interaction among applications – so that one application is forbidden to generate even a single-cycle delay on others – applications are allowed to interfere each other as long as that interference can be bounded.

The FlexPRET [51] processor balances WCET and average performance by designing an specific simultaneous multi-threaded (SMT) architecture that simultaneously executes both 'hard' real-time tasks and 'soft' real-time tasks. Hardware changes (e.g. avoiding data hazards via specific forwarding paths) and a smart instruction scheduler make the execution of soft

real-time tasks transparent, so not affecting hard real-time tasks execution. This is similar to parMERASA core architecture [35] that also aims at transparent execution of low-critical SMT threads by for instance preventing non-preemtable multi-cycle operations. FlexPRET also makes other important hardware changes like adding new timing instructions to the baseline RISC-V ISA, while LEOPARD sticks to SPARC v8+ ISA. Further, in PRET caches are replaced by scratchpads to provide more predictability. Overall, different approaches are pursued by FlexPRET that introduce non-negligible real-time specific hardware components to support WCET analysis. Conversely, LEOPARD focuses on MBTA requirements and aims at introducing minimum changes on the baseline architecture.

Other recent approaches have focused on understanding the analysability properties of complex COTS multicore processors. A summary of some of these works can be found in [15] and [37]. Some works suggest a separate analysis approach [41, 12]: they propose a separate analysis for contention and, frequently, rely on splitting tasks into sub-tasks or phases so that worst-case alignment in (typically) TDMA-based arbiters can be reasonably computed. In [24] authors derive a model to handle multicore contention based on resource stressing kernels and performance monitoring counters for deriving both, the access latencies to resources and the impact that tasks can suffer in the access to hardware shared resources. For static timing analysis, the work in [34] proposes an approach to factor in contention in single-core execution times by considering the interference (number of parallel contender requests) the requests of the task under analysis need to be arbitrated against, instead of considering the worst latency. Also, an enforcing mechanism (safety net) is put in place to ensure that tasks do not try to use more than its assigned budget in terms of request count.

Another software approach for time predictability is the $WCET(m)$ [29]. It creates, via OS support, resource partitions: private cache partitions for the last level cache; memory bandwidth partitioning by controlling and enforcing a maximum number of access counts considering worst-case memory latency L_{max}; and bank partitioning to further reduce memory latency. As a result, each core receives a resource partition and the WCET of each task depends on the number of active cores, m. Despite each task receives a hardware partition of resources, those resources can be jittery such as multi-level caches and FPU, and handling this intra-partition jitter complicates deriving WCET estimates as summarized in Section 2. LEOPARD hardware designs simplify providing evidence that measurements taken for a given task (under a resource partition) are representative and capture the impact of the jittery resources. LEOPARD support for multicore, e.g. the on-chip bus not covered in [29], contention has been shown to improve the baseline ones and further is compatible with the WCET(m) approach.

Overall, a commonality in these software models is that they assume that execution times are representative. For example, these works consider the task under analysis suffers no jitter either in time or access counts due to memory mapping (cache layout), or the particular values operated in the floating-point unit. However, as we have shown in this paper, both factors can create jitter. This assumption of representative measurements also holds in the typical worst-case analysis approach [39]. LEOPARD, by deploying deterministic and probabilistic jitter bounding, helps all these software approaches to increase the representativeness on the observations without requiring additional testing efforts.

7 Conclusions

We have shown the design of hardware support for easing MBPTA and its implementation resulting in an-RTL based prototype. While hardware concepts are simple and their im-

plementation has low or moderate complexity in a simulator, their real integration with other hardware elements such as virtual/physical address management and cache flushing has been challenging. Moreover, during their integration and test we have discovered performance limitations that have been addressed proposing, implementing and integrating more efficient designs. Also, tracing capabilities needed by the timing analysis tools were not powerful enough in the baseline FPGA prototype. The net results is that we have successfully implemented all features required to make the FPGA prototype be MBPTA compliant, i.e. controlling on-core and on-chip sources of jitter. We have further designed and integrated a new high-speed tracing feature able to dump traces through the Ethernet port at high speed and without affecting normal execution of programs. Overall, LEOPARD benefits hardware designers to better understand how MBTA compliance can be achieved with low overhead and no impact on average performance while allowing to reach a growing market with time predictability needs. We also expect LEOPARD to motivate embedded system practitioners to push for those small changes to be implemented in other designs, increasing the effectiveness of existing software approaches to control jitter.

References

1 Jaume Abella, Damien Hardy, Isabelle Puaut, Eduardo Quiñones, and Francisco J. Cazorla. On the comparison of deterministic and probabilistic WCET estimation techniques. In *26th Euromicro Conference on Real-Time Systems, ECRTS 2014, Madrid, Spain, July 8-11, 2014*, 2014. doi:10.1109/ECRTS.2014.16.

2 Jaume Abella, Carles Hernández, Eduardo Quiñones, Francisco J. Cazorla, Philippa Ryan Conmy, Mikel Azkarate-askasua, Jon Perez, 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*, 2015. doi:10.1109/SIES.2015.7185039.

3 Irune Agirre, Mikel Azkarate-askasua, Carles Hernández, Jaume Abella, Jon Perez, Tullio Vardanega, and Francisco J. Cazorla. IEC-61508 SIL 3 compliant pseudo-random number generators for probabilistic timing analysis. In *2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, August 26-28, 2015*, 2015. doi:10.1109/DSD.2015.26.

4 Benny Akesson, Andreas Hansson, and Kees Goossens. Composable resource sharing based on latency-rate servers. In *12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools, DSD 2009, 27-29 August 2009, Patras, Greece*, 2009. doi:10.1109/DSD.2009.167.

5 Benny Akesson, Liesbeth Steffens, and Kees Goossens. Efficient service allocation in hardware using credit-controlled static-priority arbitration. In *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2009, Beijing, China, 24-26 August 2009*, 2009. doi:10.1109/RTCSA.2009.13.

6 Peter Alfke. *Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators*. Xilinx, 1996.

7 ARM. Amba bus specification. URL: http://www.arm.com/products/system-ip/amba/amba-open-specifications.php.

8 Francisco J. Cazorla, Eduardo Quiñones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, Emery D. Berger, Jaume Abella, Franck Wartel, Michael Houston, Luca Santinelli, Leonidas Kosmidis, Code Lo, and Dorin Maxim. PROARTIS: probabilistically analyzable real-time systems. *ACM Trans. Embedded Comput. Syst.*, 12(2s), 2013. doi:10.1145/2465787.2465796.

9 Certification Authorities Software Team. *CAST-32A Multi-core Processors*, 2016.

10 Cobham Gaisler. *Quad Core LEON4 SPARC V8 Processor – GR740-UM-DS-D1 – Data Sheet and Users Manual, 2015*.

11 Liliana Cucu-Grosjean, Luca Santinelli, Michael Houston, Code Lo, Tullio Vardanega, Leonidas Kosmidis, Jaume Abella, Enrico Mezzetti, Eduardo Quiñones, and Francisco J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *24th Euromicro Conference on Real-Time Systems, ECRTS 2012, Pisa, Italy, July 11-13, 2012*, 2012. doi:10.1109/ECRTS.2012.31.

12 Dakshina Dasari, Vincent Nélis, and Benny Akesson. A framework for memory contention analysis in multi-core platforms. *Real-Time Systems*, 52(3), 2016. doi:10.1007/s11241-015-9229-9.

13 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 *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, 2016. doi:10.4230/OASIcs.WCET.2016.1.

14 Boris Dreyer, Christian Hochberger, Alexander Lange, Simon Wegener, and Alexander Weiss. Continuous non-intrusive hybrid WCET estimation using waypoint graphs. In *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, 2016. doi:10.4230/OASIcs.WCET.2016.4.

15 Gabriel Fernandez, Jaume Abella, Eduardo Quiñones, Christine Rochange, Tullio Vardanega, and Francisco J. Cazorla. Contention in multicore hardware shared resources: Understanding of the state of the art. In *14th International Workshop on Worst-Case Execution Time Analysis, WCET 2014, July 8, 2014, Ulm, Germany*, 2014. doi:10.4230/OASIcs.WCET.2014.31.

16 E. Francis. Autonomous cars: no longer just science fiction. *Automotive Industries*, 2014.

17 Cobham Gaisler. *Leon3 Processor*. http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=13&Itemid=53.

18 Sylvain Girbal, Miquel Moretó, Arnaud Grasset, Jaume Abella, Eduardo Quiñones, Francisco J. Cazorla, and Sami Yehia. On the convergence of mainstream and mission-critical markets. In *The 50th Annual Design Automation Conference 2013, DAC'13, Austin, TX, USA, May 29 – June 07, 2013*, 2013. doi:10.1145/2463209.2488962.

19 Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. Compsoc: A template for composable and predictable multi-processor system on chips. *ACM Trans. Design Autom. Electr. Syst.*, 14(1), 2009. doi:10.1145/1455229.1455231.

20 Carles Hernandez, Jaume Abella, Francisco J. Cazorla, Jan Andersson, and Andrea Gianarro. Towards making a LEON3 multicore compatible with probabilistic timing analysis. In *Proceedings of the 20th Data Systems In Aerospace Conference, DASIA, 2015, Barcelona, Spain*, 2015.

21 Carles Hernández, Jaume Abella, Andrea Gianarro, Jan Andersson, and Francisco J. Cazorla. Random modulo: a new processor cache design for real-time critical systems. In *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*, 2016. doi:10.1145/2897937.2898076.

22 HiPEAC. hiPEAC vision, 2017. https://www.hipeac.net/publications/vision/.

23 Infineon. AURIX – TriCore datasheet. highly integrated and performance optimized 32-bit microcontrollers for automotive and industrial applications, 2012.

24 Javier Jalle, Mikel Fernandez, Jaume Abella, Jan Andersson, Mathieu Patte, Luca Fossati, Marco Zulianello, and Francisco Cazorla. Bounding Resource Contention Interference in the Next-Generation Microprocessor (NGMP). In *8th ERTS*, 2016.

25 Javier Jalle, Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J. Cazorla. Bus designs for time-probabilistic multicore processors. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, 2014. doi:10.7873/DATE.2014.063.

26 Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J. Cazorla. A cache design for probabilistically analysable real-time systems. In *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, 2013. doi:10.7873/DATE.2013.116.

27 Samuel Kotz and Saralees Nadarajah. *Extreme value distributions: theory and applications*. World Scientific, 2000.

28 Stephen Law and Iain Bate. Achieving appropriate test coverage for reliable measurement-based timing analysis. In *28th Euromicro Conference on Real-Time Systems, ECRTS 2016, Toulouse, France, July 5-8, 2016*, 2016. doi:10.1109/ECRTS.2016.21.

29 Renato Mancuso, Rodolfo Pellizzoni, Marco Caccamo, Lui Sha, and Heechul Yun. Wcet(m) estimation in multi-core systems using single core equivalence. In *27th Euromicro Conference on Real-Time Systems, ECRTS 2015, Lund, Sweden, July 8-10, 2015*, 2015. doi:10.1109/ECRTS.2015.23.

30 Enrico Mezzetti and Tullio Vardanega. A rapid cache-aware procedure positioning optimization to favor incremental development. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9-11, 2013*, 2013. doi:10.1109/RTAS.2013.6531084.

31 Suzana Milutinovic, Jaume Abella, and Francisco J. Cazorla. Modelling probabilistic cache representativeness in the presence of arbitrary access patterns. In *19th IEEE International Symposium on Real-Time Distributed Computing, ISORC 2016, York, United Kingdom, May 17-20, 2016*, 2016. doi:10.1109/ISORC.2016.28.

32 Kevin Mueller, Georg Sigl, Benoit Triquet, and Michael Paulitsch. On MILS I/O sharing targeting avionic systems. In *2014 Tenth European Dependable Computing Conference, Newcastle, United Kingdom, May 13-16, 2014*, 2014. doi:10.1109/EDCC.2014.35.

33 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*, 2014. doi:10.1109/ECRTS.2014.20.

34 Jan Nowotsch, Michael Paulitsch, Arne Henrichsen, Werner Pongratz, and Andreas Schacht. Monitoring and WCET analysis in COTS multi-core-soc-based mixed-criticality systems. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, 2014. doi:10.7873/DATE.2014.080.

35 Marco Paolieri, Jörg Mische, Stefan Metzlaff, Mike Gerdes, Eduardo Quiñones, Sascha Uhrig, Theo Ungerer, and Francisco J. Cazorla. A hard real-time capable multi-core SMT processor. *ACM Trans. Embedded Comput. Syst.*, 12(3):79:1–79:26, 2013. doi:10.1145/2442116.2442129.

36 Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, 2009. doi:10.1145/1555754.1555764.

37 Michael Paulitsch, Oscar Medina Duarte, Hassen Karray, Kevin Mueller, Daniel Münch, and Jan Nowotsch. Mixed-criticality embedded systems – A balance ensuring partitioning and performance. In *2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, August 26-28, 2015*, 2015. doi:10.1109/DSD.2015.100.

38 PROXIMA. Probabilistic real-time control of mixed-criticality multicore and manycore systems, oct 2014. URL: http://www.proxima-project.eu/.

39 Sophie Quinton, Torsten T. Bone, Julien Hennig, Moritz Neukirchner, Mircea Negrean, and Rolf Ernst. Typical worst case response-time analysis and its use in automotive network design. In *The 51st Annual Design Automation Conference 2014, DAC'14, San Francisco, CA, USA, June 1-5, 2014*, 2014. doi:10.1145/2593069.2602977.

40 Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, and San Vo. A statistical test suite for the validation of random number generators and pseudo random number generators for cryptographic applications. Special publication 800-22rev1a, US National Institute of Standards and Technology (NIST), 2010.

41 Simon Schliecker, Mircea Negrean, Gabriela Nicolescu, Pierre G. Paulin, and Rolf Ernst. Reliable performance analysis of a multicore multithreaded system-on-chip. In *Proceedings of the 6th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2008, Atlanta, GA, USA, October 19-24, 2008*, 2008. doi:10.1145/1450135.1450172.

42 Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, and Christian W. Probst. Towards a time-predictable dual-issue microprocessor: The patmos approach. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems, DATE Workshop PPES 2011, March 18, 2011, Grenoble, France.*, 2011. doi:10.4230/OASIcs.PPES.2011.11.

43 Mladen Slijepcevic, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. Design and implementation of a fair credit-based bandwidth sharing scheme for buses. In *DATE conference*, 2017.

44 Sysgo. *PikeOS Safe and Secure Virtualization*, 2010.

45 Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*, 2016. doi:10.1109/RTAS.2016.7461361.

46 Augusto Vega, Chung-Ching Lin, Karthik Swaminathan, Alper Buyuktosunoglu, Sharathchandra Pankanti, and Pradip Bose. Resilient, uav-embedded real-time computing. In *33rd IEEE International Conference on Computer Design, ICCD 2015, New York City, NY, USA, October 18-21, 2015*, 2015. doi:10.1109/ICCD.2015.7357189.

47 Franck Wartel, Leonidas Kosmidis, Code Lo, Benoit Triquet, Eduardo Quiñones, Jaume Abella, Adriana Gogonel, Andrea Baldovin, Enrico Mezzetti, Liliana Cucu, Tullio Vardanega, and Francisco J. Cazorla. Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study. In *8th IEEE International Symposium on Industrial Embedded Systems, SIES 2013, Porto, Portugal, June 19-21, 2013*, 2013. doi:10.1109/SIES.2013.6601497.

48 Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter P. Puschner. Measurement-based timing analysis. In *Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008, Porto Sani, Greece, October 13-15, 2008. Proceedings*, 2008. doi:10.1007/978-3-540-88479-8_30.

49 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), 2008. doi:10.1145/1347375.1347389.

50 Marco Ziccardi, Enrico Mezzetti, Tullio Vardanega, Jaume Abella, and Francisco Javier Cazorla. EPC: extended path coverage for measurement-based probabilistic timing analysis. In *2015 IEEE Real-Time Systems Symposium, RTSS 2015, San Antonio, Texas, USA, December 1-4, 2015*, 2015. doi:10.1109/RTSS.2015.39.

51 Michael Zimmer, David Broman, Chris Shaver, and Edward A. Lee. Flexpret: A processor platform for mixed-criticality systems. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014*, 2014. doi:10.1109/RTAS.2014.6925994.

Budgeting Under-Specified Tasks for Weakly-Hard Real-Time Systems*

Zain A. H. Hammadeh[1], Sophie Quinton[2], Marco Panunzio[3], Rafik Henia[4], Laurent Rioux[5], and Rolf Ernst[6]

1 **Technische Universität Braunschweig, Braunschweig, Germany**
hammadeh@ida.ing.tu-bs.de
2 **Inria Grenoble – Rhône-Alpes, Grenoble, France**
sophie.quinton@inria.fr
3 **Thales Alenia Space, France**
marco.panunzio@thalesaleniaspace.com
4 **Thales Research & Technology, France**
rafik.henia@thalesgroup.com
5 **Thales Research & Technology, France**
laurent.rioux@thalesgroup.com
6 **Technische Universität Braunschweig, Braunschweig, Germany**
ernst@ida.ing.tu-bs.de

Abstract

In this paper, we present an extension of slack analysis for budgeting in the design of weakly-hard real-time systems. During design, it often happens that some parts of a task set are fully specified while other parameters, e.g. regarding recovery or monitoring tasks, will be available only much later. In such cases, slack analysis can help anticipate how these missing parameters can influence the behavior of the whole system so that a resource budget can be allocated to them. It is, however, sufficient in many application contexts to budget these tasks in order to preserve weakly-hard rather than hard guarantees. We thus present an extension of slack analysis for deriving task budgets for systems with hard and weakly-hard requirements. This work is motivated by and validated on a realistic case study inspired by industrial practice.

1998 ACM Subject Classification B.8.2 Performance Analysis and Design Aids

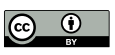
Keywords and phrases real-time, weakly-hard, slack analysis, execution budget, fixed priority

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.17

1 Introduction

In the design of real-time systems, it is not uncommon for some parts of a task set to be fully specified while other parameters, e.g. regarding recovery or monitoring tasks, will be available only much later. In such cases, slack analysis can help anticipate how these missing parameters can influence the behavior of the whole system so that a resource budget can be allocated to them. It is, however, sufficient in many application contexts to budget these tasks so as to preserve *weakly-hard* rather than hard guarantees. Such guarantees allow for a bounded number of consecutive deadline misses (*"at most m out of k deadlines may be*

* This work has been partially funded by the German Research Foundation (DFG) as part of the project "TypicalCPA" under the contract number TWCA ER168/30-1.



missed"). We thus present an extension of slack analysis for budgeting under-specified tasks for systems with hard and weakly-hard requirements.

The four main contributions of this paper are:

- an extension of slack analysis [11] to compute the maximum slack which guarantees that no more than m deadline misses out of k consecutive executions can happen;
- an execution time budget for under-specified tasks based on the multiframe task model [15] where we consider that each under-specified task has two execution times: a long one and short one;
- a methodology that explains why and how this method should be used safely in the design of systems that have hard and weakly-hard requirements;
- a case study dealing with satellite on-board software which shows the practical usefulness of weakly-hard constraints and how to guarantee them.

This paper is organized as follows. Section 2 introduces the application context of this paper, that is a satellite on-board software system. Section 3 then explains the timing verification problem that we are facing and why the proposed analysis is the appropriate solution to address it. Section 4 provides the preliminaries on response-time analysis which are needed to introduce our work. These preliminaries include principles of standard worst-case response-time analysis, typical worst-case analysis and slack analysis. Section 5 shows how to budget the under-specified tasks to satisfy hard real-time constraints. Our major contribution is in Section 6 where we present an extension of slack analysis and our general approach for budgeting based on the multiframe task model. We summarize the methodology that we propose in Section 7, present our experimental results in Section 8 and discuss related work in Section 9. Section 10 concludes.

2 Motivational Example

In this section we introduce the case study which motivates the work presented in this paper.

2.1 State of the practice for the timing analysis of satellite software

A satellite is made of two major parts: the platform and the payload. The payload realizes the main satellite mission, and comprises scientific instruments, telescopes or telecommunication antennas, according to the mission of the satellite. The payload is typically characterized by high computation requirements but in the general case its software is considered at best firm or soft real-time.

The platform is the service module that governs the satellite and ensures the execution of the mission. The platform on-board software (OBSW) implements all major functions of the satellite: e.g., the Attitude and Orbit Control System (AOCS), the Thermal Control System (TCS), mode management, Data Handling System (DHS).

A subset of those OBSW functions are characterized by hard real-time requirements. For example, sending thruster commands at the wrong moment during an attitude modification or an orbital maneuver (e.g., the main orbit insertion of a deep-space orbiter) may lead to mission failure.

In contrast, some tasks executing some less critical functions, may occasionally miss deadlines without dreadful consequences on the mission, and at most some performance degradation. One example is the AOCS functions itself, where sensor acquisition and processing are somehow robust to occasional deadline misses because of the intrinsic robustness of the implemented control laws.

The OBSW is however traditionally designed, analyzed and implemented with techniques typical of safety-critical, hard real-time systems. This implies that all tasks defined for the OBSW are considered as hard real-time and treated as such in the schedulability analysis used to confirm the system feasibility. The analysis is performed using representative worst-case operational scenarios. The reason for this choice is twofold:

1. It is much easier to prove to clients that the system is schedulable and fulfills the mission goals by treating all tasks as hard real-time, with a design process and analysis equations consolidated along several years, and without admitting exceptions on the treatment of task deadlines.
2. The OBSW development team does not know completely the possible consequences of deadline misses from the point of view of performance degradation or function losses, as such knowledge requires deep analysis at system / avionics level. It is therefore not obvious to understand if deadline misses are admissible in the overall mission context.

2.2 System model and use case

Current satellite OBSW is typically executed on a single-core processor, and using a Fixed-Priority Preemptive scheduling policy (FPP).

Table 1 shows a representative task set and the real-time attributes of each task. The attributes are representative of a high-load scenario for the OBSW in a mission operational mode. Each task τ_i in the system is characterized by its:

- priority index π_i; for simplicity of notation, we assume that tasks are given in order of their static priority, i.e., τ_j has higher priority than τ_i for every $j < i$;
- type of task release pattern: periodic (P), possibly with static offset, software sporadic (S), hardware sporadic (HWS), i.e., triggered by an interrupt, background task;
- worst-case execution time C_i; this value is not based on static analysis but rather on the observed execution times;
- period or interarrival time T_i;
- offset φ_i if applicable;
- relative deadline D_i – all deadlines are constrained;
- maximum blocking time b_i; execution in mutual exclusion is enforced with semaphores or protected objects (monitors) for which the maximum blocking can be bounded.

Note that some tasks specified as sporadic have in fact a pseudo-periodic behavior.

Table 1 includes two different kinds of tasks:

(i) *nominal tasks*: tasks that are active and executed in the represented operational scenario;

(ii) *recovery tasks*: tasks that are involved in asynchronous fault handling or recovery activities and are triggered only on given fault / error occurrences. They are marked as gray in the table.

Among the nominal tasks, some have real-time constraints that we will consider as hard real-time; others can be considered as weakly-hard real-time, as they can withstand occasional deadline misses without significant system-level consequences.

3 Problem Statement

The specification of recovery tasks typically occurs in the latest development phases, and therefore their characteristics are not known until late in the development cycle. The execution of such recovery tasks may however perturb the execution of nominal tasks, leading to deadline misses which would potentially induce a degradation of the system performance.

Table 1 A task set representative of on-board software. π, C, T, φ, D and b denote respectively: priority, worst-case execution time, period/minimum distance, offset, deadline, blocking time. The time unit is ms.

Name	π	Type	C	T	φ	D	b
τ_1	1	HWS	0.56	15.625		15.625	0.1
τ_2	2	P	0.76	15.625		15.625	0.1
τ_3	3	P	15	125		31.25	0.1
τ_4	4	P	25.03	125	78.125	46.875	0.1
τ_5	5	P	7.5	62.5		62.5	0.1
τ_6	6	P	6.15	125		125	0.1
τ_7	7	P	1.2	125	93.750	125	0.1
τ_8	8	P	0.9	1000	500	500	0.1
τ_9	9	P	1.95	250		250	0.1
τ_{10}	10	S		10000		125	0.1
τ_{11}	11	S				125	0.1
τ_{12}	12	P	1.2	125		125	0.1
τ_{13}	13	P	5.15	250	46.875	203.125	0.1
τ_{14}	14	P	1.2	1000		500	0.1
τ_{15}	15	P	22.5	500		500	0.1
τ_{16}	16	P	3.5	250		250	0.1
τ_{17}	17	P	27	500		500	0.1
τ_{18}	18	HWS	1.5	1000		1000	0.1
τ_{19}	19	P	16	1000		1000	0.1
τ_{20}	20	P	19.1	1000		1000	0.1
τ_{21}	21	S				1000	0.1
τ_{22}	22	P	88.8	2000		2000	0.1
τ_{23}	23	P	2	32000		32000	0.1
τ_{24}	24	P	1	32000		32000	0.1
τ_{25}	25	P	1	1000		1000	0.1
τ_{26}	26	S	20	1000		1000	0.1
τ_{27}	27	S	40	2000		2000	0.1
τ_{28}	28	S	1.5	2000		2000	0.1
τ_{29}	29	S	1.5	2000		2000	0
τ_{30}	30	P	0.2	32000		32000	0

Configuring the timing attributes of the recovery tasks represents a challenging timing issue for the real-time architect. It is important to guarantee that the reconfiguration and recovery tasks can accomplish their functions, which are related to the safety of the spacecraft. At the same time, it is necessary to preserve a sound timing behavior for the nominal tasks.

Moreover, the timing behavior of the recovery tasks must be established and assessed as early as possible in the development, as in later phases the development of the rest of the software system approaches completion, with little freedom for significant modifications.

The reader should note that the problem statement regards finding a convenient method for assigning attributes and guarantees to such tasks, rather than establishing a complete fault tolerance strategy [19][14] for the on-board software and the satellite. The latter requires a much more global reasoning at system level and it is not in the scope of this paper. Those tasks would be just some among several mechanisms (hardware and software) that are devoted to the implementation of such global fault tolerance strategy for a given satellite, and the method we seek would simply concur to their definition in a convenient manner.

To solve this problem, there is a need for a timing verification method fulfilling two conditions:

(i) applicability at early design stages;

(ii) a guarantee on the provided upper bounds for the tasks' response times.

Worst-case response time analysis seems well adapted to solve the timing challenge mentioned above, since its applicability already starts with the early conceptual design phases and it provides formal proofs based on a mathematical model of the system timing behavior. These proofs allow calculating safe lower and upper bounds on the response times, thus guaranteeing corner-case coverage.

Classic worst-case response-time analysis would however not be able to take into account the weakly-hard nature of some tasks, and would just check that deadlines of those tasks are met in the worst case. This would lead to an under-estimation of the timing budget available for the recovery tasks (and therefore to ensure the safety functions), which could be delicate, especially in case of a system with a high CPU load.

A method that takes into account the weakly-hard nature of some tasks and can provide to the real-time architect means to perform tradeoffs on the budget to be assigned to the recovery tasks would be considered attractive in this context.

Let us now formulate our problem. We consider a single processing resource which schedules a task set $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ according to a Fixed Priority Preemptive (FPP) policy. Each task $\tau_i \in \mathcal{T}$ is modeled by its

- worst-case execution time C_i
- worst-case activation pattern η_i^+ (see below)
- priority π_i
- constrained deadline D_i

The tasks described in the motivating example of Section 2 are either periodic or sporadic. We will in this paper use the more general model of *arrival curves* to describe activation patterns, such that we can model sporadic tasks (in particular the recovery tasks that we want to budget) less conservatively than using a model based on the minimum interarrival time. We do not however handle offsets and conservatively assume that all periodic tasks can be activated at the same time. We leave to future the formal proof that offset analysis is compatible, as we conjecture, with the analysis presented in this paper. In contrast, blocking times are not mentioned in the rest of the paper for readability but they can easily be included in the analysis (and they are accounted for in the experiments).

▶ **Definition 1.** *Arrival curves* are functions $\eta_i^+, \eta_i^- : \mathbb{N} \to \mathbb{N}$ to model the possible activations of a task τ_i such that for any time window Δ, $\eta_i^+(\Delta)$ defines the maximum number of activations of τ_i that might occur within Δ, and , and η_i^- the minimum (in this paper we only use η_i^+). The pseudo-inverse of arrival curves, namely $\delta_i^-, \delta_i^+ : \mathbb{N} \to \mathbb{N}$, such that $\delta_i^-(k)$ (respectively $\delta_i^+(k)$) defines the minimum (respectively maximum) time that might pass between the first and the last activation in any sequence of k consecutive activations of τ_i.

In this context, the fact that deadlines are constrained translates into $D_i \leq \delta_i^-(2)$.

Our task set $\mathcal{T}$ is partitioned into *nominal tasks*, which are fully specified, and *recovery tasks*, for which only priorities and deadlines are known, such that we call these *under-specified tasks*. We denote by $\mathcal{N}$ the set of nominal tasks and $\mathcal{R}$ the set of under-specified tasks. Under-specified tasks are considered to be sporadic. Weakly-hard constraints are assumed to be given for nominal tasks.

Our problem is to provide a set of constraints on the execution times and the activation patterns of the tasks in $\mathcal{R}$ that is sufficient (and ideally necessary too) to guarantee (m, k)-schedulability of all tasks in $\mathcal{N}$, where a task is said to be (m, k)-*schedulable* if it cannot miss more than m deadlines out of a sequence of k consecutive executions.

4 Preliminaries on Response-Time Analysis

In this section we recall some state-of-the-art definitions and results on response-time analysis which we will use in the rest of the paper, based on the notations introduced at the end of our problem statement. We specifically present results related to worst-case response-time analysis, Typical Worst-Case Analysis (TWCA) and slack analysis.

Note that we suppose throughout this paper a representation of time based on natural numbers. This is reasonable since we consider single processor systems, which operate according to a unique, discrete clock.

4.1 Worst-case response-time analysis

A standard approach to establish schedulability of a system is to compute the worst-case response time of each task based on the concept of *busy window*. In this section we present results for the case where deadlines are arbitrary as we will need these later.

▶ **Definition 2.** A *level-i busy window* (originally called busy period in [10]) is a maximal time interval during which the resource still has activations of tasks of equal or higher priority than τ_i pending.

The longest such window, called *worst-case level-i busy window* and denoted BW_i, is built by assuming the occurrence of a so-called *critical instant*, where τ_i and higher-priority tasks are all activated at the same time, inducing maximum interference with τ_i. It is also assumed that all tasks are activated as early as possible after the critical instant, and that they always use their maximum execution time. The maximum level-i busy window stops at the first instant when no activation of τ_i or any higher priority task remains incomplete. It has been proven that the worst-case response time of task τ_i can be found in the longest level-i busy window.

▶ **Definition 3.** For a task τ_i and $q \geq 1$, the *multiple event busy time*, denoted $B_i(q)$, represents the maximum time it may take to process q activations of τ_i within a level-i busy

window starting with the first of these q activations.

$$B_i(q) = \min\{\Delta T \geq 0 \mid \Delta T = q \times C_i + \sum_{\tau_j \in hp(i)} \eta_j^+(\Delta T) \times C_j\} \tag{1}$$

where $hp(i)$ denotes the set of tasks with higher priority than τ_i (we assume that all tasks have distinct priorities).

The maximum number K_i of activations of τ_i in a level-i busy window is then

$$K_i = \min\{q \geq 1 \mid B_i(q) < \delta_i^-(q+1)\}$$

K_i is the smallest number such that the resource would be able to start processing the $(K_i + 1)$-th activation before this activation can occur according to δ_i, which implies an idle time. The worst-case level-i busy window can then be determined as $BW_i = B_i(K_i)$.
The response time of every activation of τ_i is bounded by

$$RT_i(q) = B_i(q) - \delta_i^-(q)\,.$$

The response time of τ_i is bounded by

▶ **Theorem 4.**

$$WCRT_i = \max_{1 \leq q \leq K_i} \{RT_i(q)\}\,. \tag{2}$$

We refer the reader to [21] for detailed explanations about the FPP response-time analysis.

4.2 Typical Worst-Case Analysis

Typical Worst-Case Analysis (TWCA) as presented e.g. in [17] [23] aims at providing weakly-hard guarantees for real-time systems, where a weakly-hard guarantee states that in no more than m out of k consecutive executions of a task, a deadline is missed. TWCA relies on the assumption that deadline misses in a system are due to transient overload resulting e.g. from sporadic activations.

We present here a specific application scenario of TWCA where activations of some specific tasks are considered as overload while activations of all other tasks are classified as typical.

We say that the system is in the *typical case* in a time interval in which there are no past or currently pending/executing overload activations which could impact the behavior of the system. We require the system to be schedulable in the typical case. The alternative case is called the *worst case* scenario where some overload activations may incur transient overload and therefore deadline misses.

The objective of TWCA is to compute a *deadline miss model* (DMM) for each task.

▶ **Definition 5.** A *deadline miss model* (DMM) for task τ_i is a function $dmm_i : \mathbb{N} \to \mathbb{N}$, with the property that out of any sequence of k consecutive activations (called k-sequence) of τ_i, at most $dmm_i(k)$ might miss their deadline D_i.

In the basic TWCA as introduced in [17], $dmm_i(k)$ is computed in four steps:

1. Computation of N_i, the number of deadline misses that occur in the longest level-i busy window BW_i. Note that one overload activation of any task cannot result in more than N_i deadline misses of τ_i as it can only impact activations of τ_i which are in the same level-i busy window.

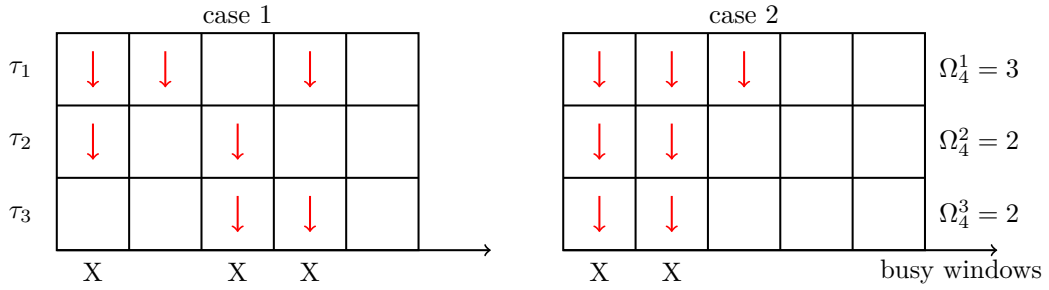


Figure 1 Packing overload activations into busy windows of task τ_4 (X = deadline miss).

2. Computation of ΔT_k^i, the longest time window during which an overload activation (of any task) can impact the response time of activations in the k-sequence. Activations in different busy windows cannot influence each other's response time. As a result, only activations of an overload task occurring at most BW_i time before the first activation of τ_i in the k-sequence and before the last activation finishes can have an impact. This time interval is thus bounded by:

$$\Delta T_k^i = BW_i + \delta_i^+(k) + WCRT_i\,.$$

3. Computation of Ω_i, the maximum number of higher-priority overload activations that may occur within a window of size ΔT_k^i:

$$\Omega_i = \sum_{\tau_j \in hp(i) \cap \mathcal{O}} \eta_j^+(\Delta T_k^i)$$

where $\mathcal{O}$ denotes the set of overload tasks.

4. We can then safely define $dmm_i(k) = \min\{k, \Omega_i \times N_i\}$.

The improved TWCA of [23] uses an additional concept called *combinations* to improve the accuracy of DMMs.

▶ **Definition 6.** A *combination* is a subset of the overload tasks, the idea being that one overload activation alone is usually not sufficient to cause a deadline miss as most tasks have some slack. Here we distinguish the overload due to different overload tasks:

$$\forall \tau_j \in \mathcal{O} \cap hp(i),\ \Omega_i^j = \eta_j^+(\Delta T_k^i)\,.$$

A bound on the maximum number of deadlines that τ_i may miss in a k-sequence is then obtained by packing overload activations into level-i busy windows.

▶ **Example 7.** As an example see Figure 1 where we consider a system with 4 tasks: τ_1, τ_2, τ_3 are overload tasks while τ_4 is a typical task. To bound the maximum number of deadlines that τ_4 may miss within a given time interval, we pack respectively Ω_i^1, Ω_i^2 and Ω_i^3 activations into the busy windows of τ_4. Figure 1 shows two possibilities of packing where in case 1 the number of deadline misses is 3 while in case 2 τ_4 may miss only 2 deadlines. Notice that not all combinations may lead to deadline misses.

▶ **Theorem 8.** *The following function is a DMM.*

$$dmm_i(k) = \min\{k, N_i \times \max\{\sum_{\overline{c} \in \mathcal{U}} x_{\overline{c}} \mid \forall \tau_j, \sum_{\{\overline{c} \in \mathcal{U} | \tau_j \in \overline{c}\}} x_{\overline{c}} \leq \Omega_i^j\}\}$$

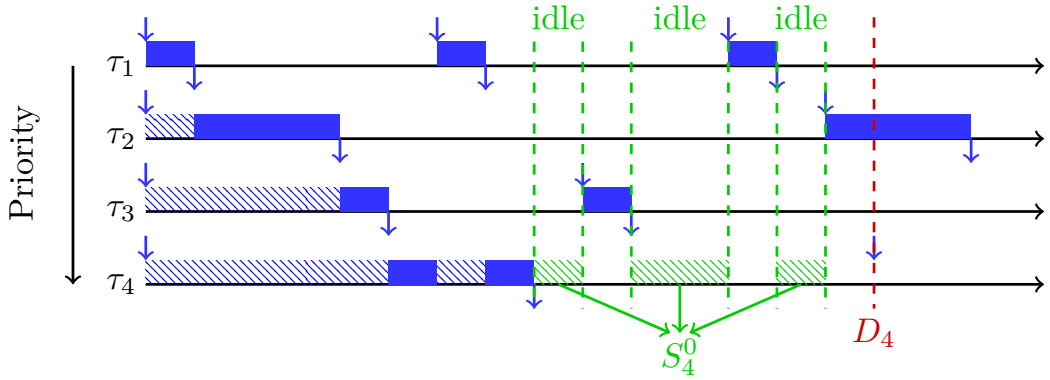


Figure 2 Worst-case busy window analysis. The slack S_4^0 of τ_i is shown.

where $\mathcal{U}$ is the set of unschedulable *combinations, i.e. combinations $\bar{c}$ which may lead to a deadline miss if all tasks in $\bar{c}$ are activated in the same level-i busy window. Note that in this approach it is assumed that all unschedulable may result in N_i deadline misses.*

[23] additionally provides an efficient criterion to determine whether a combination is schedulable as well as an efficient ILP solution to compute the above DMM.

4.3 Slack analysis

Finally, we now recall some results related to slack analysis [5],[11],[18], [20].

▶ **Definition 9.** The *slack* S_i^0 of task τ_i is the maximum amount of processing time which may be stolen from any job of τ_i without causing its deadline to be missed.

The slack of a task τ_i can be computed by noticing that any level-i idle time between the completion of a job of τ_i and its deadline can be used for computation of that job without causing it to miss its deadline.

▶ **Definition 10.** By *level-i idle time* we refer to any maximal time interval between two level-i busy windows.

▶ **Theorem 11.** *For FPP scheduling, the slack of τ_i is equal to the sum of all level-i idle times between the critical instant and D_i in the worst-case busy window.*

This is illustrated in Figure 2.

5 Budgeting with Hard Real-Time Constraints

In this section, we first focus on the problem of providing a set of constraints on the load incurred by the tasks in $\mathcal{R}$ (i.e. recovery tasks, a.k.a. under-specified tasks) that is sufficient to guarantee schedulability of all tasks in the nominal mode, before we move to discuss weakly-hard schedulability.

Let us first focus on a task τ_i in the nominal mode. Denote $\mathcal{R}_i$ the set of under-specified tasks with a priority higher than τ_i. We can directly reuse the concept of slack to budget the under-specified tasks.

▶ **Lemma 12.** *Let S_i^0 be the slack of τ_i in the system made of only nominal tasks (i.e. excluding under-specified tasks). If $\sum_{\tau_r \in \mathcal{R}_i} C_r \leq S_i^0$ and $\delta_r^-(2) > D_i$ then τ_i is schedulable.*

Proof. This follows directly from the definition of slack. Note that we need to ensure that at most one activation of any under-specified task will interfere with a given job of τ_i for the result to hold. ◀

We can generalize the above result by splitting the load allocated to an under-specified task among several of its jobs.

▶ **Lemma 13.** *Let BW_i^0 be the longest level-i busy window obtained by analyzing the nominal task set with an additional load of size S_i^0. That is:*

$$BW_i^0 = \min\{\Delta T \geq 0 \mid \Delta T = C_i + S_i^0 + \sum_{\tau_j \in \mathcal{N} \cap hp(i)} \eta_j^+(\Delta T) \times C_j\}.$$

If $\sum_{\tau_r \in \mathcal{R}_i} \eta_r^+(BW_i^0) \times C_r \leq S_i^0$ then τ_i is schedulable.

Proof. Again, this follows directly from the definition of slack. In this case the slack used by an under-specified task τ_r is shared among several of its jobs. ◀

We can now state our general result on how to budget under-specified tasks to guarantee hard real-time schedulability of all nominal tasks.

▶ **Theorem 14.** *If for all $\tau_i \in \mathcal{N}$*

$$\sum_{\tau_r \in \mathcal{R}_i} \eta_r^+(BW_i^0) \times C_r \leq S_i^0 \tag{3}$$

then the system is schedulable.

Proof. The above equation and Lemma 13 guarantee together than all nominal tasks remain schedulable in presence of under-specified tasks satisfying the given constraints. ◀

If this budget is acceptable then there is no need to consider budgeting for the weakly-hard case. The rest of this paper is dedicated to proposing solutions if a larger budget is needed for execution times of the under-specified tasks.

6 Budgeting with Weakly-Hard Real-Time Constraints

Our problem is now to provide a set of constraints on the load incurred by the tasks in $\mathcal{R}$ that is sufficient to guarantee *weakly-hard* schedulability of all tasks in the nominal mode rather than (hard) schedulability.

Again, we first focus on a task τ_i in the nominal mode, this time supposing that it has an (m, k) weakly-hard requirement, i.e. τ_i may miss no more than m out of k deadlines. Denote $\mathcal{R}_i$ the set of under-specified tasks with a priority higher than τ_i.

As recalled in Section 4.2, the standard way to establish (m, k)-schedulability using Typical Worst-Case Analysis [23] is to consider a sequence of k consecutive activations of τ_i and to prove that no more than m activations in this sequence may miss their deadline. In our case the activations of under-specified tasks can be considered as overload since they are not taken into account by the initial worst-case analysis. We can therefore adapt TWCA to our context. We reuse in particular the following notations.

- N_i, the number of deadline misses that occur in the longest level-i busy window BW_i of the system with nominal and under-specified tasks.

- ΔT_k^i, the longest time window during which an activation of an under-specified task can impact the response time of activations in the k-sequence.

$$\Delta T_k^i = BW_i + \delta_i^+(k) + WCRT_i\,.$$

- Ω_i^r, the maximum number of activations of higher-priority under-specified task τ_r that may occur within a window of size ΔT_k^i:

$$\Omega_i^r = \eta_r^+(\Delta T_k^i)$$

and Ω_i the sum over all higher-priority under-specified tasks:

$$\Omega_i = \sum_{\tau_r \in \mathcal{R}_i} \Omega_i^r\,.$$

Notice here that budgeting according to constraints on N_i and ΔT_i^k is not easy as these parameters themselves depend on the parameters of the under-specified tasks. In the next section we first focus on how to relate the load budget of recovery tasks and N_i, i.e. the maximum number of deadline misses in a single busy window.

6.1 Extending the concept of slack to weakly-hard systems

Let us start with a few lemmas.

▶ **Lemma 15.** *There can be more than one activation of a given task τ_i in one level-i busy window only if that task misses its deadline in that busy window. Formally: $K_i \geq 2$ only if $WCRT_i > D_i$.*

Proof. By definition of K_i, $B_i(K_i) \leq \delta_i^-(K_i+1)$ and for any $q < K_i$, $B_i(q) > \delta_i^-(q+1)$. For $q = 1 : B_i(1) > \delta_i^-(2)$. We work with constrained deadlines so $D_i \leq \delta_i^-(2)$ so $B_i(1) > D_i$. As $B_i(1) = RT_i(1)$ and therefore $WCRT_i \geq B_i(1)$ we can conclude that $WCRT_i > D_i$. ◀

This lemma is easily generalized to consecutive deadlines misses: $\forall q < K_i, RT_i(q) > D_i$. This result is useful for us as it directly relates the number of deadline misses in a busy window with the length of that busy window. In particular, we obtain that $N_i = K_i - 1$ if $B_i(K_i) \leq \delta_i^-(K_i) + D_i$.

Let us now go one step further and extend the slack analysis of Section 4 to systems in which a bounded number of deadline misses are allowed.

▶ **Definition 16.** For $\mu \in \mathbb{N}$, the μ-slack of a task τ_i, denoted S_i^μ, is the maximum amount of processing time which may be stolen from τ_i in a level-i busy window without causing more than μ deadlines of τ_i to be missed in a row.

The μ-slack of a task τ_i can be computed in a way similar to the usual slack but focusing on the $(\mu+1)$-th deadline instead of the first deadline.

▶ **Theorem 17.** *For FPP scheduling, the μ-slack of τ_i is equal to the sum of all level-i idle times between the critical instant and $\delta_i^-(\mu+1) + D_i$ in the worst-case busy window.*

Proof. The above condition guarantees that the $(\mu+1)$-th deadline is met. ◀

Let us now introduce a definition which will be useful to bound BW_i and $WCRT_i$.

▶ **Definition 18.** Let BW_i^μ be the longest level-i busy window obtained by analyzing the nominal task set with an additional load of size S_i^μ. We know that such a busy window contains exactly $\mu + 1$ activations of τ_i so:

$$BW_i^\mu = \min\{\Delta T \geq 0 \mid \Delta T = (\mu+1) \times C_i + S_i^\mu + \sum_{\tau_j \in \mathcal{N} \cap\, hp(i)} \eta_j^+(\Delta T) \times C_j\}\,.$$

Since τ_i may not miss more than m deadlines in a row, we can conclude that $BW_i \leq BW_i^m$. Similarly $WCRT_i$ is bounded by the response times of τ_i observed in BW_i^m. We thus know how to define ΔT_i^k. Let us now state the condition which guarantees that τ_i may not miss more than m deadlines in a row, and thus $N_i = m$.

▶ **Lemma 19.** *If $\sum_{\tau_r \in \mathcal{R}_i} \eta_r^+(BW_i^m) \times C_r \leq S_i^m$ then τ_i cannot miss more than m deadlines in a row.*

Proof. This is a direct consequence of the definition of m-slack. ◀

At this point, it may seem that the intuitive, if pessimistic, way to budget the under-specified tasks is to require that $\sum_{\tau_r \in \mathcal{R}_i} \eta_r^+(\Delta T_i^k) \times C_r \leq S_i^m$. This, however, is not a sufficient condition for (m,k)-schedulability. The reason is that the same load incurred by under-specified tasks may result in more deadline misses if they happen in different busy windows. This is the meaning of the following lemma.

▶ **Lemma 20.**

$$\forall \mu \in \mathbb{N}^+ : S_i^\mu \geq (\mu+1) \times S_i^0\,.$$

Proof. Consider a sequence of $\mu + 1$ consecutive activations of τ_i. Remember that S_i^0 is the sum of all level-i idle times between the critical instant and D_i in the worst-case busy window. Because deadlines are constrained, this is smaller than or equal to the sum of all level-i idle times between the critical instant and $\delta_i^-(2)$ in the worst-case busy window. Allowing only S_i^0 slack for each activation in the sequence furthermore assumes that the critical instant may repeat for each activation, which is pessimistic compared to the way S_i^μ is computed. As a result, S_i^μ provides more slack than $(\mu+1) \times S_i^0$. ◀

The consequence of this is that a safe bound on the budget for the under-specified tasks must be based for now on S_i^0.

▶ **Lemma 21.** *Let $\Lambda_i = (m+1) \times S_i^0$. If*

$$\sum_{\tau_r \in \mathcal{R}_i} \eta_r^+(\Delta T_i^k) \times C_r \leq \Lambda_i$$

then τ_i is (m,k)-schedulable.

Proof. We have to prove that a load of Λ_i within ΔT_i^k causes no more than m consecutive deadline misses if it occurs in one level-i busy window of τ_i, and no more than m non-consecutive deadline misses if it distributes over several busy windows.

- The first condition is directly satisfied by Lemmas 19 and 20.
- Suppose now that Λ_i is distributed over n level-i busy windows with l_b denoting the load in each busy window: $\sum_{b=1}^n l_b = \Lambda_i$. For each l_b let μ_b denote the maximum number of (consecutive) deadline misses that may be caused by l_b ($\mu_b \geq 0$). We have to prove that

$\sum_{b=1}^{n} \mu_b \leq m$. By definition we know that $l_b > S_i^{\mu_b - 1}$ for all l_b so from Lemma 20 we can derive that $l_b > \mu_b \times S_i^0$. If we now sum this over all l_b we get

$$\sum_{b=1}^{n} l_b > \sum_{b=1}^{n} \mu_b \times S_i^0 \,.$$

Since $\sum_{b=1}^{n} l_b = \Lambda_i = (m+1) \times S_i^0$ we can conclude that $m + 1 > \sum_{b=1}^{n} \mu_b$, which is what we had to prove.

◀

▶ **Theorem 22.** *If for all $\tau_i \in \mathcal{N}$ with an (m, k) schedulability constraint*

$$\sum_{\tau_r \in \mathcal{R}_i} \eta_r^+(\Delta T_i^k) \times C_r \leq (m+1) \times S_i^0 \tag{4}$$

then the system satisfies its hard and weakly-hard requirements.

Proof. This results is a direct consequence of Lemma 21. ◀

This result is obviously quite pessimistic. It is clear at this point that obtaining better bounds requires us to use a more fine-grained model of how load distributes over busy windows. We investigate this possibility in the next section.

6.2 Budgeting for multiframe tasks

In the following, we focus on a specific application scenario and assume that each under-specified task performs two activities:

- A frequent monitoring activity with a relatively short execution time aiming at analyzing deviations from safe state in the system and perform some rapid recovery or triggering higher-level recovery, characterized by a short minimum distance between two consecutive occurrences.
- A less frequent failure recovery activity (e.g., an avionics reconfiguration procedure) which requires a longer execution time and characterized by a longer minimum time distance between two consecutive executions.

Based on the behavior described above, the execution time model of any under-specified task τ_r can be characterized by (C_r^l, C_r^s, x) where:

- C_r^s is the short execution time corresponding to the recovery activity of the task;
- C_r^l is the long execution time corresponding to the error handling activity of the task;
- x is the number of short execution times between two long execution times.

Based on this new model we again address the problem of providing a set of constraints on the execution times and activation patterns of the tasks in $\mathcal{R}$ that is sufficient to guarantee weakly-hard schedulability of all tasks τ in the nominal mode.

Let us first focus on a task τ_i in the nominal mode with an (m, k) weakly-hard requirement, i.e. τ_i may miss no more than m out of k deadlines. Denote $\mathcal{R}_i$ the set of under-specified tasks with a priority higher than τ_i, $\Omega_i^r = \eta_r^+(\Delta T_i^k)$ for all $\tau_r \in \mathcal{R}_i$ and $\Omega_i = \sum_{r \in \mathcal{R}_i} \Omega_i^r$.

Let us first by formulating a hypothesis which is consistent with the application scenario mentioned at the beginning of this section.

▶ **Hypothesis 1.** *For each task $\tau_r \in \mathcal{R}_i$, we allow only one instance out of Ω_i^r to have a long execution time C_r^l. The other $\Omega_i^r - 1$ activations of τ_r within ΔT_i^k will be bounded by the short execution time bound C_r^s.*

In a way that is similar to the state of the art in TWCA as explained in Section 4.2 we now introduce the concept of combinations.

▶ **Definition 23.** A *level-i combination* is a tuple $\bar{c} = (c_1, c_2, \ldots, c_{|\mathcal{R}_i|})$ such that each task $\tau_r \in \mathcal{R}_i$ corresponds to one c_r in the tuple and $c_r = 0$ or $c_r = C_r^s$ or $c_r = C_r^l$.

We use the notation $c_r^{\bar{c}}$ to refer to the execution time of τ_r in combination $\bar{c}$. Note that we exclude here the possibility for several activations of the same under-specified task to be in the same level-i busy window. That is, we suppose that $\forall \tau_r \in \mathcal{R}_i : \delta_r^-(2) > BW_i^m$.

▶ **Definition 24.** Let $\mu(\bar{c})$ denote the maximum number of deadlines misses which may be caused by a combination $\bar{c}$. Formally we have:

$$S_i^{\mu(\bar{c})-1} < \sum_{\tau_r \in \mathcal{R}_i} c_r^{\bar{c}} \leq S_i^{\mu(\bar{c})}$$

with the convention that $S_i^{-1} = 0$. If $\mu(\bar{c}) = 0$ then $\bar{c}$ is called *schedulable*, otherwise it is said to be *unschedulable*.

Of course $\mu(\bar{c})$ depends on the values chosen for the various execution times C_r^l and C_r^s for $\tau_r \in \mathcal{R}_i$. Our strategy for budgeting the under-specified tasks is to first assign values on $\mu(\bar{c})$ for all combinations and then in a second step to assign execution time budgets.

▶ **Hypothesis 2.** *We suppose that a combination containing only short execution times of under-specified tasks cannot be unschedulable. That is,* $\sum_{\tau_r \in \mathcal{R}_i} C_r^s \leq S_i^0$.

Again this hypothesis seems realistic given the application context.

Based on the notion of combination we can define *gangs* which correspond to distributions of the Ω_i instances within ΔT_i^k. More specifically, a gang is a packing of activations of the under-specified tasks into the level-i busy windows of ΔT_i^k.

▶ **Definition 25.** A *gang* $\mathcal{G}$ is a set of combinations which contain at least one long execution time and such that for all $\tau_r \in \mathcal{R}_i$

- $\#\{\bar{c} \in \mathcal{G} \mid c_r^{\bar{c}} > 0\} \leq \Omega_i^r$
- $\#\{\bar{c} \in \mathcal{G} \mid c_r^{\bar{c}} = C_r^l\} = 1$

Notice that we ignore combinations which do not contain any long execution time as they cannot lead to deadline misses. Note also that each combination appears at most once in a gang (since there can be only one long execution time of each task within ΔT_i^k).

We use $\mathbb{G}_i$ to denote all possible gangs with respect to τ_i.

▶ **Lemma 26.** *If* $\forall \mathcal{G} \in \mathbb{G}_i : \sum_{\bar{c} \in \mathcal{G}} \mu(\bar{c}) \leq m$ *then* τ_i *is* (m, k)*-schedulable.*

Proof. The above condition guarantees that no matter how activations of under-specified tasks align, they can never result in more than m deadline misses. ◀

This lemma trivially extends to upper bounds on the $\mu(\bar{c})$ as we formulate now.

▶ **Lemma 27.** *For all* $\bar{c}$*, let* $\mu_{\bar{c}}$ *be an upper bound on* $\mu(\bar{c})$*. If* $\forall \mathcal{G} \in \mathbb{G}_i : \sum_{\bar{c} \in \mathcal{G}} \mu_{\bar{c}} \leq m$ *then* τ_i *is* (m, k)*-schedulable.*

Now, one thing which does not appear in the above lemma is that the $\mu(\bar{c})$ are not independent from each other.

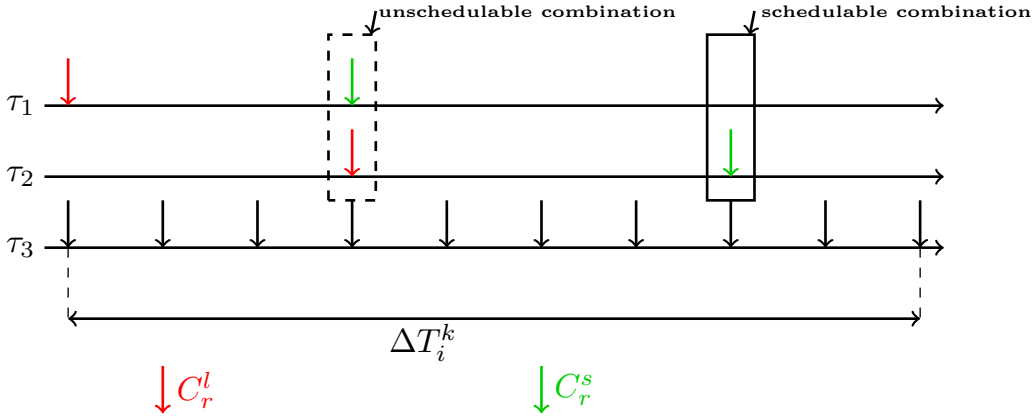


Figure 3 A gang of τ_1 and τ_2 within ΔT_i^k where τ_3 has a real time constraints (2,10).

▶ **Definition 28.** There exists a partial order $\leq$ on combinations such that $\bar{c}_1 \leq \bar{c}_2$ if and only if the execution times in $\bar{c}_1$ are all smaller than their counterpart in $\bar{c}_2$, i.e.,

$$\forall \tau_r \in \mathcal{R}_i : c_r^{\bar{c}_1} \leq c_r^{\bar{c}_2} .$$

▶ **Lemma 29.** *If $\bar{c}_1 \leq \bar{c}_2$ then $\mu(\bar{c}_1) \leq \mu(\bar{c}_2)$.*

Proof. This directly follows from the fact that $\bar{c}_1 \leq \bar{c}_2$ implies that the load incurred within one level-i busy window by the under-specified tasks in $\bar{c}_1$ is smaller than that in $\bar{c}_2$. ◀

▶ **Theorem 30.** *Suppose that you have assigned the $\mu_{\bar{c}}$ such that $\forall \mathcal{G} \in \mathbb{G}_i : \sum_{\bar{c} \in \mathcal{G}} \mu_{\bar{c}} \leq m$. Then any assignment of the $c_r^{\bar{c}}$ such that for all combination $\bar{c}$, $\sum_{r \in \mathcal{R}_i} c_r^{\bar{c}} \leq S_i^{\mu_{\bar{c}}}$ guarantees the (m,k)-schedulability of τ_i.*

Proof. This follows directly from Lemma 27 and the definition of $\mu_{\bar{c}}$-slack. ◀

Note that there always exists such an assignment.

Now that we have presented our solution for budgeting under-specified tasks based on the multiframe execution time model, let us show how it proceeds on an illustrative example.

▶ **Example 31.** Consider as an example a system with only one task τ_3 in the nominal mode and two under-specified tasks τ_1 and τ_2, as illustrated in Figure 3. Task τ_3 has a $(2, 10)$ weakly-hard requirement. τ_1 and τ_2 have priorities higher than the priority of τ_3, and no more than 2 instances within ΔT_i^k.

Figure 3 shows gang $\mathcal{G} = \{\bar{c}_1, \bar{c}_4, \bar{c}_7\}$ where $\bar{c}_1 = (C_1^l)$, $\bar{c}_4 = (C_1^s, C_2^l)$ and $\bar{c}_7 = (C_2^s)$ – to improve readability we omit 0s in the representation of combinations.
There are five combinations containing at least one long execution time:

$$\bar{c}_1 = (C_1^l), \bar{c}_2 = (C_2^l), \bar{c}_3 = (C_1^l, C_2^s), \bar{c}_4 = (C_1^s, C_2^l), \bar{c}_5 = (C_1^l, C_2^l) .$$

There are three more combinations containing at least one short execution time:

$$\bar{c}_6 = (C_1^s), \bar{c}_7 = (C_1^s), \bar{c}_8 = (C_1^s, C_2^s) .$$

Let us now focus on gangs. Remember that gangs consist of combinations containing at least one long execution time and that two combinations with the long same execution time cannot be in the same gang. We only list here maximal gangs.

$$\mathcal{G}_1 = \{\bar{c}_1, \bar{c}_2\}, \mathcal{G}_2 = \{\bar{c}_1, \bar{c}_4\}, \mathcal{G}_3 = \{\bar{c}_2, \bar{c}_3\}, \mathcal{G}_4 = \{\bar{c}_3, \bar{c}_4\}, \mathcal{G}_5 = \{\bar{c}_5\} .$$

This yields the following constraints, the first five of which are directly derived from the gangs while the remaining four constraints are obtained by comparing combinations.

1. $\mu_{\bar{c}_1} + \mu_{\bar{c}_2} \leq 2$
2. $\mu_{\bar{c}_1} + \mu_{\bar{c}_4} \leq 2$
3. $\mu_{\bar{c}_2} + \mu_{\bar{c}_3} \leq 2$
4. $\mu_{\bar{c}_3} + \mu_{\bar{c}_4} \leq 2$
5. $\mu_{\bar{c}_5} \leq 2$
6. $\mu_{\bar{c}_1} \leq \mu_{\bar{c}_3}$
7. $\mu_{\bar{c}_3} \leq \mu_{\bar{c}_5}$
8. $\mu_{\bar{c}_2} \leq \mu_{\bar{c}_4}$
9. $\mu_{\bar{c}_4} \leq \mu_{\bar{c}_5}$

One solution to this set of constraints is e.g. $\mu_{\bar{c}_1} = 1, \mu_{\bar{c}_2} = 1, \mu_{\bar{c}_3} = 1, \mu_{\bar{c}_4} = 1, \mu_{\bar{c}_5} = 2$.

Assuming we have chosen the above assignment for the $\mu_{\bar{c}}$ we now the define the constraints to be satisfied by the execution times of tasks, one per combination and then one for the short execution times.

1. $C_1^l \leq S_i^1$
2. $C_2^l \leq S_i^1$
3. $C_1^l + C_2^s \leq S_i^1$
4. $C_1^s + C_2^l \leq S_i^1$
5. $C_1^l + C_2^l \leq S_i^2$
6. $C_1^s + C_2^s \leq S_i^0$

Any solution to this set of constraints guarantees (m, k)-schedulability of τ_i.

7 Methodology and Discussion

Let us now summarize the methodology that we propose to provide the architect with *simple answers* helping him/her dimension the tasks that are still under-specified in the system.

1. We first compute an execution time budget for the under-specified tasks which guarantees *hard real-time* constraints (zero deadline misses). If this execution time budget is acceptable for the architect then we do not need to go further.
2. If, however there is a need for larger execution times for the under-specified tasks, we then compute a second execution time budget which guarantees weakly-hard constraints. Taking into account *weakly-hard constraints* we can allow more load within shorter time windows but over longer time windows the load available for under-specified tasks is still limited.
3. If the activation patterns of the under-specified tasks are known and a *multiframe execution time model* is meaningful we can propose more relaxed bounds on execution times budgets.

8 Experimental Results

Let us now provide some experimental results we have obtained using the cplex constraint solver on budgeting under-specified tasks. We first address the motivational example of Section 2 and then present experiments made on synthetic test cases.

8.1 The OBSW case study

The case study presented in Section 2 is a system made of a single resource and a task set shown in Table 1 where 27 tasks are in the nominal mode and there are 3 recovery and reconfiguration tasks τ_{10}, τ_{11}, τ_{21} which are under-specified.

As discussed before in Section 2, all on-board software is currently typically analyzed with hard real-time techniques; and yet by experience, the overall system is still quite robust to

Table 2 Real-time constraints of tasks in $\mathcal{T}'$. (m, w) represents the maximum number of allowed deadline misses m every w seconds, (m, k) means that a task may miss at most m deadline out of k consecutive activations.

task	τ_{12}	τ_{13}	τ_{14}	τ_{15}	τ_{16}	τ_{17}	τ_{18}	τ_{19}	τ_{20}
(m, w)	(1,2)	(1,4)	(1,8)	(1,4)	(1,4)	(1,4)	(1,8)	(1,8)	(1,8)
(m, k)	(1,16)	(1,16)	(1,8)	(1,8)	(1,16)	(1,8)	(1,8)	(1,8)	(1,8)
task	τ_{22}	τ_{23}	τ_{24}	τ_{25}	τ_{26}	τ_{27}	τ_{28}	τ_{29}	τ_{30}
(m, w)	(1,16)	hard	hard	(1,8)	(1,8)	(1,16)	(1,16)	(1,16)	hard
(m, k)	(1,8)	hard	hard	(1,8)	(1,8)	(1,8)	(1,8)	(1,8)	hard

occasional deadline misses, although at the moment there is no necessity to formally evaluate such tolerance in the state-of-the-practice process.

For the sake of the case study we propose some weakly-hard constraints for tasks that are purposely quite aggressive: the reader could notice that in some cases a tolerance of 1 deadline every 2 seconds is admitted for some tasks. This would permit to ascertain the robustness (at least from the point of view of real-time constraints) of such representative task set even in case of severe degradation (which would require high sporadic load for the recovery activities).

The worst-case response time analysis of the nominal mode shows that the system is schedulable. Our goal is to synthesize a load budget for the under-specified tasks τ_{10}, τ_{11}, τ_{21} which guarantees that all weakly-hard real-time constraints described in Table 2 are satisfied. We show first the constraints on the execution times and activation models of the tasks in $\mathcal{R}$ which guarantee absence of any deadline miss before providing the same result when a few deadline misses are tolerated.

Note that tasks $\{\tau_1, \ldots, \tau_9\}$ have higher priority than the recovery and reconfiguration tasks so their timing properties do not depend on the budget of tasks in $\mathcal{R}$. They will therefore be excluded from our study. We denote by $\mathcal{T}'$ the remaining tasks with lower priority, that is: $\mathcal{T}' = \mathcal{T} \setminus \{\tau_1, \ldots, \tau_9\}$.

8.1.1 Budgeting with hard real-time constraints

If we want to guarantee that the system is schedulable then the budget to be shared between the under-specified tasks is $S_i^0 = 48.01$ms. If this budget is not sufficient for the architect we can propose a budget with weakly-hard real-time guarantees.

8.1.2 Budgeting with weakly-hard real-time constraints

If the architect can accept to work with weakly-hard rather than hard guarantees then the available budget for the recovery tasks is $(m + 1) \times S_i^0 = 96.02$ ms.

This budget is twice as much as the budget for the hard real-time case. We can obtain even better bounds by using a more fine-grained model of how load distributes over busy windows.

8.1.3 Budgeting for multiframe tasks

Let us assume that for all $\tau_i \in \mathcal{N}$ there are at most $\Omega_{10} = 1$, $\Omega_{11} = 3$ and $\Omega_{21} = 2$ activations of the under-specified tasks within ΔT_i^k. The following execution times guarantee

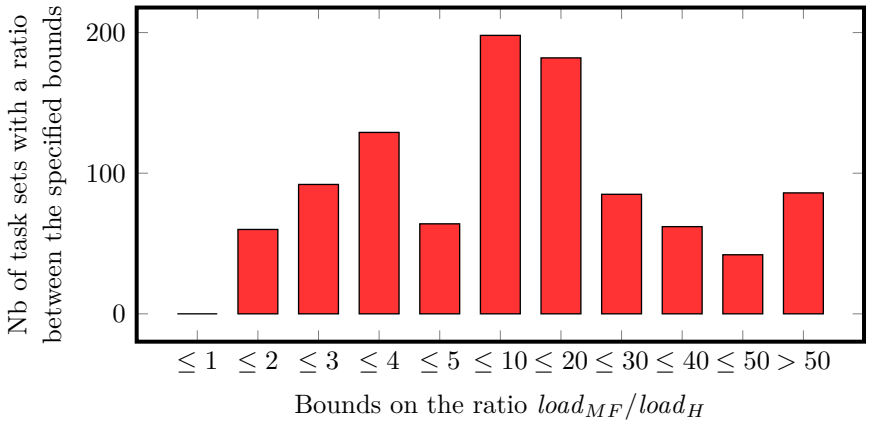


Figure 4 The relation between $load_{MF}$ and $load_H$.

(m, k)-schedulability of all tasks.

$$C_{11}^l = 24.005,\ C_{11}^s = 12.0025$$

$$C_{10}^l = 24.005,\ C_{10}^s = 12.0025$$

$$C_{21}^l = 24.005,\ C_{21}^s = 12.0025$$

This means in particular that the budget that is available for the under-specified tasks within ΔT_i^k is at least 108.015 ms. Note that there are many other possible assignments for the μ values which lead to different execution times.

8.2 Synthetic examples

In this section, we present a set of synthetic test cases to test more extensively our approach on a variety of systems. In this experiment we study the impact of different characteristics such as utilization, (m, k) constraints, system size, etc.

For that purpose we generated 1000 task sets randomly depending on UUniFast [7]. We define a set of tasks $\mathcal{T}$ with a priority, a worst-case execution time, a period, a deadline, and an (m, k) real-time constraint. The standard approach is to first define the system utilization and then assign a share of it to each task [7]. We picked up a utilization among $\{0.4, 0.5, 0.6, 0.7, 0.8\}$, then the number of tasks are chosen to be $\in [1, 20]$ and periods are *harmonic*. The worst-case execution time is then computed $C_i = U_i * T_i$. Deadlines $= \{0.6, 0.8, 1\} * T_i$ as our approach supports only constrained and implicit deadlines. We generate a random (m, k) for each task in the system such that: $k \in [2, 100], m \in [1, k-1]$. The number of under-specified tasks is limited to $r = 3$ and the maximum number of instances of each under-specified task is generated randomly to be in $[1, r^2]$.

8.2.1 Results

Figure 4 shows in the form of a histogram how much we gain in terms of load budget for the under-specified tasks by using a multiframe task model with weakly-hard constraints instead of using a single worst-case execution time with hard real-time constraints. Note that the results in the former case are obviously at least as good as those for the latter case.

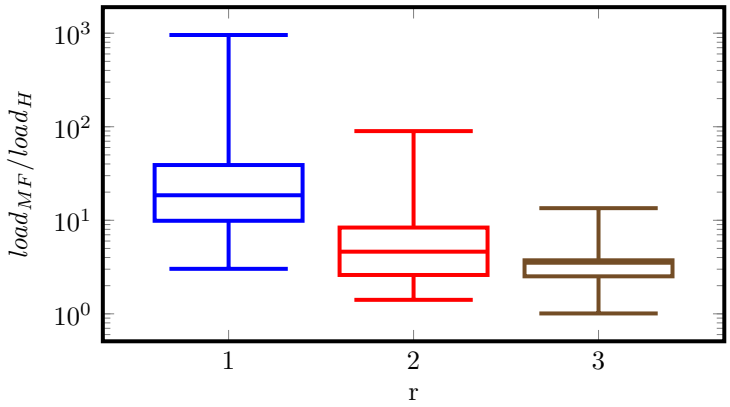


Figure 5 The relation between the gain of load and r.

Figure 4 shows for example that for 198 task sets the load budget in the multiframe case ($load_{MF}$) is between 5 and 10 times larger than the load budget in the hard case ($load_H$), that is:

$$5 < \frac{load_{MF}}{load_H} \leq 10\,.$$

The load we gain, however, is related to the number of under-specified tasks. Figure 5 shows that the larger the number of under-specified tasks the less load we gain, that is due to sharing the available slack among more under-specified tasks which makes the long execution C^l shorter.

The results shows that there is no impact of the utilization on the load we gain. Number of periodic tasks causes no degradation on the load we gain by using multiframe task model. Note that we have repeated our experiment 10 times and observed similar results.

9 Related Work

The work presented in this paper most closely relates to sensitivity analysis, slack analysis, multiframe task systems and weakly-hard real-time systems. Note that determination of bounds on unspecified system parameters is the scope of *Parametric Model Checking* [6] [9]. Even if such approaches are known to have difficulty scaling up to even simple settings, it would be interesting to see if these approaches could apply to our problem.

This work focuses on budgeting under-specified tasks for weakly-hard real-time systems. Although the under-specified tasks in our case study (OBSW) are recovery tasks, schedulability analysis of fault-tolerant real-time systems [4] [12] is not in the scope of this paper.

Sensitivity analysis is used to provide guarantees on the schedulability of a system in case of uncertainty on the system parameters. In [2] Bini et al. introduced an analytical sensitivity analysis for FPP scheduled periodic task sets with constrained deadlines (i.e. $D \leq T$). Work by [22] and [16] propose solutions for sensitivity analysis of systems with activation patterns specified with arrival curves.

In contrast to all these papers, our work proposes for the first time a solution for the sensitivity analysis of weakly-hard real-time systems: We constrain the admissible load that under-specified tasks in the system can use without violating weakly-hard real-time constraints in FPP scheduled task sets with arbitrary activation patterns and constrained deadlines.

Slack stealing is a scheduling algorithm proposed by [11] to schedule aperiodic tasks by stealing all the processing time it can from the periodic tasks without causing their deadlines to be missed. Similar algorithms based on slack stealing have been proposed by other authors [5] [18] [20]. These algorithms do not take into account any weakly hard guarantees and they, therefore, bound the maximum slack in a window of size D_i. In our approach, however, we consider (m, k) weakly-hard requirements and we thus bound the maximum slack in a window of size $\delta_i^-(m+1) + D_i$.

Multiframe task model was invented originally in [15] to provide a less pessimistic schedulability test than [13] for hard real-time systems. This model assigns to each *periodic* task N execution times $(C^0, C^1, \ldots, C^N)$, the execution time alternates between them where the execution time of the i-th instance of the task is $C^{((i-1) mod N)}$ where $i \geq 1$. In this paper we use a specific case of the multiframe task model for *sporadic* tasks which assigns two execution times: long C^l and short C^s where within a time window Δ one instance of the task uses the long execution time while the rest use the short execution times. We propose our multiframe task model in a context of budgeting *under-specified* recovery tasks (sporadic) to provide the recovery tasks with more load for weakly-hard real-time systems.

Weakly-hard systems [1] is a concept which guarantees that out of k consecutive executions of a task, not more than m deadline misses may occur. The approach of [17] and the related articles provide analyses to verify such constraints. In this paper we reuse the concepts developed in these papers to better budget under-specified tasks.

10 Conclusion

In this paper, we have shown how to budget under-specified tasks in the early design of weakly-hard real-time systems by providing sufficient conditions which guarantee (m, k) schedulability. This is particularly useful in industrial practice because it often happens during design that some parts of a task set are fully specified while other parameters, e.g. regarding recovery or monitoring tasks, do not become available before much later. Existing budgeting techniques, which are restricted to hard real-time constraints, can help anticipating how these missing parameters influence the behavior of the whole system, but they are likely to yield execution time budgets that are too tight to be useful. We have shown that using weakly-hard rather than hard guarantees, whenever possible, results in much more applicable execution time budgets. Our results are thus of real practical value for the design of systems such as the on-board software system discussed in the paper.

Note that in this paper we have not at all addressed the issue of the complexity of the analysis. The reason for that is that this does not appear to be a limiting factor for industrial applicability at this point. It would however be interesting to better understand how far the approach presented in this paper can scale and how much we can improve its efficiency.

Finally, we need to acknowledge the need for complementary work related to weakly-hard real-time systems as mentioned in Section 2, in particular in relation with the impact of deadline misses on system functions. Recent work [8, 3] in this direction indicate that this question is indeed considered as relevant in the research community as well as in the industry.

References

1 Guillem Bernat, Alan Burns, and Albert Llamosí. Weakly hard real-time systems. *IEEE Trans. Comput.*, 50(4):308–321, April 2001. doi:10.1109/12.919277.

2 Enrico Bini, Marco Di Natale, and Giorgio C. Buttazzo. Sensitivity analysis for fixed-priority real-time systems. In *18th Euromicro Conference on Real-Time Systems,*

ECRTS'06, 5-7 July 2006, Dresden, Germany, Proceedings, pages 13–22, 2006. doi:10.1109/ECRTS.2006.26.

3 Rainer Blind and Frank 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 2015, Osaka, Japan, December 15-18, 2015*, pages 7510–7515, 2015. doi:10.1109/CDC.2015.7403405.

4 A. Burns, R. Davis, and S. Punnekkat. Feasibility analysis of fault-tolerant real-time task sets. In *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems*, pages 29–33, June 1996. doi:10.1109/EMWRTS.1996.557785.

5 R. I. Davis, K. W. Tindell, and A. Burns. Scheduling slack time in fixed priority pre-emptive systems. In *Real-Time Systems Symposium, 1993., Proceedings.*, pages 222–231, December 1993. doi:10.1109/REAL.1993.393496.

6 Conrado Daws. Symbolic and parametric model checking of discrete-time markov chains. In *Proceedings of the First International Conference on Theoretical Aspects of Computing*, ICTAC'04, pages 280–294. Springer-Verlag, 2005. doi:10.1007/978-3-540-31862-0_21.

7 P. Emberson, R. Stafford, and R. I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, pages 6–11, July 2010.

8 Goran Frehse, Arne Hamann, Sophie Quinton, and Matthias Woehrle. Formal analysis of timing effects on closed-loop properties of control software. In *Proceedings of the IEEE 35th IEEE Real-Time Systems Symposium, RTSS 2014, Rome, Italy, December 2-5, 2014*, pages 53–62, December 2014. doi:10.1109/RTSS.2014.28.

9 Thomas Hune, Judi Romijn, Mariëlle Stoelinga, and Frits Vaandrager. Linear parametric model checking of timed automata. *The Journal of Logic and Algebraic Programming*, 52:183–220, 2002. doi:10.1016/S1567-8326(02)00037-1.

10 John P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *IEEE Real-Time Systems Symposium*, pages 201–213, 1990. doi:10.1.1.379.6163.

11 J. P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Real-Time Systems Symposium, 1992*, pages 110–123, December 1992. doi:10.1109/REAL.1992.242671.

12 George Lima and Alan Burns. Scheduling fixed-priority hard real-time tasks in the presence of faults. In *Proceedings of the Second Latin-American Conference on Dependable Computing*, LADC'05, pages 154–173, 2005. doi:10.1007/11572329_14.

13 C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973. doi:10.1145/321738.321743.

14 Ma Maode and H. Babak. A fault-tolerant strategy for real-time task scheduling on multiprocessor system. In *Parallel Architectures, Algorithms, and Networks, 1996. Proceedings., Second International Symposium on*, pages 544–546, June 1996. doi:10.1109/ISPAN.1996.509038.

15 Aloysius K. Mok and Deji Chen. A multiframe model for real-time tasks. *IEEE Trans. Software Eng.*, 23(10):635–645, October 1997. doi:10.1109/32.637146.

16 Moritz Neukirchner, Sophie Quinton, Tobias Michaels, Philip Axer, and Rolf Ernst. Sensitivity analysis for arbitrary activation patterns in real-time systems. In *Proc. of Design Automation and Test in Europe (DATE)*, March 2013. URL: http://dx.doi.org/10.7873/DATE.2013.041, doi:10.7873/DATE.2013.041.

17 Sophie Quinton, Matthias Hanke, and Rolf Ernst. Formal analysis of sporadic overload in real-time systems. In *DATE*, pages 515–520, March 2012. doi:10.1109/DATE.2012.6176523.

18 S. Ramos-Thuel and J. P. Lehoczky. On-line scheduling of hard deadline aperiodic tasks in fixed-priority systems. In *Real-Time Systems Symposium, 1993., Proceedings.*, pages 160–171, December 1993. doi:10.1109/REAL.1993.393504.

19 P. Rubel, M. Gillen, J. Loyall, R. Schantz, A. Gokhale, J. Balasubramanian, A. Paulos, and P. Narasimhan. Fault tolerant approaches for distributed real-time and embedded systems. In *MILCOM 2007 – IEEE Military Communications Conference*, pages 1–8, October 2007. doi:10.1109/MILCOM.2007.4455043.

20 Too-Seng Tia, Jane W.-S. Liu, and Mallikarjun Shankar. Algorithms and optimality of scheduling soft aperiodic requests in fixed-priority preemptive systems. *Real-Time Systems*, 10(1):23–43, 1996. doi:10.1007/BF00357882.

21 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, March 1994. doi:10.1007/BF01088593.

22 Ernesto Wandeler and Lothar Thiele. Real-time interfaces for interface-based design of real-time systems with fixed priority scheduling. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT'05, pages 80–89, 2005. doi:10.1145/1086228.1086246.

23 Wenbo Xu, Zain A. H. Hammadeh, Alexander Kröller, Sophie Quinton, and Rolf Ernst. Improved deadline miss models for real-time systems using typical worst-case analysis. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems*, Lund, Sweden, July 2015. doi:10.1109/ECRTS.2015.29.

Mixed-Criticality Scheduling With Dynamic Redistribution of Shared Cache*

Muhammad Ali Awan[1], Konstantinos Bletsas[2], Pedro F. Souto[3], Benny Akesson[4], and Eduardo Tovar[5]

1 **CISTER Research Centre and ISEP, Porto, Portugal**
muaan@isep.ipp.pt
2 **CISTER Research Centre and ISEP, Porto, Portugal**
ksbs@isep.ipp.pt
3 **University of Porto, Faculty of Engineering, Porto, Portugal; and CISTER Research Centre, Porto Portugal**
pfs@fe.up.pt
4 **CISTER Research Centre and ISEP, Porto, Portugal**
kbake@isep.ipp.pt
5 **CISTER Research Centre and ISEP, Porto, Portugal**
emt@isep.ipp.pt

Abstract

The design of mixed-criticality systems often involves painful tradeoffs between safety guarantees and performance. However, the use of more detailed architectural models in the design and analysis of scheduling arrangements for mixed-criticality systems can provide greater confidence in the analysis, but also opportunities for better performance. Motivated by this view, we propose an extension of Vestal's model for mixed-criticality multicore systems that (i) accounts for the per-task partitioning of the last-level cache and (ii) supports the dynamic reassignment, for better schedulability, of cache portions initially reserved for lower-criticality tasks to the higher-criticality tasks, when the system switches to high-criticality mode. To this model, we apply partitioned EDF scheduling with Ekberg and Yi's deadline-scaling technique. Our schedulability analysis and scalefactor calculation is cognisant of the cache resources assigned to each task, by using WCET estimates that take into account these resources. It is hence able to leverage the dynamic reconfiguration of the cache partitioning, at mode change, for better performance, in terms of provable schedulability. We also propose heuristics for partitioning the cache in low- and high-criticality mode, that promote schedulability. Our experiments with synthetic task sets, indicate tangible improvements in schedulability compared to a baseline cache-aware arrangement where there is no redistribution of cache resources from low- to high-criticality tasks in the event of a mode change.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

Keywords and phrases mixed criticality scheduling, vestal model, dynamic redistribution of shared cache, shared last-level cache analysis, cache-aware scheduling

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.18

* This work was partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within the CISTER Research Unit (CEC/04234); also by by FCT/MEC and the EU ARTEMIS JU within project ARTEMIS/0001/2013- JU grant nr. 621429 (EMC2).

29th Euromicro Conference on Real-Time Systems (ECRTS 2017).
Editor: Marko Bertogna; Article No. 18; pp. 18:1–18:21

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

1 Introduction

Many real-time embedded systems (automotive, avionics, aerospace) host functions of different criticalities. A deadline miss by a high-criticality function can be disastrous, but losing a low-criticality function only moderately affects the quality of service. Scalability and cost concerns favour mixed-criticality (MC) systems, whereby tasks of different criticalities are scheduled on the same core(s). However, this brings challenges. Lower-criticality tasks interfering unpredictably with higher-criticality tasks can be catastrophic. Conversely, rigid prioritisation by criticality leads to inefficient processor usage. Therefore, researchers have been working on scheduling models and techniques for (i) efficient use of processing capacity and (ii) schedulability guarantees for all tasks under typical conditions subject to (iii) ensured schedulability of high-criticality tasks in all cases. Most works [11] are based on Vestal's model [26, 5], which views the system operation as different modes, whereby only tasks of a certain criticality or above execute; additionally, different worst-case task execution times (WCETs) are assumed for the same task in each mode that it can be a part of, with corresponding degrees of confidence. This is because the cost of provably safe WCET estimation (and the associated pessimism) is justified only for high-criticality tasks. Other tasks have less rigorous WCET estimates, which might be exceeded, very rarely.

Many variants of the Vestal task model have been explored in recent years, with ever more sophisticated scheduling approaches and corresponding schedulability analysis techniques being devised for those. Yet, more progress is needed in terms of making the platform model more realistic, by incorporating more details about the architecture. The potential benefits could be (i) more accurate, hence safer, schedulability analysis, but also (ii) improved performance, from scheduling arrangements that acknowledge and leverage those architectural details. In particular, one could look for inspiration at efforts from the general-purpose (i.e., non-mixed-criticality) real-time systems domain, towards more *cache-aware* scheduling and analysis. Notably, Mancuso et al. [18], in the context of the Single-Core Equivalence (SCE) framework [24], consider (i) a cache-partitioned multicore architecture and (ii) task WCET estimates that are cognisant of the cache-partitioning.

Our work is inspired from the SCE framework and specifically seeks to integrate the effects of one particular shared resource, the last-level cache, into a dual-criticality Vestal model. We assume a last-level cache shared by all cores and partitioned among the different tasks via the Coloured Lockdown approach, to mitigate intra- and inter-core interference. For better resource usage and schedulability, instead of a static cache partitioning, we reclaim the cache pages allocated to low-criticality tasks (L-tasks) and redistribute those to high-criticality tasks (H-tasks), upon a switch to high-criticality mode (H-mode). In turn, the additional resources afforded to those tasks drive down their (cache-cognisant) H-mode WCETs. We propose a new mixed-criticality schedulability analysis that takes into account these effects, allowing for improvements in the guaranteed schedulability of the system. In a summary, these are the main contributions of our work:

1. We integrate the shared platform resources into a mixed-criticality model and dynamically redistribute those resources as a part of mixed-criticality scheduling. We demonstrate this principle by applying to the shared last-level cache.
2. We formulate schedulability analysis for the proposed model, assuming EDF scheduling using Ekberg and Yi's deadline scaling. Our analysis leverages the fact that cache resources are reclaimed from low-criticality tasks, in the event of a mode change, and redistributed to high-criticality tasks. This allows for improved schedulability.
3. We propose a two-staged allocation heuristic for allocating cache resources to the tasks, in the two modes of operation, and implement it by Integer Linear Programming (ILP).

Our experiments with synthetic task sets indicate appreciable schedulability improvements over approaches that perform no reclamation of cache-resources at mode change.

This paper is organised as follows. Section 2 presents the related work. The system model and the assumptions are discussed in Section 3. The schedulability analysis for that model is presented in Section 4, followed by some proposed heuristics for cache allocation to the tasks in the two modes, in Section 5. Section 6 presents and discusses the experiments used to evaluate the performance of the proposed approach. Conclusions are drawn in Section 7.

2 Related Work

Several feasibility tests are known for Vestal-model systems scheduled under, e.g., EDF or Fixed Priorities. One drawback, when using EDF, is that an H-task too close to its deadline, at the moment of a mode change, may be unable to accommodate its outstanding execution time (associated with its H-WCET) until its deadline, leading to a deadline miss. Therefore, the *deadline-scaling* technique was conceived [4, 13, 20, 16], to avert such scenarios if possible. It originated with EDF-VD [4], which uses standard EDF scheduling rules but, instead of reporting the real deadlines to the EDF scheduler for scheduling decisions, it reports shorter deadlines (if needed) for H-tasks during L-mode operation. This helps with the schedulability of H-tasks in the case of a switch to H-mode, because it prioritises H-tasks more than conventional EDF would, over parts of the schedule. This allows them to be sufficiently "ahead of schedule" and catch up with their true deadlines if any task overruns its L-WCET. In H-mode, the true H-task deadlines are used for scheduling and L-tasks are "dropped" (i.e., idled). EDF-VD proportionately shortens the H-task deadlines according to a single common scalefactor and its schedulability test considers the task utilisations in both modes. Ekberg and Yi [13] improved upon EDF-VD by enabling and calculating distinct scale factors for different H-tasks and using a more precise demand bound function (dbf) based schedulability test [6], for better performance. The scalefactor calculation is an iterative task-by-task process (for details, see [13, 14]).

However, the aforementioned scheduling solutions typically only consider the task execution on the processor cores and do not consider other platform resources, such as interconnects, caches and main memory. Some other works do consider interference on shared resources and propose mechanisms for its mitigation, albeit for single-criticality systems. For instance, several software-based approaches are proposed for mitigating cache and memory interference in multi-core platforms [18, 29, 21, 15, 8, 25]. Some of these works integrate the interference on shared resources to the schedulability analysis of the system. Mancuso et al. [19] integrate the effect of multiple shared resources (cache, memory bus, DRAM memory) on a multicore platform under partitioned fixed-priority preemptive scheduling. Pellizzoni and Yun [22] generalise the arrangement (and the analysis from [19]) to uneven memory budgets per core and propose a new analysis for different memory scheduling schemes. Behnam et al. [8] incorporated the effect of interference on shared resources under server-based hierarchical scheduling, that provides isolation between independent applications.

A software-based memory throttling mechanism for explicitly controlling the memory interference under fixed-priority preemptive scheduling is proposed in [28], although it only considers the timing requirements of tasks on a single critical core, whereupon all critical tasks are scheduled. The rest of the cores (interfering cores) are assumed to have non-critical tasks. Nevertheless, the analyses in existing works that consider the shared resources in the context of scheduling, assume that resources are statically allocated. Our proposed mixed-criticality algorithm considers the dynamic redistribution of shared resources, in order

to efficiently exploit their availability and improve the schedulability of the system. In this work, we demonstrate this principle with one particular resource: the last-level cache.

3 System model and assumptions

3.1 Platform

We assume a multicore platform composed of m identical cores accessing main memory via a shared memory controller. A core can have multiple outstanding (i.e., not served yet) memory requests . Prefetchers and speculative units are disabled. Our assumptions about the memory subsystem are inspired by those of the SCE [24] framework. We assume that all cores share a big last-level cache, but have dedicated upper-level caches (closer to the cores). Colored Lockdown [18] is used, to mitigate the intra-/inter-core interference. It allows a task to lock its σ most frequently used pages (hot pages) in the last-level cache, which facilitates upper-bounding the number of residual memory accesses (i.e., last-level cache misses) and, by extension, the WCET as a function of σ. In this work, we only analyse the integration and dynamic redistribution of one particular resource (the shared last-level cache) into a mixed criticality scheduling theory, as proof of concept, and we genuinely believe that a similar approach can be used to integrate other shared resources. The SCE framework also deploys the OS-level memory bandwidth regulation mechanism Memguard [30] and the DRAM-bank partitioning mechanism PALLOC [27] to mitigate the interference on those shared resources. In the future, we intend to also exploit these SCE mechanisms and dynamically redistribute memory access budgets at the mode switch.

3.2 Task model

Consider Vestal's base model with two criticality levels, high and low, as a starting point. Each task has an associated criticality, low or high. High-criticality tasks (H-tasks) have two WCET estimates: The L-WCET, which is *de facto* deemed safe, and the H-WCET, which is provably safe and possibly much greater. For low-criticality tasks (L-tasks), only the L-WCET is defined. There are two modes of operation. The system boots and remains in low-criticality mode (L-mode) as long as no job (instance of a task) executes for longer than its L-WCET. However, if any job exceeds its L-WCET, then the system immediately switches into high-criticality mode (H-mode) and permanently dispenses with the execution of all L-tasks. It is pessimistically assumed that in H-mode all jobs by H-tasks (including any existing jobs at the time of the mode switch) may execute for up to their H-WCET. Under these assumptions, it must be provable by an offline schedulability test that (i) no task misses a deadline in L-mode and (ii) no H-task misses a deadline in H-mode. We extend this basic model and assume that both the L-WCET and the H-WCET are functions of the number of the task's hottest pages locked in the last-level cache. In detail:

Our task set consists of n independent sporadic tasks ($\tau \stackrel{\text{def}}{=} \{\tau_1, \tau_2, \ldots, \tau_n\}$). Each task $\tau_i \in \tau$ has a minimum inter-arrival time T_i, a relative deadline D_i and a criticality level $\kappa_i \in \{L, H\}$ (low or high, respectively). The subsets of low-criticality and high-criticality tasks are defined as $\tau(L) \stackrel{\text{def}}{=} \{\tau_i \in \tau | \kappa_i = L\}$ and $\tau(H) \stackrel{\text{def}}{=} \{\tau_i \in \tau | \kappa_i = H\}$. We assume constrained deadlines, i.e., $D_i \leq T_i$. The original Vestal model is extended based on the following assumptions:

- The (actual) WCET of a task depends on the number of its pages (selected in order of access frequency) locked in place in the last-level cache.

- Different *estimates* of that WCET (derived via different techniques), are to be used for the L-mode and H-mode.

For each task τ_i, the L-WCET $C_i^L(\cdot)$ and the H-WCET $C_i^H(\cdot)$ are not single values, but rather functions of the pages locked in the last-level cache. For example $C_i^L(6)$ denotes the L-WCET of τ_i when this task is configured with its 6 "hottest" pages locked in the cache. How the ordered list of hot pages per task is obtained (and its accuracy) is beyond the scope of this paper and orthogonal to both the WCET estimation techniques and the safety of our analysis, as long as the same σ pages were assumed locked in cache when deriving $C_i^L(\sigma)$ and $C_i^H(\sigma)$. In practice, the profiling framework in [18] can be used for ranking each tasks's pages by access frequency. Estimating the WCET in isolation, for each task, assuming that the top σ pages in the list are locked in the cache, allows for the construction of a *progressive lockdown curve* (WCET vs number of locked pages in last-level cache). More locked pages in the last-level cache means fewer last-level cache misses (i.e., fewer residual memory requests) and, consequently, also a smaller WCET.

The technique in [18] for generating the progressive lockdown curve is measurement-based, so its output is not provably safe, but it can serve as the L-WCET progressive lockdown curve $C_i^L(\cdot)$. Moreover, some static analysis tools comprehensively cover all possible control flows (or even some infeasible paths) in a task, and these can be used to estimate the H-WCETs. By safely modelling accesses to the hot pages locked-in by Colored Lockdown as "always hit upon reuse", the static analysis tool can derive tighter WCET estimates than it would without this knowledge – and the improvement will be greater the more pages are locked in the cache. Hence, a progressive lockdown curve similarly exists for the H-WCET $C_i^H(\cdot)$.

To demonstrate the concept, Fig. 1 shows (imaginary) H- and L-mode progressive lockdown curves of a task τ_i. The x and y axes show the number of locked pages and WCET, respectively. Ideally, these two curves are non-increasing functions[1]. Let us assume that σ_i^L and σ_i^H denote the number of pages of a task τ_i locked in last-level cache in L- and H-mode, respectively. Then, the utilisation of a task in the L-mode (H-mode) is defined as $U_i^L(\sigma_i^L) \stackrel{\text{def}}{=} \frac{C_i^L(\sigma_i^L)}{T_i}$ (resp., $U_i^H(\sigma_i^H) \stackrel{\text{def}}{=} \frac{C_i^H(\sigma_i^H)}{T_i}$). In this paper we assume that the L- and H-mode progressive lockdown curves for each task are already provided to us as input. We also assume fully partitioned scheduling, i.e., no task ever migrates.

In case the overheads of unlocking and locking pages in the cache at mode change would be excessive, one could use per-task cache partitions *without any locking* (i.e., populated with lines dynamically). Techniques like APTA [23] could derive the equivalent of a parametric WCET curve as a function of the partition size in the L-mode and the H-mode. However, for simplicity, in the rest of the paper we assume the use of page locking.

3.3 Impact of mode change upon WCET

Under our model, a job by an H-task τ_i released in L-mode has its σ_i^L hottest pages in the cache and a job by the same task released in H-mode has its σ_i^H hottest pages in the cache. Both σ_i^L and σ_i^H are decided at design time (with $\sigma_i^H \geq \sigma_i^L$, $\forall \tau_i \in \tau(H)$). We assume that, as soon as a mode change occurs, the system can reclaim the cache pages hitherto allocated to L-tasks, for redistribution to the H-tasks. However, it is conservatively assumed that only new jobs by H-tasks, released after the mode change, benefit from the additional cache pages (either because it is only opportune to distribute them at the next release, or because, in the

[1] In the general case, the progressive lockdown curves are not necessarily convex, and we make no such assumption nor does our approach depend on such a property (convexity).

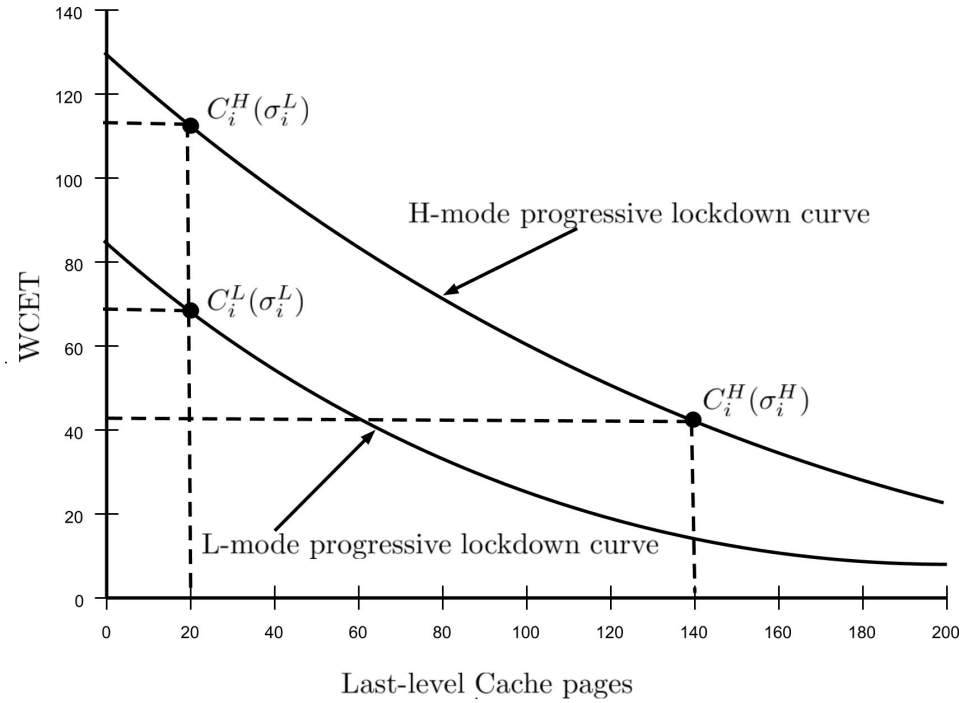


Figure 1 H-mode and L-mode progressive lockdown curves.

worst-case, the improvement from additional pages afforded to a job already having started its execution might not be quantifiable). For analysis purposes, we therefore conservatively assume that any H-job caught in the mode change may execute for up to $C_i^H(\sigma_i^L)$ time units, whereas any subsequent job by the same task only executes for up to $C_i^H(\sigma_i^H) \leq C_i^H(\sigma_i^L)$.

One interesting counter-intuitive property of our model is that there may be cases when $C_i^H(\sigma_i^H) \leq C_i^L(\sigma_i^L)$, unlike what holds for the classic Vestal model, where $C_i^H \geq C_i^L$ in all cases. This can happen if the reduction in last-level cache misses from the additional pages allocated to the task in the H-mode offsets the pessimism from using a more conservative estimation technique for H-WCETs than for L-WCETs. Fig. 1 illustrates this possibility. Leveraging such cases in the analysis can lead to improvements in provable schedulability, over approaches that do not reallocate cache pages in the event of a mode switch.

3.4 Aspects of deadline scaling

As already mentioned, in L-mode, the H-tasks report to the EDF scheduler a shorter deadline $D_i^L \leq D_i$, for the purpose of scheduling decisions. In H-mode, the true deadline is used (i.e., $D_i^H = D_i$). The designer has freedom over the selection of L-mode deadlines and the process that determines them is called deadline scaling. In [13], Ekberg and Yi propose a heuristic that, starting with $D_i^L = D_i$ for every task, iteratively tinkers with the task L-mode deadlines, using their schedulability test to guide the heuristic to identify opportunities to decrease a deadline by a notch. In our work, we also use the same heuristic (details in [13]), with no changes except for the fact that our new schedulability analysis, cognisant of cache reclamation by H-tasks at mode change, is used, instead of the original analysis in [13].

4 Schedulability analysis

In this section, we propose a schedulability analysis, drawing from that of Ekberg and Yi [13, 14], for the system model described earlier. It assumes that the number of hot pages in the two modes (σ_i^L and σ_i^H) for each task is given. Similarly, we also assume that the scaled L-mode deadline D_i^L, with $D_i^H \stackrel{\text{def}}{=} D_i$, is given for each task. As explained, this analysis is to be coupled with the heuristic of Ekberg and Yi to guide the derivation of the L-mode scaled deadlines. How to assign values to σ_i^L and σ_i^H, is discussed in the next section.

Ekberg and Yi's analysis is based on the *demand bound function*, $dbf(\ell)$, which upper-bounds the execution demand over any time interval of length ℓ by all jobs whose scheduling windows are fully contained in ℓ. The *scheduling window* of a job is the time interval between its release and its deadline. The schedulability analysis for the L-mode can be done using standard dbf for EDF, in which the computation demand of a task is maximum when a job is released at the beginning of the time interval. In H-mode, if the time interval under consideration begins at the mode switch, in addition to the demand of jobs whose scheduling windows are fully contained in ℓ, we need to consider the demand of *carry-over* jobs of H-tasks, i.e. jobs of H-tasks that were released, but not finished, at the time of the mode switch. Thus, for H-mode analysis, we consider that the scheduling window of a carry-over job always starts at the mode switch.

A key result of Ekberg and Yi's analysis is the following lemma, which allows to upper bound the demand in H-mode of a carry-over job:

▶ **Lemma 1** (Demand of carry-over jobs, Ekberg and Yi's [14]'s Lemma 1). *Assume that EDF uses relative deadlines D_i^L and D_i^H, with $D_i^L \leq D_i^H = D_i$ for high-criticality task τ_i, and that we can guarantee that the demand is met in low-criticality mode (using D_i^L). If the switch to high-criticality mode happens when a job from τ_i has a remaining scheduling window of x time units left until its true deadline, then the following hold:*

1. *If $x < D_i^H - D_i^L$, then the job has already finished before the switch.*
2. *If $x \geq D_i^H - D_i^L$, then the job may be a carry-over job, and no less than $[\![C_i^L - x + D_i^H - D_i^L]\!]_0$ time units of the job's work were finished before the switch.*

where

$$[\![z]\!]_{\min}^{\max} \stackrel{\text{def}}{=} \begin{cases} \min & \text{if } z < \min \\ z & \text{if } \min \leq z \leq \max \\ \max & \text{if } z > \max \end{cases}$$

and the bound arguments min and max can be omitted, if they are $-\infty$ or $+\infty$, respectively.

In the classic Vestal model there is no resource reallocation upon a mode switch, except for CPU time. Therefore, the computation demand of a carry-over job never exceeds the demand of a full job, and the maximum demand in any time interval of length ℓ corresponds to executions that maximise the number of full jobs after the mode switch as shown in Fig. 2(i). In this scenario, the time interval of length ℓ under consideration ends with a deadline for each task τ_i present in H-mode. Accordingly, the subinterval of length $x = \ell \bmod T_i$, which starts with the mode switch, is maximised, under the constraint that the number of full jobs is maximum, maximizing its demand, because, by Lemma 1, the maximum demand of a carry-over job is non-decreasing with the size of its scheduling window in H-mode.

In our model, $\sigma_i^L < \sigma_i^H$, therefore the maximum demand does not necessarily occur in executions as shown in Fig. 2(i). The reason is that a full job in H-mode executes with σ_i^H pages locked in the cache, whereas a carry-over job executes with only σ_i^L pages locked in

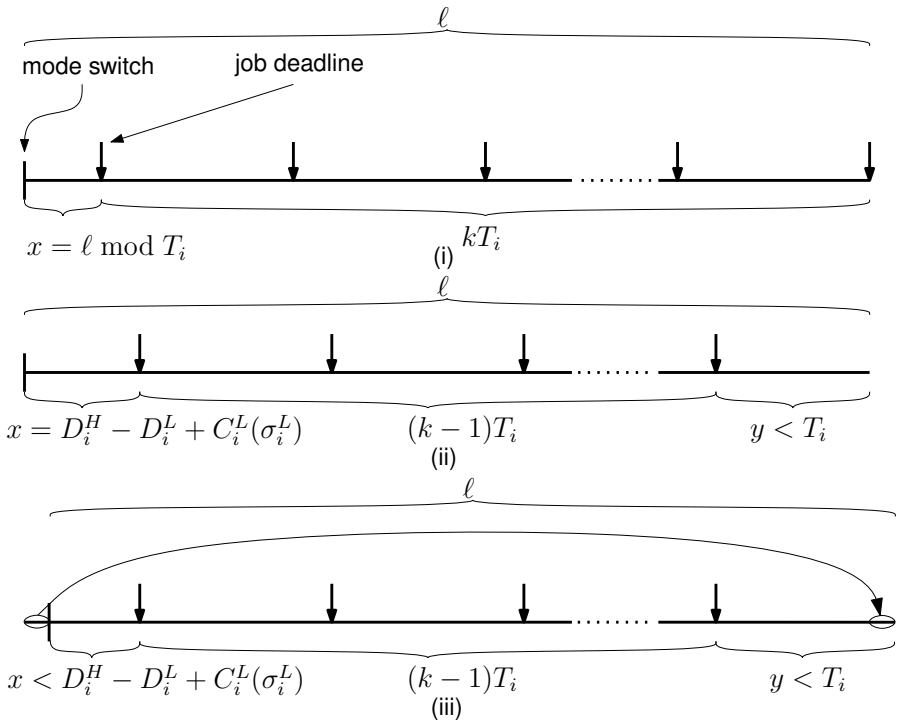


Figure 2 Execution with maximum demand for $C_i^H(\sigma_i^H) \geq C_i^L(\sigma_i^L)$.

the cache until the mode switch, and thus, for safety, we assume that it executes with only σ_i^L pages locked in the cache throughout its execution. Therefore, the demand in H-mode of a full job is not necessarily larger than the (outstanding) demand in H-mode of a carry-over job. For example, if $\ell = T_i$, then the execution shown in Fig. 2(i) has no carry-over job, and the maximum demand is $C_i^H(\sigma_i^H)$. However, for such a value of ℓ, we can have an execution in which there is a *maximal carry-over job*, i.e. a carry-over job with maximum demand in H-mode, $C_i^H(\sigma_i^L)$. If σ_i^L and σ_i^H are such that $C_i^H(\sigma_i^L) > C_i^H(\sigma_i^H)$, i.e. if the extra assigned cache lines are useful to the task and reduce its execution time, the latter execution scenario has a demand that is higher than the former.

Because, Ekberg and Yi's analysis, assumes that the demand of a carry-over job is never larger than the demand of a full-job, we need new analysis, built on the following lemma.

▶ **Lemma 2.** *In H-mode, for any time interval of length ℓ, the demand by the jobs of an H-task τ_i whose scheduling windows are fully contained in ℓ is maximum:*

1. *either in executions with the maximum number of full jobs after a carry-over job, if one fits, as illustrated in Fig. 2(i),*
2. *or in executions with the maximal carry-over job with the earliest possible deadline followed by as many full jobs as can fit in the remaining time and that arrive as soon as possible, as illustrated in Fig. 2(ii).*

Proof. The scheduling window of a carry-over job in H-mode always starts at the mode switch. Thus, in H-mode, a time interval of length ℓ can include the scheduling windows of at most one carry-over job of τ_i, and of a number of full jobs (that is, jobs released at or after the mode switch).

If $\ell < D_i^H - D_i^L + C_i^L(\sigma_i^L)$, no full job contributes to the demand, because the shortest length of the scheduling window of a full job is D_i and $\ell < D_i^H$ (because $C_i^L(\sigma_i^L) \leq D_i^L$).

Thus, in this case, execution scenario (1) maximises the amount of time that can be used by a carry-over job and, by Lemma 1, its demand is maximum.

Let $\ell \geq D_i^H - D_i^L + C_i^L(\sigma_i^L)$. Let y be the length of the right-most subinterval of ℓ, from the deadline of the last job contained in ℓ, if any, until the end of ℓ. If $0 < y < T_i$, execution scenario (2) maximises the demand at the beginning of the interval, because the earliest deadline of a maximal carry-over job is $D_i^H - D_i^L + C_i^L(\sigma_i^L)$, by Lemma 1 (after substituting C_i^L with $C_i^L(\sigma_i^L)$); but, the demand during y does not increase, because the deadlines of two consecutive jobs of τ_i must be T_i time units apart, and therefore subinterval y cannot contain the scheduling window of a full job. If x decreases by some amount and y increases by the same amount, as illustrated in Fig. 2(iii), the demand of the carry-over job decreases without increasing the demand at the end of the interval, unless y becomes T_i. If this happens, we transform execution scenario (2) into execution scenario (1) and increase the total demand of full jobs, but decrease the demand of the carry-over job, possibly eliminating it. Thus, if the total demand in ℓ increases when y becomes T_i, then execution scenario (1) has maximum demand, else execution scenario (2) has maximum demand. Decreasing x by a larger amount than necessary for y to become equal to T_i, does not increase the demand w.r.t. execution scenario (1), since it increases neither the total demand of full jobs nor the demand of the carry-over job. Finally, if $y = 0$, then execution scenarios (1) and (2) are identical and both have a maximal carry-over job of τ_i with the earliest deadline, and the maximum number of full-jobs of τ_i that can fit in ℓ, therefore their demand is maximum. ◀

Thus, a tight demand bound function for any execution in H-mode is the maximum of the demands of execution scenarios (1) and (2), illustrated respectively in Fig. 2(i) and (ii). Next, we adapt Ekberg and Yi's demand bound function for execution scenario (1) to take into account a different number of pages locked in the cache per mode. After that, we develop the demand bound function for execution scenario (2), which was not relevant in previous work.

In [14], Ekberg and Yi provide a bound for the demand of execution scenario (1) in a time interval of length ℓ, as follows:

$$full_i^H(\ell) - done_i^H(\ell) \tag{1}$$

where $full_i^H(\ell)$, given by (2), is the maximum demand by all jobs of τ_i whose scheduling window is fully contained in that interval (in H-mode, the scheduling window of a carry-over job begins at the mode switch and ends at its deadline), and $done_i^H(\ell)$, given by (3), is the minimum demand of any carry-over job that must be satisfied before the mode switch.

$$full_i^H(\ell) = \left[\!\!\left[\left(\left\lfloor \frac{\ell - \left(D_i^H - D_i^L\right)}{T_i} + 1 \right\rfloor \right) C_i^H \right]\!\!\right]_0 \tag{2}$$

$$done_i^H(\ell) = \begin{cases} [\![C_i^L - (\ell \bmod T_i) + D_i^H - D_i^L]\!]_0, & \text{if } (D_i^H - D_i^L) \leq \ell \bmod T_i < D_i^H \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

We now derive the new expressions for $full_i^H(\ell)$ and $done_i^H(\ell)$ to take into account that the number of pages locked in the cache in the L-mode and in the H-mode may be different. In this derivation, like Ekberg and Yi in [14], we assume that there is a carry-over job, if one fits. At the end of this section, we show that, for any time interval ℓ after the mode switch, the demand is maximum when there is a carry-over job.

So, assuming that the first job is a carry-over job, if one fits, we modify (2) (originally (2) in [14]) as follows:

$$full_i^H(\ell) = \left[\!\!\left[\left\lfloor \frac{\ell - (D_i^H - D_i^L)}{T_i} \right\rfloor + 1 \right]\!\!\right]_0^1 C_i^H(\sigma_i^L) + \left[\!\!\left[\left\lfloor \frac{\ell - (D_i^H - D_i^L)}{T_i} \right\rfloor \right]\!\!\right]_0 C_i^H(\sigma_i^H)\,. \tag{4}$$

The first term bounds the demand in H-mode of the carry-over job. As shown by Lemma 1, $D_i^H - D_i^L$ is the smallest scheduling window (in H-mode) of a carry-over job of τ_i. To be safe, we assume that the number of locked pages of the carry-over job is σ_i^L, therefore the maximum demand of the carry-over job, ignoring any demand that may have been satisfied before the mode switch, is $C_i^H(\sigma_i^L)$. The second term bounds the demand of the jobs that are released after the mode switch and therefore we use their maximum execution time with the respective number of locked pages in H-mode, $C_i^H(\sigma_i^H)$.

Likewise, for $done_i^H(\ell)$, we modify (3) (originating as (3) in [14]) by substituting C_i^L with $C_i^L(\sigma_i^L)$. That is, we make explicit that any computation before the mode switch must have been performed with σ_i^L pages locked in the cache.

Thus, by replacing (2) and (3) (i.e., (2) and (3) in [14]) with their versions aware of the number of pages locked in the cache, (1) provides a bound for execution scenario (1) when the number of pages locked in the cache is changed from σ_i^L to σ_i^H upon a switch to H-mode.

The demand under execution scenario (2) is a step function and is given by (5).

$$step_i^H(\ell) = \left[\!\!\left[\left\lfloor \frac{\ell - \left(D_i^H - D_i^L + C_i^L(\sigma_i^L)\right)}{T_i} \right\rfloor + 1 \right]\!\!\right]_0^1 C_i^H(\sigma_i^L) + \left[\!\!\left[\left\lfloor \frac{\ell - \left(D_i^H - D_i^L + C_i^L(\sigma_i^L)\right)}{T_i} \right\rfloor \right]\!\!\right]_0 C_i^H(\sigma_i^H) \tag{5}$$

where the first term bounds the demand in H-mode of the carry-over job, which is maximum and has a deadline at the earliest time instant, and the second term bounds the demand of the maximum number of full jobs that fit after the carry-over job.

Thus, a demand bound function for any interval of length ℓ in H-mode is:

$$dbf_i^H(\ell) = max\big(step_i^H(\ell),\ full_i^H(\ell) - done_i^H(\ell)\big) \tag{6}$$

Finally, we show that executions with a carry-over job have a higher demand than executions without a carry-over job, an assumption we made above in the derivation of $dbf_i^H(\ell)$.

▶ **Lemma 3.** *For any sporadic task τ_i, its maximum demand in H-mode in a time interval of length ℓ with only full jobs is not higher than its maximum demand in a time interval of the same length ℓ with a carry-over job.*

Proof. The demand of an execution of τ_i in H-mode with only full jobs can be bounded by the standard dbf for sporadic tasks with the appropriate parameters:

$$\left[\!\!\left[\left(\left\lfloor \frac{\ell - D_i^H}{T_i} \right\rfloor + 1 \right) C_i^H(\sigma_i^H) \right]\!\!\right]_0 . \tag{7}$$

If $\ell < D_i^H$ the demand is zero. Let $\ell \geq D_i^H$. Consider a time interval of length ℓ starting at the mode switch. Consider an execution in which the carry-over job has maximum demand,

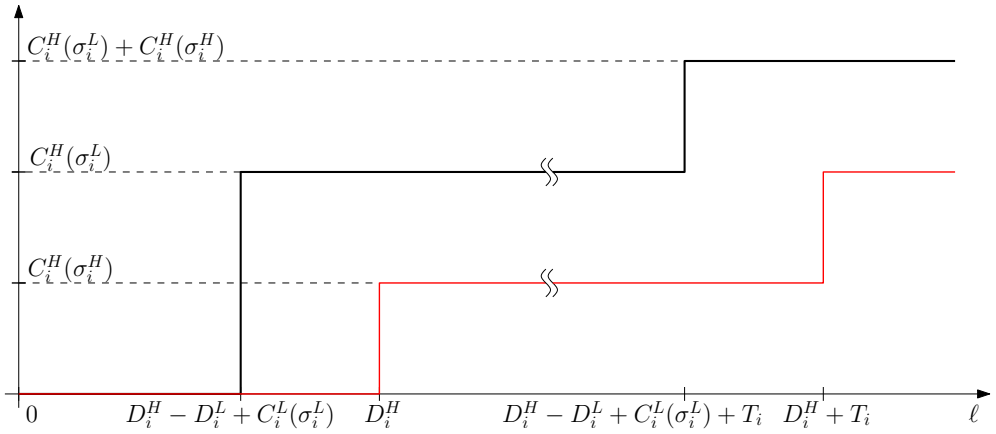


Figure 3 The demand over an interval of length ℓ starting at mode switch when the carry-over job is maximum at the earliest time (black line) dominates the maximum demand over an interval of the same length without a carry-over job.

$C_i^H(\sigma_i^L)$, at the earliest possible time, $D_i^H - D_i^L + C_i^L(\sigma_i^L)$, and the following jobs of τ_i arrive as soon as possible. For any time interval of length ℓ, the demand of such an execution is never lower than the demand of an execution without a carry-over job (see Fig. 3 for further intuition). Indeed:

1. $C_i^H(\sigma_i^L) \geq C_i^H(\sigma_i^H)$, because $\sigma_i^L \leq \sigma_i^H$.
2. $D_i^H - D_i^L + C_i^L(\sigma_i^L) \leq D_i^H$, because $D_i^L \geq C_i^L(\sigma_i^L)$. ◀

5 Last-level cache allocation

The description of our analysis assumed that for each task the number of its hottest pages that are locked in the cache in each mode is already determined. We now propose a heuristic for this allocation. Our objective is to efficiently distribute the last-level cache among the tasks, for improved schedulability.

A heuristic is needed because a brute-force combinatorial exploration of all possible allocations is intractable and the arbitrary nature of progressive lockdown curves means that there is no structure in the problem to employ for strict optimality, in the general case. Additionally, since any possible allocation configuration of the cache for the L-mode can be re-configured in many ways for the H-mode, we opt for a two-staged heuristic. We first determine the L-mode allocation and then, subject to the constraints stemming from that, we determine the H-mode allocation. Since the schedulability analysis is conceptually complex (and made even more so by the deadline scaling), our idea is to optimise, in each mode, for a metric that strongly correlates with schedulability: the task set utilisation. So, we first (i) assign values to the σ_i^L variables (corresponding to the number of locked pages in L-mode) for each task so that the L-mode utilisation ($\sum_{\tau_i \in \tau} \frac{C_i^L(\sigma_i^L)}{T_i}$ is minimised and subsequently (ii) assign values to the σ_i^H variables for the H-tasks (with $\sigma_i^H \geq \sigma_i^L$) such that the (steady) H-mode utilisation ($\sum_{\tau_i \in \tau(H)} \frac{C_i^H(\sigma_i^H)}{T_i}$) is minimised. Next, we discuss the ILP formulation implementing this heuristic.

5.1 L-mode allocation

Let σ_i^L be the number of pages by τ_i in the last-level cache in the L-mode and σ^T be the total number of pages that fit in that cache. Intuitively, lower utilisation correlates with better schedulability, hence, our objective is to set the σ_i^L values such that the total task set utilisation in L-mode is minimised. To model this heuristic with ILP formulation, we define a binary decision variable variable $UL_{i,j}$ such that:

$$UL_{i,j} = \begin{cases} 1, & \text{if } j \text{ pages are assigned to } \tau_i \in \tau \text{ in L-mode} \\ 0, & \text{otherwise} \end{cases}$$

Since our aim is to minimise the system utilisation in L-mode, the objective function and constraints take the form:

$$\text{Minimise} \sum_{\forall \tau_i \in \tau} \sum_{j=0}^{\sigma^T} UL_{i,j} \times U_i^L(j) \tag{8}$$

$$\textbf{s. t.} \sum_{j=0}^{\sigma^T} UL_{i,j} = 1, \ \forall \tau_i \in \tau \tag{9}$$

$$\sum_{\forall \tau_i \in \tau} \sum_{j=0}^{\sigma^T} j \times UL_{i,j} \leq \sigma^T \tag{10}$$

$$\sum_{j=0}^{\sigma^T} UL_{i,j} \times U_i^L(j) \leq 1, \ \forall \tau_i \in \tau \tag{11}$$

$$\sum_{\forall \tau_i \in \tau} \sum_{j=0}^{\sigma^T} UL_{i,j} \times U_i^L(j) \leq m, \ \forall \tau_i \in \tau \tag{12}$$

The $U_i^L(j)$ constants are derivable from the tasks' progressive lockdown curves. The set of constraints given by (9) ensures that each task is considered for allocation in the last-level cache. A task can be allocated any number of pages from zero to all σ^T pages in the cache. However, the sum of all pages allocated to tasks should not exceed the cache capacity (i.e., $\sum_{\forall \tau_i \in \tau} \sigma_i^L \leq \sigma^T$), which is ensured by (10). Additionally, the utilisation of each task in the L-mode for the selected number of pages should not exceed one (i.e., $U_i^L(\sigma_i^L) \leq 1$), which is ensured by (11); otherwise, the task will not be unschedulable. Finally, the set of constraints given by (12) ensures that the total utilisation of the task set in L-mode, under the particular allocation, should not exceed the number of cores in the platform ($\sum_{\forall \tau_i \in \tau} U_i^L(\sigma_i^L) \leq m$); otherwise the task set would be unschedulable in the L-mode, under these parameters.

5.2 H-mode allocation

In this second stage of our allocation heuristic, we determine how the pages reclaimed from the idled L-tasks at the switch from L-mode to H-mode, are to be redistributed to the H-tasks. Let σ_i^H denote the number of cache pages in the last-level cache allocated to a task $\tau_i \in \tau(H)$ in the H-mode. Our ILP formulation for the H-mode allocation derives σ_i^H for each task $\tau_i \in \tau(H)$ in such a way that the overall steady H-mode system utilisation is minimised. We define a binary decision variable $UH_{i,j}$ such that:

$$UH_{i,j} = \begin{cases} 1, & \text{if } j \text{ pages are assigned to } \tau_i \in \tau(H) \text{ in H-mode} \\ 0, & \text{otherwise} \end{cases}$$

The objective function in this stage minimises the H-mode utilisation and the ILP formulation is given below:

$$\text{Minimise} \sum_{\forall \tau_i \in \tau(H)} \sum_{j=0}^{\sigma^T} UH_{i,j} \times U_i^H(j) \tag{13}$$

$$\textbf{s. t.} \sum_{j=0}^{\sigma^T} UH_{i,j} = 1,\ \forall \tau_i \in \tau(H) \tag{14}$$

$$\sum_{\forall \tau_i \in \tau(H)} \sum_{j=0}^{\sigma^T} j \times UH_{i,j} \leq \sigma^T \tag{15}$$

$$\sum_{j=0}^{\sigma^T} UH_{i,j} \times U_i^H(j) \leq 1,\ \forall \tau_i \in \tau(H) \tag{16}$$

$$\sum_{\forall \tau_i \in \tau(H)} \sum_{j=0}^{\sigma^T} UH_{i,j} \times U_i^H(j) \leq m,\ \forall \tau_i \in \tau(H) \tag{17}$$

$$\sum_{j=0}^{\sigma^T} j \times UH_{i,j} \geq \sigma_i^L,\ \forall \tau_i \in \tau(H) \tag{18}$$

The constraints given by (14)-(17) are similar to those given by (9)-(12) for the L-mode. These constraints ensure that every H-task is considered for allocation, the sum of allocated pages does not exceed the total number of pages in the cache ($\sum_{\forall \tau_i \in \tau(H)} \sigma_i^H \leq \sigma^T$), each task has utilisation not greater than one ($U_i^H(\sigma_i^H) \leq 1$) and sum of their utilisations is less than or equal to the number of cores ($\sum_{\tau_i \in \tau(H)} U_i^H(\sigma_i^H) \leq m$). As for the set of constraints given by (18), they express the fact that $\sigma_i^H \geq \sigma_i^L,\ \forall \tau_i \in \tau(H)$. In other words, in the H-mode, an H-task may be allocated additional pages, reclaimed from the idled L-tasks, but never fewer. The reason for restricting the solution space in this manner is practical: Unlike cache pages allocated to L-tasks in the L-mode which are reclaimable immediately after a mode switch (since no L-tasks execute in the H-mode), the instant that some cache page could be taken away from an H-task is ill-defined if there is a carry-over job from that task. Even if it is assumed that a page can be taken away from that H-task, once its carry-over job completes, this would introduce an arbitrarily long (in the general case) transition to steady H-mode, in the case of a carry-over job with long outstanding execution time and even longer deadline. The schedulability analysis would then become extremely complicated, with hardly any gains expected from such a more general model.

As a final note, one might consider optimising σ_i^L and σ_i^H jointly in a single step but this would be non-trivial due to lack of a single meaningful objective function to minimise.

6 Evaluation

We experimentally explore the effectiveness of our proposed allocation heuristics and dynamic redistribution mechanism in terms of schedulability.

6.1 Experimental Setup

We developed a Java tool for our experiments. (Sources found at [3].) Its first module generates the synthetic workload (task sets). A second module implements the ILP models

Table 1 Overview of Parameters.

Parameters	Values
Task-set size (n)	{10 , 13, 15, 20}
Inter-arrival time T_i	10 to 100 msec (1 msec resol.)
Fraction of H-tasks in τ	{20%, 40%, 60%, 80%}
Ratio of C_i^H to C_i^L	{4, 6, 8, 10, 12}
Lower bound α on $C_i^L(0)/C_i^H(0)$	{0.1, 0.2, 0.4, 0.8}
Mean (λ) for x-coordinate (in pages) of the taper point (X, Y) in the prog. lockdown curve	{5, 10, 15, 20, 25, 30}
Cache size	{512 KB, 1 MB, 2 MB, 4 MB}
Number of cores (m)	{1, 2, 4, 8}
Nominal L-mode utilisation ($\frac{1}{m}\sum_{\tau_i \in \tau} U_i^L(0)$)	{0.1 : 0.1 : 1.5}
Page size	4 KB

for the allocation heuristics. Using the generated task-set and platform information as input, it partitions the cache to the tasks. A third module uses the task set, platform information and cache assignment as an input and performs the schedulability analysis and task-to-core allocation. The following parameters control the task set generation:

- We generate the L-mode task utilisations with zero locked pages ($U_i^L(0)$) for a given target task set L-mode utilisation ($\sum_{i \in \tau} U_i^L(0)$) using UUnifast-discard [9, 12], for unbiased distribution of task utilisations.
- Task periods are generated with a log-uniform distribution in the range 10-100 ms. We also assume implicit deadlines, even if our analysis holds for constrained deadlines.
- The L-mode progressive lockdown curve of a task τ_i is derived as follows. $C_i^L(0)$ is obtained as $U_i^L(0) \cdot T_i$. Then the L-WCET with full cache ($C_i^L(\sigma^T)$) is randomly generated with uniform distribution over $[\alpha \cdot C_i^L(0), C_i^L(0)]$, where $\alpha < 1$ is a user-defined parameter. Then we add a "bending point" with random coordinates (X,Y). X is sampled from a Poisson distribution with median λ (user-defined parameter) and Y is sampled from a uniform distribution in the range $[C_i^L(\sigma^T),\ Z]$, where Z is the y-coordinate of the point where the $x = X$ axis intersects the line $((0, C_i^L(0)),\ (\sigma^T, C_i^L(\sigma^T)))$. See Figure 4 for an illustration. The two linear segments $((0, C_i^L(0)),\ (X, Y))$ and $((X, Y),\ (\sigma^T, C_i^L(\sigma^T)))$, joined at an angle at (X,Y) define our L-mode progressive lockdown curve. This generation scheme can output (i)"L-shaped" curves, where the L-WCET drops sharply with a few pages and then stays flat; (ii) flat L-WCET curves, largely insensitive to the number of pages; and (iii) in-between[2].
- Based on the target fraction of H-tasks in the task set (user-specified), a number of tasks (rounded-up) will be H-tasks. For those ones, an H-mode progressive lockdown curve is generated, by up-scaling of the respective L-mode curve. The scalefactor (multiplier) is user-specified.

For each set of input parameters, we generate 100 task sets. We use independent pseudo-random number generators for the utilisations, minimum inter-arrival times/deadlines, Poisson distribution and $C_i^L(\sigma^T)$ generation, and reuse their seeds [17].

[2] We thank Renato Mancuso, for having shared with us his empirical observations about the shapes of progressive lockdown curves.

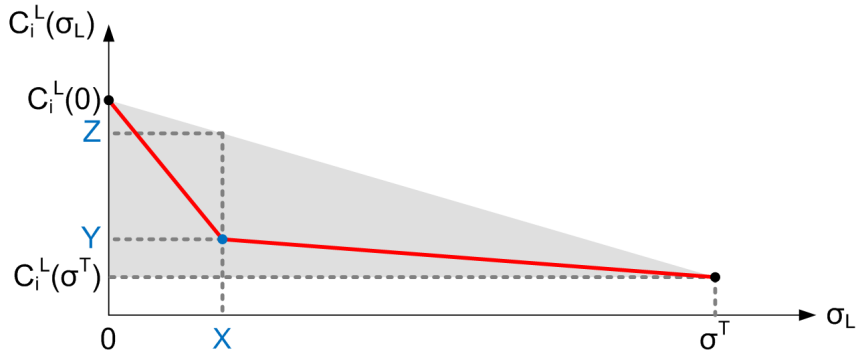


Figure 4 Illustration of progressive lockdown curve generation.

The second module of our tool models the ILP formulations for an input task set on IBM ILOG CPLEX v12.6.3 and interfaces it with the Java tool using CONCERT technology. In all experiments, the defaults for different parameters are underlined in Table 1. Note that, since the target utilisation when invoking UUnifast-discard corresponds to $\sum_{i\in\tau} U_i^L(0)$ (i.e., the L-mode task set utilisation when *none* of the tasks uses any cache), it is possible that some task sets where this nominal utilisation is greater than m may be in fact schedulable, since the allocation of cache pages to their tasks may drive down their L-WCETs and decrease the L-mode utilisation to below m. So, we explore tasks sets with nominal utilisation up to 1.5. To generate task sets with nominal utilisation greater that m, we first generate a task set with target nominal utilisation of m using UUnifast-discard, and subsequently multiply with the desired scalar. This preserves the properties of UUnifast-discard.

Finally, note that in our experiments, after the values of σ_i^L and σ_i^H are determined for the tasks, we employ First-Fit bin-packing for task-to-core assignments. The new analysis (introduced in Section 4) is used as a schedulability test, on each processor, for testing the assignments. Note that our new analysis is also used for the derivation of the deadline scalefactors for the H-tasks, using Ekberg's and Yi's (otherwise, unmodified) approach [13]. The bin-packing ordering is by decreasing criticality and decreasing deadline – a task ordering that works well with Ekberg and Yi's algorithm, as shown in our previous work [2].

6.2 Results

Since presenting plots for each possible combination of parameters would be impractical, each experiment varies only one parameter, with the rest conforming to the respective defaults from Table 1. Even so, the number of plots would still be too high to accommodate. So, instead of providing plots comparing the approaches in terms of scheduling success ratio (i.e., the fraction of task sets deemed schedulable under the respective schedulability test), we condense this information by providing plots of *weighted schedulability*.[3] This performance metric, adopted from [7], condenses what would have been three-dimensional plots into two dimensions. It is a weighted average that gives more weight to task-sets with higher utilisation, which are supposedly harder to schedule. Specifically, using the notation from [10]:

Let $S_y(\tau, p)$ represent the binary result (0 or 1) of the schedulability test y for a given task-set τ with an input parameter p. Then $W_y(p)$, the weighted schedulability for that

[3] The plots of (non-weighted) schedulability can still be found in the Appendix of our TR [1].

schedulability test y as a function p, is:

$$W_y(p) = \frac{\sum_{\forall \tau} \left(\bar{U}^L(\tau) \cdot S_y(\tau, p) \right)}{\sum_{\forall \tau} \bar{U}^L(\tau)} \,. \tag{19}$$

In (19), (adapted from [10]), $\bar{U}^L(\tau) \stackrel{\text{def}}{=} \frac{U^L(\tau)}{m}$ is the system utilisation in L-mode, normalised by the number of cores. m.

The purpose for our experiments was to quantify the schedulability improvement over a system model without cache reallocation at mode switch. However, the state-of-the-art scheduling algorithm by Ekberg and Yi, assumed for the latter, is cache-agnostic: whether the L-WCETs and H-WCETs estimates used are cache-cognisant or not is opaque to the algorithm. Therefore, in order to have a fair comparison, we needed to specify an efficient cache partitioning heuristic, even for the case of no cache reallocation.

The different curves depicted on our plots are the following:

VT: This "Validity Test", for indicative purposes, is a *necessary* condition for a task set to be mixed-criticality schedulable at all (i.e., under any possible scheduling arrangement). The actual condition, verifiable with low computational complexity, is:

$$\begin{aligned} &\left(U_i^L(\sigma^T) \leq 1,\ \forall \tau_i \in \tau\right) \ \wedge \qquad && \left(U_i^H(\sigma^T) \leq 1,\ \forall \tau_i \in \tau(H)\right) \ \wedge \\ &\left(\sum_{\tau_i \in \tau} U_i^L(\sigma^T) \leq m\right) \ \wedge && \left(\sum_{\tau_i \in \tau(H)} U_i^H(\sigma^T) \leq m\right) \end{aligned}$$

ILP: A tighter *necessary* condition, for a task set to be mixed-criticality schedulable at all. It is tested via our ILP (i.e., if it succeeds). It holds if and only if there exists an assignment of values to the σ_i^L and σ_i^H variables such that:

$$\begin{aligned} &\left(\sigma_i^L \leq \sigma^T,\ \forall \tau_i \in \tau\right) \wedge \left(\sigma_i^L \leq \sigma_i^H \leq \sigma^T,\ \forall \tau_i \in \tau(H)\right) \wedge \\ &\qquad \left(\sum_{\tau_i \in \tau} \sigma_i^L \leq \sigma^T\right) \wedge \left(\sum_{\tau_i \in \tau(H)} \sigma_i^H \leq \sigma^T\right) \wedge \\ &\left(U_i^L(\sigma_i^L) \leq 1,\ \forall \tau_i \in \tau\right) \wedge \left(U_i^H(\sigma_i^H) \leq 1,\ \forall \tau_i \in \tau(H)\right) \wedge \\ &\qquad \left(\sum_{\tau_i \in \tau} U_i^L(\sigma_i^L) \leq m\right) \wedge \left(\sum_{\tau_i \in \tau(H)} U_i^H(\sigma_i^H) \leq m\right) \end{aligned}$$

V-Ekb: Similar to "ILP", but with the added constraint that $\sigma_i^L = \sigma_i^H,\ \forall \tau_i \in \tau(H)$. Hence, it is a *necessary* condition for mixed-criticality schedulability for any approach that does not redistribute cache pages reclaimed from L-tasks to the H-tasks.

Z-Ekb: This is a *sufficient* test for partitioned scheduling using Ekberg and Yi's algorithm [13], using the specified bin-packing, if the system is crippled by disabling of the last-level cache. In that case, $\sigma_i^L = 0,\ \forall \tau_i \in \tau$ and similarly $\sigma_i^H = 0,\ \forall \tau_i \in \tau(H)$, meaning that Ekberg and Yi's original analysis is applied, with $C_i^L = C_i^L(0)$ and $C_i^H = C_i^H(0)$. Intuitively "Z-Ekb" is meant as a lower-bound for the performance by this approach, once the cache is taken into account.

E-Ekb: A *sufficient* test for partitioned scheduling using Ekberg and Yi's algorithm, using the specified bin-packing, when (i) the cache is distributed equally to the tasks in the L-mode; i.e., $\sigma_i^L = \left\lfloor \frac{\sigma^T}{n} \right\rfloor,\ \forall \tau_i \in \tau$ and (ii) there is no redistribution of cache pages, i.e., $\sigma_i^L = \sigma_i^H,\ \forall \tau_i \in \tau(H)$. Since Ekberg and Yi's algorithm is cache-agnostic, dividing the cache equally is a simple, reasonable heuristic.

Table 2 Improvement in weighted schedulability.

	Improvement (Manberg vs V-Ekb)	
Experiment / parameter varied	absolute	relative
Task-set size (n)	1.36%–2.36%	13.98%–30.59%
Fraction of H-tasks in τ	0.13%–2.34%	6.18%–16.05%
Ratio of C_i^H to C_i^L	1.41%–3.64%	10.24%–27.13%
L. bound α on $C_i^L(0)/C_i^H(0)$	0.25%–1.65%	3.36%–15.62%
λ	1.60%–2.32%	12.95%–20.25%
Cache size	1.64%–2.23%	12.67%–15.81%
Number of cores (m)	-0.00%–1.21%	-0.67%–17.45%

N-Ekb: Another *sufficient* test, but which instead uses the output of the ILP, aimed at minimising L-mode utilisation ($\sum_{\tau_i \in \tau} U_i^L(\sigma_i^L)$), as values to the σ_i^L variables. Again, there is no cache redistribution at mode change, i.e., $\sigma_i^L = \sigma_i^H$, $\forall \tau_i \in \tau(H)$. Re-using the ILP solution for the L-mode both (i) enables a fair comparison (by equipping the "opponent" with the same good heuristic for the L-mode allocation, and (ii) takes the ILP and the L-mode allocation heuristic out of the equation, as much as possible, and isolates the improvement originating from to the dynamic cache redistribution.

Manberg: A *sufficient* test for our approach (named after Mancuso and Ekberg), which redistributes cache pages at mode switch. The σ_i^L and σ_i^H variables are set to the respective outputs of the ILP, under the heuristic that picks the σ_i^L values that minimise the L-mode utilisation and, subject to that selection, the σ_i^H values that minimise the H-mode utilisation.

Note that VT theoretically dominates ILP, which in turn theoretically dominates all other curves. Additionally, V-Ekb theoretically dominates all *-Ekb curves.

As observed (Fig. 5–11), the curves of N-Ekb and V-Ekb almost coincide. This means that the heuristic of choosing the σ_i^L values that minimise the L-mode utilisation is very efficient, if cache redistribution at mode change is not permitted (bin-packing considerations aside[4]). Even when no dynamic cache reallocation is performed, the improvement just from using this particular heuristic, vs using the still reasonable "E"-heuristic is respectable (Fig. 5–11). Even if it only goes up to 10% higher weighted schedulability (and usually around 2% to 3%), it still means a large increase in the number of provably schedulable task sets. This is because in all plots except the two (Fig. 8 and 9) in which the difference between N-Ekb and E-Ekb is greatest in absolute terms, even the V-Ekb necessary test stays below 18% in weighted schedulability. As can be seen, the relative improvement of N-Ekb over E-Ekb is more significant when the execution time is more sensitive to the cache resources (Fig. 5) or the scarcer the latter are (Fig. 6).

As for our approach (Manberg), in all experiments it outperforms V-Ekb, i.e., what is theoretically possible without cache redistribution (bin-packing considerations aside). It is often nearer to the ILP curve (a necessary schedulability test, for any algorithm, with cache redistribution permitted) than to the V-Ekb curve. The absolute improvement in weighted schedulability is up to 3.64% but, in relative terms it can be up to 30.59%. This means

[4] What we mean is that our experiments cannot possibly account for all possible task assignments, since bin-packing is an NP-complete problem; we limit ourselves to just one reasonable and popular bin-packing heuristic.

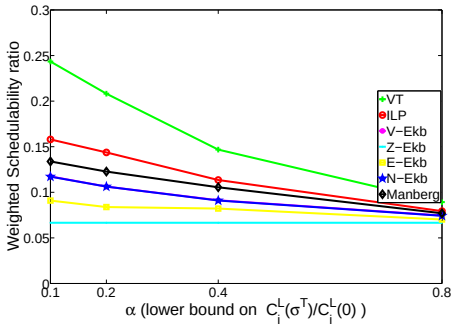


Figure 5

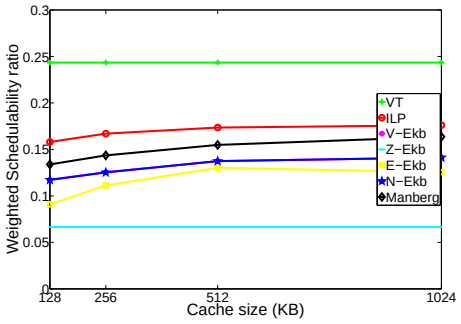


Figure 6

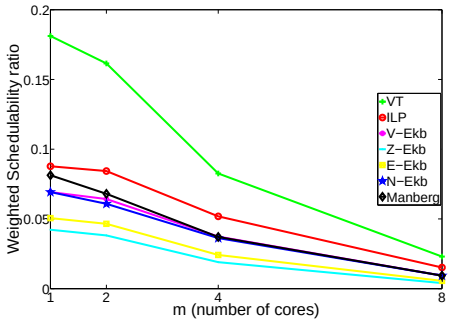


Figure 7

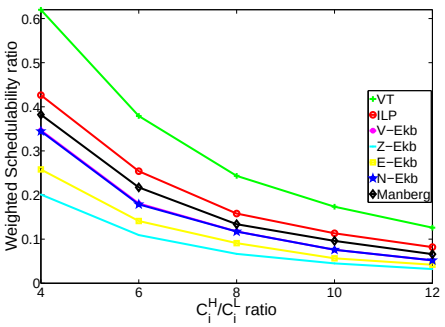


Figure 8

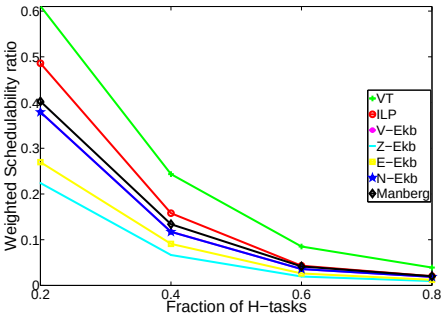


Figure 9

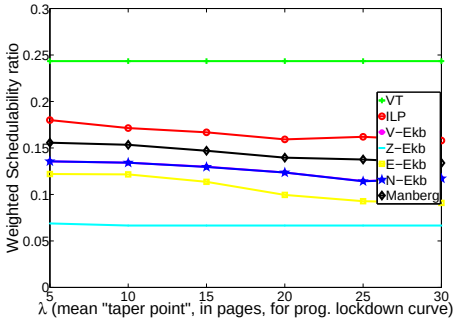


Figure 10

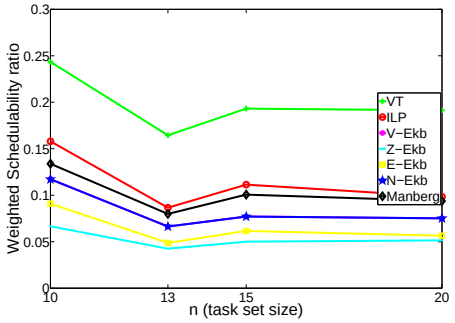


Figure 11

many more tasks sets (especially among higher-utilisation ones) found schedulable. Table 2 summarises the improvement. Note that the more sensitive to cache allocations the WCETs are, the greater the relative gains by our approach (Fig. 5). Similarly when the C_i^H/C_i^L ratio is higher, i.e., when the mode switch is harder to accommodate (Fig. 8). We believe that these findings validate our approach.

(Note that two near-zero negative values in Table 2 reflect the fact that V-Ekb is a necessary test, whose success does *not* imply that bin-packing will be feasible!)

In our experiments, the ILP run-time was a few seconds (upto four seconds for the feasible solutions), but the deadline-scaling (which repeatedly invokes the schedulability test) took 23 hours for 6000 task sets.

7 Conclusions

In this work, we proposed the redistribution of resources from low-criticality tasks to high-criticality tasks, at mode change, for better scheduling performance. Focusing on one particular resource, the last-level cache, we formulated analysis and showed the potential gains. This validates the notion that more detailed models of the platform and the allocation of its resources, can be used to improve both the performance and the confidence in the analysis of mixed-criticality systems. In the future, we plan to consider additional system resources. We also intend to explore efficient non-ILP-based cache allocation heuristics.

References

1. M. A. Awan, K. Bletsas, P. F. Souto, Benny Åkesson, and E. Tovar. Mixed-criticality scheduling with dynamic redistribution of shared cache, 2017. arXiv:1704.08876, `http://arxiv.org/abs/1704.08876`.
2. M. A. Awan, K. Bletsas, P. F. Souto, and E. Tovar. Semi-partitioned mixed-criticality scheduling. In *30th Int. Conf. on the Architecture of Computing Systems (ARCS)*, pages 205–218, 2017. `doi:10.1007/978-3-319-54999-6_16`.
3. Muhammad Ali Awan. Source code for our tool, 2017. `https://goo.gl/jNVcbJ`.
4. S. Baruah, V. Bonifaci, G. DAngelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 145–154, July 2012. `doi:10.1109/ECRTS.2012.42`.
5. Sanjoy Baruah and Alan Burns. Implementing mixed criticality systems in Ada. In *16th Ada-Europe Conference*, pages 174–188, 2011.
6. S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *11th Real-Time Systems Symposium (RTSS 1990)*, pages 182–190, Dec 1990. `doi:10.1109/REAL.1990.128746`.
7. A. Bastoni, B. B. Brandenburg, and J. H. Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. *Proceedings of OSPERT*, pages 33–44, 2010.
8. M. Behnam, R. Inam, T. Nolte, and M. Sjödin. Multi-core composability in the face of memory-bus contention. *ACM SIGBED Review*, 10(3):35–42, 2013. `doi:10.1145/2544350.2544354`.
9. E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, May 2005. `doi:10.1007/s11241-005-0507-9`.
10. A. Burns and R. I. Davis. Adaptive mixed criticality scheduling with deferred preemption. In *35th IEEE Real-Time Systems Symposium (RTSS 2014)*, pages 21–30, Dec 2014. `doi:10.1109/RTSS.2014.12`.

11 Alan Burns and Robert Davis. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep*, 2013.

12 R. I. Davis and A. Burns. Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In *30th IEEE Real-Time Systems Symposium (RTSS 2009)*, pages 398–409, Dec 2009. doi:10.1109/RTSS.2009.31.

13 P. Ekberg and W. Yi. Bounding and shaping the demand of mixed-criticality sporadic tasks. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 135–144, July 2012. doi:10.1109/ECRTS.2012.24.

14 P. Ekberg and W. Yi. Bounding and shaping the demand of generalized mixed-criticality sporadic task systems. *Journal of Real-Time Systems*, 50(1):48–86, January 2014. doi:10.1007/s11241-013-9187-z.

15 J. Flodin, K. Lampka, and W. Yi. Dynamic budgeting for settling dram contention of co-running hard and soft real-time tasks. In *9th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 151–159, June 2014. doi:10.1109/SIES.2014.6871199.

16 X. Gu and A. Easwaran. Dynamic budget management with service guarantees for mixed-criticality systems. In *37th IEEE Real-Time Systems Symposium (RTSS)*, pages 47–56, Nov 2016. doi:10.1109/RTSS.2016.014.

17 Raj Jain. *The art of computer systems performance analysis – techniques for experimental design, measurement, simulation, and modeling*. Wiley professional computing. Wiley, 1991.

18 R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2013)*, pages 45–54, April 2013. doi:10.1109/RTAS.2013.6531078.

19 R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun. WCET(m) estimation in multi-core systems using single core equivalence. In *27th Euromicro Conference on Real-Time Systems (ECRTS 2015)*, pages 174–183, July 2015. doi:10.1109/ECRTS.2015.23.

20 A. Masrur, D. Müller, and M. Werner. Bi-level deadline scaling for admission control in mixed-criticality systems. In *21st IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 100–109, 2015. doi:10.1109/RTCSA.2015.35.

21 J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *26th Euromicro Conf. on Real-Time Systems (ECRTS)*, pages 109–118, 2014. doi:10.1109/ECRTS.2014.20.

22 R. Pellizzoni and H. Yun. Memory servers for multicore systems. In *22nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2016)*, pages 97–108, April 2016. doi:10.1109/RTAS.2016.7461339.

23 J. Reineke and J. Doerfert. Architecture-parametric timing analysis. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 189–200, 2014.

24 Lui Sha, Marco Caccamo, Renato Mancuso, Jung-Eun Kim, Man-Ki Yoon, Rodolfo Pellizzoni, Heechul Yun, Russel Kegley, Dennis Perlman, Greg Arundale, Bradford Richard, et al. Single core equivalent virtual machines for hard real—time computing on multicore processors. Technical report, Univ. of Illinois at Urbana Champaign, 2014.

25 P. K. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *22nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, April 2016. doi:10.1109/RTAS.2016.7461361.

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, Dec 2007. doi:10.1109/RTSS.2007.47.

27 H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *20th IEEEReal-Time and*

Embedded Technology and Applications Symposium (RTAS 2014), pages 155–166, April 2014. doi:10.1109/RTAS.2014.6925999.

28 H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *24th Euromicro Conference on Real-Time Systems (ECRTS 2012)*, pages 299–308, July 2012. doi:10.1109/ECRTS.2012.32.

29 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 *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2013)*, pages 55–64, April 2013. doi:10.1109/RTAS.2013.6531079.

30 H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Transactions on Computers*, 65(2):562–576, Feb 2016. doi:10.1109/TC.2015.2425889.

Improving the Quality-of-Service for Scheduling Mixed-Criticality Systems on Multiprocessors*

Risat Mahmud Pathan

Chalmers University of Technology, Göteborg, Sweden
risat@chalmers.se

Abstract

The traditional Vestal's model of Mixed-Criticality (MC) systems was recently extended to Imprecise Mixed-Critical task model (IMC) to guarantee some minimum level of (degraded) service to the low-critical tasks even after the system switches to the high-critical behavior. This paper extends the IMC task model by associating specific Quality-of-Service (QoS) values with the low-critical tasks and proposes a fluid-based scheduling algorithm, called MCFQ, for such task model. The MCFQ algorithm allows some low-critical tasks to provide full service even during the high-critical behavior so that the QoS of the overall system is increased. To the best of our knowledge MCFQ is the first algorithm for IMC task sets considering multiprocessor platform and QoS values.

By extending the recently proposed MC-Fluid and MCF fluid-based multiprocessor scheduling algorithms for IMC task model, empirical results show that MCFQ algorithm can significantly improve the QoS of the system in comparison to that of both MC-Fluid and MCF. In addition, the schedulability performance of MCFQ is very close to the optimal MC-Fluid algorithm. Finally, we prove that the MCFQ algorithm has a speedup bound of 4/3, which is optimal for IMC tasks.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems, D.4.1 Scheduling

Keywords and phrases mixed-criticality systems, real-time systems, multiprocessor scheduling, quality of service, imprecise computation

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.19

1 Introduction

The computation power of multicore processors offers real-time embedded system designers the opportunity to integrate multiple components with different levels of criticality on a common hardware platform. Such Mixed-Criticality (MC) systems are often certified by certification authorities (CAs). This paper proposes a new multiprocessor scheduling algorithm for implicit-deadline dual-criticality sporadic tasks where a task is either high critical (HI) or low critical (LO).

In a dual-criticality system, the correctness of the high-critical tasks needs to be demonstrated under rigorous (often pessimistic) assumptions. Based on Vestal's model for MC tasks [21], the worst-case execution time (WCET) of each HI-critical task, according to the assumptions of the CA, is larger than or equal to that considered by the system designer. Each high-critical task τ_i has two different WCETs: $C_i^{\mathtt{L}}$ and $C_i^{\mathtt{H}}$ where $C_i^{\mathtt{L}} \leq C_i^{\mathtt{H}}$ and the WCET of each LO-critical task τ_i is $C_i^{\mathtt{L}}$. Most of the earlier works on scheduling MC systems [5, 4, 6, 17, 10, 8] consider that if some HI-critical task does not complete execution after

* This research has been funded by the MECCA project under the ERC grant ERC-2013-AdG 340328-MECCA.

executing for at most $C_i^{\mathtt{L}}$ time units – the system is said to switch from LO- to HI-critical behavior in such case – then all the LO-critical tasks are aborted (definition of critical behavior is formally presented in Section 2). Such "abortion" of the LO critical tasks may not be acceptable, for example, in many control applications as pointed out in [14]. Moreover, the system designer considers the execution of the LO-critical tasks important to achieve the mission of the system.

Some works addressed this limitation by allowing the LO-critical tasks to provide delayed results, for example, by executing them less frequently after the system switches to HI-critical behavior (e.g., weakly hard MC task model [9], elastic MC task model [20, 19, 11]). However, such delayed results may not be acceptable for some applications that prefer to have results on time even if such results are imprecise (e.g., degraded). Based on the imprecise computation model [15, 16], some recent works [14, 7, 2] have proposed a new MC task model, called the Imprecise Mixed-Critical (IMC) task model, in which each LO-critical task is also guaranteed to provide some (degraded) service even after the system switches to the HI-critical behavior. The IMC task model considers two different WCET for each task: $C_i^{\mathtt{L}}$ and $C_i^{\mathtt{H}}$ where $C_i^{\mathtt{L}} \leq C_i^{\mathtt{H}}$ if τ_i is a HI-critical task or $C_i^{\mathtt{L}} \geq C_i^{\mathtt{H}}$ if τ_i is a LO-critical task. The works in [14, 7, 2] proposed scheduling algorithms for IMC task model in which each LO-critical task executes at least $C_i^{\mathtt{H}}$ time units (i.e., it provides imprecise or degraded service rather than "no" service) during the HI-critical behavior.

The motivation of the research presented in this paper is the observation that punishing *all* the LO-critical IMC tasks by allowing them to provide only degraded service during the HI-critical behavior may not be necessary if the computing platform has slack capacity during the HI-critical behavior. No earlier work on scheduling IMC tasks considers the possibility of executing any LO-critical task to provide full service also during the HI-critical behavior. This paper proposes the Mixed-Criticality Fluid scheduling with QoS (MCFQ) algorithm for multiprocessor platform, considering a set of implicit-deadline IMC sporadic tasks, in which some (if possible all) LO-critical tasks can provide full service also during the HI-critical behavior. Allowing some of the LO-critical tasks to always provide full service would improve the overall Quality-of-Service (QoS) of the system – making the system designers "happy".

Consider an airplane or car that switches to the HI-critical behavior during its mission and all the LO-critical tasks start to provide degraded service. Such degraded service is perceived by the pilot or the driver, for example, by observing some light blinking in the cockpit/dashboard, the entertainment system being turned off, or some kind of performance loss. The pilot or driver may be uncomfortable in such situation or could even be stressed. The MCFQ algorithm considers improving such situations by allowing some LO-critical tasks to provide full service also during the HI-critical behavior.

This paper extends the IMC task model [14, 7, 2] by associating with each LO-critical task τ_i two QoS values $V_i^{\mathtt{L}}$ and $V_i^{\mathtt{H}}$ where $V_i^{\mathtt{L}} \geq V_i^{\mathtt{H}}$. The QoS value $V_i^{\mathtt{L}}$ is set to 100% based on the interpretation that if a LO-critical task τ_i is guaranteed (e.g., based on the underlying schedulability analysis) to provide full service in all possible criticality behaviors of the system, the QoS value that task τ_i provides is 100%; otherwise, the QoS value of τ_i is $V_i^{\mathtt{H}}$, which is smaller than or equal to $V_i^{\mathtt{L}}$ (the way the system designer sets these values will be presented in Section 2). The QoS values of all the high-critical tasks are assumed to be 100% for all criticality behaviors since no degradation in terms of their service is acceptable in any criticality behavior. Based on the outcome of the underlying schedulability analysis for a given MC task set, the QoS contribution of each task can be determined which in turn determines the QoS of the overall system.

The proposed MCFQ algorithm is based on a fluid-based scheduling model [12, 3, 2] in which each task τ_i has two execution rates $\theta_i^{\mathtt{L}}$ and $\theta_i^{\mathtt{H}}$ for executing task τ_i during the LO- and

HI-critical behaviors, respectively. If MCFQ is successful in determining $\theta_i^{\mathtt{L}}$ and $\theta_i^{\mathtt{H}}$, then it is guaranteed that each LO-critical task τ_i provides full and degraded service during the LO- and HI-critical behaviors of the system, respectively.

The overall objective of the MCFQ algorithm (unlike other fluid-based algorithms [12, 3]) is to maximize the sum of the LO-critical execution rates of all the tasks so that less computation is required during the HI-critical behavior. This maximization potentially implies higher slack capacity during the HI-critical behavior. Such slack is exploited to increase the HI-critical execution rate $\theta_i^{\mathtt{H}}$ of some LO-critical task so that this LO-critical task provides full service also during the HI-critical behavior, and thereby, can increase the QoS of the LO-critical task τ_i by $(V_i^{\mathtt{L}} - V_i^{\mathtt{H}})$. Given an amount of slack, the LO-critical tasks for which the HI-critical execution rates can be increased are determined based on Integer Linear Programming (ILP) to maximize the overall QoS of the system while ensuring correctness. Although the proposed MCFQ algorithm is based on a fluid-based scheduling model, there are some salient features of MCFQ that make this algorithm novel with respect to the recently proposed MC-Fluid [12] and MCF [3] algorithms. Neither the MC-Fluid [12] nor the MCF [3] algorithm considers the IMC task model, and therefore, such algorithms do not allow any LO-critical task to provide any (not even degraded) service during the HI-critical behavior. While the works in [2, 14] consider the IMC task model to allow the LO-critical tasks to provide degraded service during the HI-critical behavior, these works do not consider multiprocessors. Moreover, none of the works in [2, 14] allows any LO-critical task to provide full service in HI-critical behavior even if enough processing capacity is available. Common to all these works [12, 3, 2, 14] is that none considers maximizing the utilization of the platform during the LO-critical behavior in order to gain and exploit slack capacity during the HI-critical behavior to improve the overall QoS of the system.

This paper has the following contributions.

- First, we present an extension (i.e. generalization) of the IMC task model where each LO-critical task has two QoS values depending on whether it can provide full service in all criticality behaviors or not. This new model allows the system designers to set the values of the QoS of the LO-critical tasks based on her level of "happiness" with the degraded or full service of such tasks. Based on the QoS values of the tasks, the overall QoS of the entire system can be determined.
- Second, a new algorithm, called MCFQ, is proposed for scheduling traditional IMC task systems on a multiprocessor platform. To the best of our knowledge, MCFQ is the first multiprocessor scheduling algorithm that considers the IMC task model. The main idea of developing the MCFQ scheduling algorithm, i.e., *fully* utilizing the processors during the LO-critical behavior, has the potential to be applied to other MC scheduling algorithms that are proposed in the literature to improve the overall QoS of the system.
- Third, we formulate an ILP to select some (if possible all) LO-critical tasks such that these tasks provide full service in all the criticality behaviors of the system while maximizing the overall QoS of the system.
- We compare the schedulability of MCFQ algorithm with the recently proposed MC-Fluid [12] and MCF [3] algorithms, by extending MC-Fluid and MCF for IMC tasks, using randomly generated task sets. It is found that the MCFQ scheduling algorithm has schedulability performance very close to the optimal MC-Fluid algorithm and can significantly improve the QoS of the system in comparison to *both* MC-Fluid and MCF algorithms.
- Finally, we prove that MCFQ has a speedup bound of 4/3 which is optimal for IMC tasks.

The remainder of this paper is organized as follows: Section 2 presents the system model. Section 3 presents an overview of the proposed MCFQ algorithm. The details of MCFQ algorithm

and the proof of its correctness are presented in Section 4. The formulation of the ILP to improve the QoS of the system is presented in Section 5. Experimental results are presented in Section 6 before concluding in Section 7.

2 System Model

This paper considers the scheduling of n implicit-deadline dual-criticality sporadic tasks in set $\Gamma = \{\tau_1, \ldots \tau_n\}$ on m processors. Each task $\tau_i \in \Gamma$ generates an infinite sequence of jobs. Each task τ_i is represented using 4 parameters $\{T_i, L_i, \mathcal{C}_i, \mathcal{V}_i\}$ where[1]

- $T_i \in \mathbb{R}^+$ is the minimum inter-arrival time of the jobs (also, called period) of the task. The relative deadline of task τ_i is also T_i.
- $L_i \in \{\texttt{HI}, \texttt{LO}\}$ is the criticality of the task: `LO` and `HI` respectively specifies low- and high-critical task.
- $\mathcal{C}_i = \{C_i^{\texttt{L}}, C_i^{\texttt{H}}\}$ is a list of WCETs of task τ_i at different criticality levels. The WCET of task τ_i at criticality level `LO` and `HI` are respectively $C_i^{\texttt{L}}$ and $C_i^{\texttt{H}}$. If $L_i = \texttt{HI}$, then $C_i^{\texttt{H}} \geq C_i^{\texttt{L}}$ for a `HI`-critical task, whereas $C_i^{\texttt{H}} \leq C_i^{\texttt{L}}$ for a `LO`-critical task.
- $\mathcal{V}_i = \{V_i^{\texttt{L}}, V_i^{\texttt{H}}\}$ is a list of QoS values for each `LO`-critical task τ_i where $V_i^{\texttt{L}} \geq V_i^{\texttt{H}}$. If *all* the jobs of the `LO`-critical task τ_i are guaranteed to execute for $C_i^{\texttt{L}}$ time units (i.e., it provides full service in all behaviors), then task τ_i's QoS contribution is $V_i^{\texttt{L}}$; otherwise, τ_i's QoS contribution is $V_i^{\texttt{H}}$. Although each `LO`-critical task has two different QoS values, the contribution of each such task's QoS to the overall QoS of the system is dependent on the outcome of the schedulability analysis.

How are the QoS values assigned? The system designer sets the values of $V_i^{\texttt{L}}$ and $V_i^{\texttt{H}}$ for each `LO`-critical task based on how "happy" she is with the full and degraded service of the task, respectively. If each of the jobs of a `LO`-critical task τ_i executes for $C_i^{\texttt{L}}$ time units, then task τ_i provides full service in all the criticality behaviors and the QoS value $V_i^{\texttt{L}}$ of τ_i is 100%, i.e., $V_i^{\texttt{L}} = 1.0$. On the other hand, the value of $V_i^{\texttt{H}}$ should reflect the level of degradation of the `LO`-critical task τ_i if all the jobs of such a task cannot be guaranteed to execute for $C_i^{\texttt{L}}$ time units in `HI`-critical behavior. Note that although the `HI`-critical behavior does not necessarily require a `LO`-critical task to execute $C_i^{\texttt{L}}$ time units to ensure correctness (definition of correctness will be presented shortly), this paper seeks the opportunity to do so in order to improve the overall QoS of the system.

If the output quality of a `LO`-critical task τ_i depends on how long the task executes (i.e, the longer a task executes, the better results it produces similar to the imprecise computation models [15, 16]), then the system designer sets the QoS value $V_i^{\texttt{H}}$ of a `LO`-critical task τ_i as $V_i^{\texttt{H}} = C_i^{\texttt{H}}/C_i^{\texttt{L}}$. Note that $C_i^{\texttt{H}}/C_i^{\texttt{L}} \leq 1$ for `LO`-critical task τ_i. On the other hand, if the output quality of a task is not directly related to how long the task executes, then value of $V_i^{\texttt{H}}$ is assigned by the system designer based on her own interpretation (i.e., happiness) regarding the level of degradation of the `LO`-critical task τ_i. The system designer assigns $V_i^{\texttt{H}} = hpy_i$ where hpy_i is her level of happiness with the degraded service of the `LO`-critical task τ_i where $V_i^{\texttt{H}} = hpy_i \leq V_i^{\texttt{L}} = 1.0$.

Useful Definitions: The set of all the `HI`-critical tasks in Γ is denoted by Γ_H where $\Gamma_H = \{\tau_i \mid \tau_i \in \Gamma \text{ and } L_i = \texttt{HI}\}$. Similarly, the set of all the `LO`-critical tasks in Γ is denoted

[1] Each `HI`-critical task is represented using 3 parameters since the required QoS values for such tasks is always 100%.

by Γ_L where $\Gamma_L = \{\tau_i \mid \tau_i \in \Gamma \text{ and } L_i = \texttt{LO}\}$. The LO and HI-critical utilization of task τ_i are defined as $u_i^{\texttt{L}} = C_i^{\texttt{L}}/T_i$ and $u_i^{\texttt{H}} = C_i^{\texttt{H}}/T_i$. For all LO-critical tasks, the total LO- and HI-critical utilizations are $U_{\texttt{L}}^{\texttt{L}} = \sum_{\forall \tau_i \in \Gamma_L} u_i^{\texttt{L}}$ and $U_{\texttt{L}}^{\texttt{H}} = \sum_{\forall \tau_i \in \Gamma_L} u_i^{\texttt{H}}$, respectively. Similarly, for all HI-critical tasks, the total LO- and HI-critical utilizations are $U_{\texttt{H}}^{\texttt{L}} = \sum_{\forall \tau_i \in \Gamma_H} u_i^{\texttt{L}}$ and $U_{\texttt{H}}^{\texttt{H}} = \sum_{\forall \tau_i \in \Gamma_H} u_i^{\texttt{H}}$, respectively.

Behavior: An MC sporadic task system shows different behaviors during different runs of the system since different jobs may be released at different time instants and may have different execution times. We assume, similar to [2], that the run-time environment budgets the execution time of the jobs generated by the LO-critical tasks such that any such job will be terminated once it consumes its budgeted amount of execution, regardless of whether it has completed execution or not. The criticality level of a behavior is determined by how much execution is needed by the HI-critical jobs to complete execution in that behavior.

If each HI-critical job of task τ_i signals completion after completing at most $C_i^{\texttt{L}}$ units of execution, then the behavior of the system is defined to be a *LO-critical behavior*. If some job of a HI-critical task τ_i does not signal completion after completing at most $C_i^{\texttt{L}}$ units of execution at time t, then the system is said to *switch* from LO- to HI-critical behavior at time t. If each job of a HI-critical task τ_i signals completion after completing at most $C_i^{\texttt{H}}$ units of execution, then the behavior of the system is defined to be a *HI-critical behavior*. All other behaviors are erroneous.

Correctness: We define an algorithm for scheduling an MC system to be correct if it is able to schedule any system in such a manner that both the following properties are satisfied:

- During all the LO-critical behaviors of the system, each HI-critical job receives enough execution between its release time and deadline to complete, and each LO-critical job either completes or receives at least its LO-critical WCET, between its release time and deadline.
- During all the HI-critical behaviors of the system, each HI-critical job receives enough execution between its release time and deadline to complete, and each LO-critical job of a LO-critical task either completes or receives **at least** its HI-critical WCET, between its release time and deadline.

The proposed MCFQ algorithm first seeks to find execution rates of the tasks to ensure the correctness of the system. As it is evident from the definition of correctness that if a system is correct, then it is ensured that each LO-critical task provides degraded service during the HI-critical behavior and contributes a QoS of $V_i^{\texttt{H}}$ to the overall QoS of the system. Given that a system is correct, we exploit slack of the processors in HI-critical behavior to select some LO-critical tasks so that these tasks are guaranteed to provide full service even during the HI-critical behavior – improving the QoS of the LO-critical task τ_i from $V_i^{\texttt{H}}$ to $V_i^{\texttt{L}}$.

3 An overview of MCFQ Scheduling Algorithm

The MCFQ algorithm is based on fluid-based scheduling [12, 3] in which a task may be assigned a fraction ≤ 1 of a processor, called the *execution rate* of the task, at each time instant. The MCFQ algorithm prior to runtime determines the LO- and HI-critical execution rates, denoted respectively by $\theta_i^{\texttt{L}}$ and $\theta_i^{\texttt{H}}$, for each task $\tau_i \in \Gamma$. The execution rates $\theta_i^{\texttt{L}}$ and $\theta_i^{\texttt{H}}$ for each task τ_i are computed such that the run-time scheduling strategy presented in Figure 1 constitutes a correct scheduling strategy for task set Γ. According to the algorithm in Figure 1, each task

- Each task τ_i initially executes at a constant rate $\theta_i^{\texttt{L}}$. That is, at each time instant it is executing upon $\theta_i^{\texttt{L}}$ fraction of a processor.
- If a job of task $\tau_i \in \Gamma_H$ does not complete despite having received $C_i^{\texttt{L}}$ units of execution (equivalently, having executed for a duration $C_i^{\texttt{L}}/\theta_i^{\texttt{L}}$), then each task τ_i executes at a constant rate $\theta_i^{\texttt{H}}$. That is, at each time instant it is executing upon $\theta_i^{\texttt{H}}$ fraction of a processor.

Figure 1 Run-time scheduling strategy originally proposed in [2] for uniprocessor is also applicable to multiprocessors.

τ_i is executed with execution rate $\theta_i^{\texttt{L}}$ during the LO-critical behavior of the system. Once the system switches to HI-critical behavior, τ_i executes with execution rate $\theta_i^{\texttt{H}}$.

The system can switch back (not considered in this paper) from HI- to LO-critical behavior when there is an idle period and no job of any HI-critical task awaits for execution as is proposed by Santy et al [18]. Transforming the fluid schedule generated by MCFQ algorithm to construct a (non-fluid) schedule for real hardware can be done using the MC-DP-Fair algorithm proposed in [12].

We present the execution-rate assignment strategy of MCFQ in Figure 2 in Section 4. It will be proved in subsection 4.1 that if the MCFQ algorithm successfully determines the execution rates $\theta_i^{\texttt{L}}$ and $\theta_i^{\texttt{H}}$, then the system is correct. Given that an MC system is correct using the execution rates determined by the MCFQ algorithm in Figure 2, we then consider increasing the HI-critical execution rates $\theta_i^{\texttt{H}}$ of some LO-critical tasks to increase the QoS of the system in Section 5. The following lemmas and definitions will be used in Section 4.

Lemma 1, derived in [12], states a necessary and sufficient schedulability condition of a HI-critical task τ_k during the HI-critical behavior (including the particular scenario when the system switches from LO- to HI-critical behavior) assuming that the task is schedulable in LO-critical behavior. The condition in Eq. (1) is derived regardless of any parameters of the LO-critical tasks and is thus also applicable to IMC task systems.

▶ **Lemma 1** (From Lemma 5 in [12]). *Given a HI-critical task τ_k satisfying task-schedulability in LO-critical behavior, the task can meet its deadline if and only if*

$$u_k^L/\theta_k^L + (u_k^H - u_k^L)/\theta_k^H \;\leq\; 1 \tag{1}$$

Based on Lemma 1, a lower bound on $\theta_k^{\texttt{L}}$ for each HI-critical task $\tau_k \in \Gamma_H$ is given in Lemma 2.

▶ **Lemma 2.** *If the execution rates θ_k^L and θ_k^H of a HI-critical task $\tau_k \in \Gamma_H$ guarantees that all the jobs of τ_k meet their deadlines in all the correct behaviors of the system, then the following holds:*

$$\theta_k^L \;\geq\; u_k^L/(1 - u_k^H + u_k^L) \;\geq\; u_k^L \tag{2}$$

Proof. Since $\tau_k \in \Gamma_H$ meets its deadline, the following (from Eq. (1) of Lemma 1) holds:

$$\begin{aligned}
& u_k^{\texttt{L}}/\theta_k^{\texttt{L}} + (u_k^{\texttt{H}} - u_k^{\texttt{L}})/\theta_i^{\texttt{H}} \;\leq\; 1 \\
\Rightarrow\; & u_k^{\texttt{L}}/\theta_k^{\texttt{L}} + (u_k^{\texttt{H}} - u_k^{\texttt{L}}) \;\leq\; 1 && \text{(Since } \theta_k^{\texttt{H}} \leq 1 \text{ for any execution rate to be valid)} \\
\Leftrightarrow\; & \theta_k^{\texttt{L}} \;\geq\; u_k^{\texttt{L}}/(1 - u_k^{\texttt{H}} + u_k^{\texttt{L}})
\end{aligned}$$

For a HI-critical task τ_k, we have $0 \leq (1 - u_k^{\texttt{H}} + u_k^{\texttt{L}}) \leq 1$ because $1 \geq u_k^{\texttt{H}} \geq u_k^{\texttt{L}}$. Therefore, $u_i^{\texttt{L}}/(1 - u_k^{\texttt{H}} + u_k^{\texttt{L}}) \geq u_k^{\texttt{L}}$ and we have $\theta_k^{\texttt{L}} \;\geq\; u_k^{\texttt{L}}/(1 - u_k^{\texttt{H}} + u_k^{\texttt{L}}) \;\geq\; u_k^{\texttt{L}}$. ◀

Assumptions: $(\overline{U}^{\mathtt{L}}_{\mathtt{H}} + U^{\mathtt{L}}_{\mathtt{L}}) \leq m$, $(U^{\mathtt{H}}_{\mathtt{H}} + U^{\mathtt{H}}_{\mathtt{L}}) \leq m$, $max\{u^{\mathtt{H}}_i, u^{\mathtt{L}}_i\} \leq 1$ for all $\tau_i \in \Gamma$

1. $\theta^{\mathtt{H}}_i = u^{\mathtt{H}}_i$ and $\theta^{\mathtt{L}}_i = u^{\mathtt{L}}_i$ for all $\tau_i \in \Gamma_L$.
2. For $i = 1$ to h //Tasks in Γ_H are indexed from $1 \ldots h$

$$\theta^{\mathtt{L}}_i = min\{u^{\mathtt{H}}_i, \mathcal{F}_{i-1} \cdot \overline{u}^{\mathtt{L}}_i\} \tag{5}$$

$$\theta^{\mathtt{H}}_i = (u^{\mathtt{H}}_i - u^{\mathtt{L}}_i)/(1 - \frac{u^{\mathtt{L}}_i}{\theta^{\mathtt{L}}_i}) \tag{6}$$

3. If

$$\sum_{\tau_k \in \Gamma} \theta^{\mathtt{H}}_k \leq m \textbf{ and } \sum_{\tau_k \in \Gamma} \theta^{\mathtt{L}}_k \leq m \tag{7}$$

then declare success else declare failure.

Figure 2 Execution Rate Assignment.

Lemma 2 essentially states that if an MC task set is schedulable in all the correct behaviors of the system based on the runtime scheduling strategy in Figure 1, then it is **necessary** that the LO-critical execution rate $\theta^{\mathtt{L}}_k$ for each HI-critical task $\tau_k \in \Gamma_H$ must not be smaller than $\frac{u^{\mathtt{L}}_k}{1-u^{\mathtt{H}}_k+u^{\mathtt{L}}_k}$. We denote by $\overline{u}^{\mathtt{L}}_k$ the lower bound on the LO-critical execution rate of the HI-critical task $\tau_k \in \Gamma_H$:

$$\overline{u}^{\mathtt{L}}_k = u^{\mathtt{L}}_k/(1 - u^{\mathtt{H}}_k + u^{\mathtt{L}}_k)\,. \tag{3}$$

We also define $\overline{U}^{\mathtt{L}}_{\mathtt{H}}$ as follows:

$$\overline{U}^{\mathtt{L}}_{\mathtt{H}} = \sum_{\tau_k \in \Gamma_H} \overline{u}^{\mathtt{L}}_k = \sum_{\tau_k \in \Gamma_H} \frac{u^{\mathtt{L}}_k}{1 - u^{\mathtt{H}}_k + u^{\mathtt{L}}_k}\,. \tag{4}$$

4 Execution Rate Assignment of MCFQ

If $(U^{\mathtt{H}}_{\mathtt{H}} + U^{\mathtt{H}}_{\mathtt{L}}) > m$ or $(U^{\mathtt{L}}_{\mathtt{L}} + \overline{U}^{\mathtt{L}}_{\mathtt{H}}) > m$ or $max\{u^{\mathtt{H}}_k, u^{\mathtt{L}}_k\} > 1$ for any task $\tau_k \in \Gamma$, then no algorithm can schedule such a task set. The MCFQ algorithm considers task sets where $(U^{\mathtt{H}}_{\mathtt{H}} + U^{\mathtt{H}}_{\mathtt{L}}) \leq m$ and $(U^{\mathtt{L}}_{\mathtt{L}} + \overline{U}^{\mathtt{L}}_{\mathtt{H}}) \leq m$ and $max\{u^{\mathtt{H}}_k, u^{\mathtt{L}}_k\} \leq 1$ for each task $\tau_k \in \Gamma$.

The execution rate assignment algorithm of MCFQ is given in Figure 2. The LO- and the HI-critical execution rates $\theta^{\mathtt{L}}_i$ and $\theta^{\mathtt{H}}_i$ of **each LO-critical task** is equal to its LO- and HI-critical utilizations $u^{\mathtt{L}}_i$ and $u^{\mathtt{H}}_i$, respectively. In Step 1 of the algorithm in Figure 2, we assign $\theta^{\mathtt{L}}_i = u^{\mathtt{L}}_i$ and $\theta^{\mathtt{H}}_i = u^{\mathtt{H}}_i$ for each LO-critical task $\tau_i \in \Gamma_L$.

The LO- and HI-critical execution rates to each **HI-critical task** is assigned in Step 2 in Figure 2. Let the HI-critical tasks in set Γ_H are indexed[2] from 1 to h. The value of the LO-critical execution rate $\theta^{\mathtt{L}}_i$ of a HI-critical task $\tau_i \in \Gamma_H$ is assigned in Eq. (5) such that $\theta^{\mathtt{L}}_i = min\{u^{\mathtt{H}}_i, \mathcal{F}_{i-1} \cdot \overline{u}^{\mathtt{L}}_i$ based on a *threshold*, denoted by $\mathcal{F}_{i-1}$, which is defined in Eq. (8). Notice that the value $\theta^{\mathtt{L}}_i$ of the i^{th} HI-critical task $\tau_i \in \Gamma_H$ is assigned based on the

[2] The execution rate-assignment algorithm in Figure 2 works for any arbitrary order of considering the HI-critical tasks when assigning their execution rates in Step 2. However, sorting the HI-critical tasks in increasing order of $u^{\mathtt{H}}_i/\overline{u}^{\mathtt{L}}_i$ has schedulability performance very close to the optimal MC-Fluid algorithm (shown in Section 6).

Table 1 An example dual-criticality IMC taskset.

τ_i	T_i	$C_i^{\mathtt{L}}$	$C_i^{\mathtt{H}}$	L_i	$u_i^{\mathtt{L}}$	$u_i^{\mathtt{H}}$	$\overline{u}_i^{\mathtt{L}}$	$\mathcal{F}_{i-1}$	$V_i^{\mathtt{H}}$
τ_1	20	7	13	HI	0.35	0.65	0.5	13/9	-
τ_2	10	2	7	HI	0.2	0.7	0.4	13/8	-
τ_3	40	8	5	LO	0.2	0.125	-	-	0.6
τ_4	60	30	12	LO	0.5	0.2	-	-	0.4

$(i-1)^{th}$ threshold (i.e, $\mathcal{F}_{i-1}$) for $i = 1, \ldots h$. After the value of $\theta_i^{\mathtt{L}}$ is assigned using Eq. (5), the HI-critical execution rate $\theta_i^{\mathtt{H}}$ is assigned in Eq. (6) based on the value of $\theta_i^{\mathtt{L}}$ such that $\theta_i^{\mathtt{H}} = (u_i^{\mathtt{H}} - u_i^{\mathtt{L}})/(1 - u_i^{\mathtt{L}}/\theta_i^{\mathtt{L}})$.

If $\sum_{\tau_k \in \Gamma} \theta_k^{\mathtt{H}} \leq m$ and $\sum_{\tau_k \in \Gamma} \theta_k^{\mathtt{L}} \leq m$, we declare success; otherwise, we declare failure in Step 3. We will prove in subsection 4.1 that if the algorithm declares success, then the run-time scheduling strategy in Figure 1 constitutes a correct scheduling strategy.

Threshold $\mathcal{F}_i$: The value of threshold $\mathcal{F}_i$ in Eq. (8) for task τ_{i+1} is derived as follows. Initially, $\mathcal{F}_0 = (m - U_{\mathtt{L}}^{\mathtt{L}})/\overline{U}_{\mathtt{H}}^{\mathtt{L}}$. Task $\tau_1 \in \Gamma_H$ in Eq. (5) uses $\mathcal{F}_0$. The value of $\mathcal{F}_i$ for task $\tau_{i+1} \in \Gamma_H$ is recursively defined as follows for $i = 1, \ldots (h-1)$:

$$\mathcal{F}_i = \begin{cases} \frac{m - U_{\mathtt{L}}^{\mathtt{L}}}{\overline{U}_{\mathtt{H}}^{\mathtt{L}}} & \text{if } i = 0 \\ max\{\mathcal{F}_{i-1}, \frac{m - U_{\mathtt{L}}^{\mathtt{L}} - \sum_{k=1}^{i} u_k^{\mathtt{H}}}{(\overline{U}_{\mathtt{H}}^{\mathtt{L}} - \sum_{k=1}^{i} \overline{u}_k^{\mathtt{L}})}\} & \text{if } i > 0 \end{cases} \tag{8}$$

▶ **Example 3.** Consider the task set in Table 1 where $m = 2$. The last column shows the $V_i^{\mathtt{H}}$ values for the two LO-critical tasks τ_3 and τ_4 where $V_3^{\mathtt{H}} = hpy_3 = 0.6 \neq C_3^{\mathtt{H}}/C_3^{\mathtt{L}}$ and $V_4^{\mathtt{H}} = 0.4 = C_4^{\mathtt{H}}/C_4^{\mathtt{L}} = 12/30$. Note that $V_4^{\mathtt{H}}$ is assigned based on the imprecise computation model while $V_3^{\mathtt{H}}$ is not assigned based on the imprecise computation model. The values of $\overline{u}_i^{\mathtt{L}}$ and F_{i-1} that are required to compute the execution rates of the HI-critical tasks are shown in the eighth and ninth columns, respectively.

For the task set in Table 1, we have

$U_{\mathtt{H}}^{\mathtt{H}} = 0.65 + 0.7 = 1.35$ $\quad U_{\mathtt{L}}^{\mathtt{L}} = 0.2 + 0.5 = 0.7$ $\quad U_{\mathtt{L}}^{\mathtt{H}} = 0.125 + 0.2 = 0.325$

$\overline{U}_{\mathtt{H}}^{\mathtt{L}} = 0.5 + 0.4 = 0.9$ $\quad (U_{\mathtt{H}}^{\mathtt{H}} + U_{\mathtt{L}}^{\mathtt{L}}) = 1.675 \leq m$ $\quad (U_{\mathtt{L}}^{\mathtt{L}} + \overline{U}_{\mathtt{H}}^{\mathtt{L}}) = 1.6 \leq m$

Also note that $(U_{\mathtt{L}}^{\mathtt{L}} + U_{\mathtt{H}}^{\mathtt{H}}) = (1.35 + 0.7) = 2.05 > m$, which implies such a task set cannot allow both the LO-critical tasks to provide full service during the HI-critical behavior. Now we present how the values of $\mathcal{F}_{i-1}$ are computed for $i = 1, 2$ since there are two HI-critical tasks (i.e, $h = 2$).

$\mathcal{F}_0 = \frac{m - U_{\mathtt{L}}^{\mathtt{L}}}{\overline{U}_{\mathtt{H}}^{\mathtt{L}}} = \frac{2 - 0.7}{0.9} = 13/9$ and $\mathcal{F}_1 = max\{\mathcal{F}_0, \frac{m - U_{\mathtt{L}}^{\mathtt{L}} - u_1^{\mathtt{H}}}{\overline{U}_{\mathtt{H}}^{\mathtt{L}} - \overline{u}_1^{\mathtt{L}}}\} = max\{13/9, \frac{2 - 0.7 - 0.65}{0.9 - 0.5}\} = 13/8$

The LO-critical tasks τ_3 and τ_4 get values of $\theta_i^{\mathtt{L}}$ and $\theta_i^{\mathtt{H}}$ based on Step 1 in Figure 2 as follows: $\theta_3^{\mathtt{L}}{=}u_3^{\mathtt{L}}{=}0.2$, $\theta_3^{\mathtt{H}}{=}u_3^{\mathtt{H}}{=}0.125$ and $\theta_4^{\mathtt{L}}{=}u_4^{\mathtt{L}}{=}0.5$, and $\theta_4^{\mathtt{H}}{=}u_4^{\mathtt{H}}{=}0.2$. The HI-critical tasks τ_1 and τ_2 get values of $\theta_i^{\mathtt{L}}$ based on Eq. (5) as follows:

$\theta_1^{\mathtt{L}} = min\{u_1^{\mathtt{H}}, \mathcal{F}_0 \cdot \overline{u}_1^{\mathtt{L}}\} = min\{0.65, 13/9 \times 0.5\} = 0.65$

$\theta_2^{\mathtt{L}} = min\{u_2^{\mathtt{H}}, \mathcal{F}_1 \cdot \overline{u}_2^{\mathtt{L}}\} = min\{0.7, 13/8 \times 0.4\} = 0.65$

The HI-critical tasks τ_1 and τ_2 get values of $\theta_i^{\mathtt{H}}$ based on Eq. (6) as follows:

$\theta_1^{\mathtt{H}} = \frac{u_1^{\mathtt{H}} - u_1^{\mathtt{L}}}{1 - \frac{u_1^{\mathtt{L}}}{\theta_1^{\mathtt{L}}}} = \frac{0.65 - 0.35}{1 - \frac{0.35}{0.65}} = 13/20$ $\qquad \theta_2^{\mathtt{H}} = \frac{u_2^{\mathtt{H}} - u_2^{\mathtt{L}}}{1 - \frac{u_2^{\mathtt{L}}}{\theta_2^{\mathtt{L}}}} = \frac{0.7 - 0.2}{1 - \frac{0.2}{0.65}} = 13/18$

Since $\sum_{i=1}^{4} \theta_i^{\mathtt{L}} = 0.65 + 0.65 + 0.2 + 0.5 = 2 \leq m$ and $\sum_{i=1}^{4} \theta_i^{\mathtt{H}} = 13/20 + 13/18 + 0.125 + 0.2 = 1.6972 \leq m$, we conclude that the MCFQ algorithm returns success.

Note that the sum of the LO-critical execution rates is m (i.e., full capacity of the platform) while the sum of the HI-critical execution rates is 1.6972. The slack capacity in HI-critical behavior is $(m - 1.6972) = 0.3027$. ◀

To prove the correctness of the MCFQ algorithm, the following lemmas will be used.

▶ **Lemma 4.** *Consider the tasks in Γ_H are indexed from 1 to h. If Γ is feasible, then*

$$1 \leq \mathcal{F}_0 \leq \mathcal{F}_1 \leq \ldots \leq \mathcal{F}_{h-1}\,. \tag{9}$$

Proof. We prove this lemma using induction on $i = 0, 1, \ldots (h-1)$. Since Γ is feasible, it is necessary that $(U_{\mathtt{L}}^{\mathtt{L}} + \overline{U}_{\mathtt{H}}^{\mathtt{L}}) \leq m$. In other words, $\mathcal{F}_0 = \frac{m - U_{\mathtt{L}}^{\mathtt{L}}}{\overline{U}_{\mathtt{H}}^{\mathtt{L}}} \geq 1$. Now assume that $1 \leq \mathcal{F}_0 \leq \mathcal{F}_1 \ldots \leq \mathcal{F}_{i-1}$ for some i where $i < (h-1)$. From Eq. (8), we have

$$\mathcal{F}_i = max\{\mathcal{F}_{i-1}, \frac{m - U_{\mathtt{L}}^{\mathtt{L}} - \sum_{k=1}^{i} u_k^{\mathtt{H}}}{(\overline{U}_{\mathtt{H}}^{\mathtt{L}} - \sum_{k=1}^{i} \overline{u}_k^{\mathtt{L}})}\} \geq \mathcal{F}_{i-1}\,.$$

Therefore, we have $1 \leq \mathcal{F}_0 \leq \mathcal{F}_1 \ldots \leq \mathcal{F}_{i-1} \leq \mathcal{F}_i$. Consequently, Eq. (9) holds. ◀

▶ **Lemma 5.** *Consider a LO-critical task $\tau_i \in \Gamma_L$. We have $u_i^L \leq \theta_i^L \leq 1$ and $u_i^H \leq \theta_i^H \leq 1$.*

Proof. For each LO-critical task $\tau_i \in \Gamma_L$, we have $\theta_i^{\mathtt{L}} = u_i^{\mathtt{L}}$ and $\theta_i^{\mathtt{H}} = u_i^{\mathtt{H}}$ according to Step 1 in Figure 2. Since $u_i^{\mathtt{H}} \leq 1$ and $u_i^{\mathtt{L}} \leq 1$ (necessary conditions for schedulability), we also have that $u_i^{\mathtt{L}} \leq \theta_i^{\mathtt{L}} \leq 1$ and $u_i^{\mathtt{H}} \leq \theta_i^{\mathtt{H}} \leq 1$ for all $\tau_i \in \Gamma_L$. ◀

▶ **Lemma 6.** *Consider a HI-critical task $\tau_i \in \Gamma_H$. We have We have $u_i^L \leq \theta_i^L \leq 1$ and $u_i^H \leq \theta_i^H \leq 1$.*

Proof. From Eq. (5), we have $\theta_i^{\mathtt{L}} = min\{u_i^{\mathtt{H}}, \mathcal{F}_{i-1} \cdot \overline{u}_i^{\mathtt{L}}\}$ for $\tau_i \in \Gamma_H$. We will prove this lemma considering two cases: case (i) $\theta_i^{\mathtt{L}} = u_i^{\mathtt{H}}$, and case (ii) $\theta_i^{\mathtt{L}} = \mathcal{F}_{i-1} \cdot \overline{u}_i^{\mathtt{L}}$.

Case (i): $\boldsymbol{\theta_i^{\mathtt{L}} = u_i^{\mathtt{H}}}$. In such case, from Eq. (6) we have

$$\theta_i^{\mathtt{H}} = (u_i^{\mathtt{H}} - u_i^{\mathtt{L}})/(1 - \frac{u_i^{\mathtt{L}}}{\theta_i^{\mathtt{L}}}) = (u_i^{\mathtt{H}} - u_i^{\mathtt{L}})/(1 - \frac{u_i^{\mathtt{L}}}{u_i^{\mathtt{H}}}) = u_i^{\mathtt{H}}\,. \tag{10}$$

Therefore, $\theta_i^{\mathtt{L}} = \theta_i^{\mathtt{H}} = u_i^{\mathtt{H}}$. Since $1 \geq u_i^{\mathtt{H}} \geq u_i^{\mathtt{L}}$ for a HI-critical task $\tau_i \in \Gamma_H$, we have $1 \geq \theta_i^{\mathtt{L}} \geq u_i^{\mathtt{L}}$ and $1 \geq \theta_i^{\mathtt{H}} \geq u_i^{\mathtt{H}}$.

Case (ii): $\boldsymbol{\theta_i^{\mathtt{L}} = \mathcal{F}_{i-1} \cdot \overline{u}_i^{\mathtt{L}}}$. Since $\theta_i^{\mathtt{L}} = min\{u_i^{\mathtt{H}}, \mathcal{F}_{i-1} \cdot \overline{u}_i^{\mathtt{L}}\}$ in Eq. (5) and $\theta_i^{\mathtt{L}} = \overline{u}_i^{\mathtt{L}} \cdot \mathcal{F}_{i-1}$ for this case, it follows that $\theta_i^{\mathtt{L}} = \mathcal{F}_{i-1} \cdot \overline{u}_i^{\mathtt{L}} \leq u_i^{\mathtt{H}} \leq 1$. Because $\mathcal{F}_{i-1} \geq 1$ according to Eq. (9), we have $\theta_i^{\mathtt{L}} = \mathcal{F}_{i-1} \cdot \overline{u}_i^{\mathtt{L}} \geq \overline{u}_i^{\mathtt{L}}$. Consequently, the following holds

$$1 \geq u_i^{\mathtt{H}} \geq \theta_i^{\mathtt{L}} = (\overline{u}_i^{\mathtt{L}} \cdot \mathcal{F}_{i-1}) \geq \overline{u}_i^{\mathtt{L}}\ . \tag{11}$$

Since $0 < (1 - u_i^{\mathtt{H}} + u_i^{\mathtt{L}}) \leq 1$ for a HI-critical task $\tau_i \in \Gamma_H$, from Eq. (3) we have $\overline{u}_i^{\mathtt{L}} = u_i^{\mathtt{L}}/(1 - u_i^{\mathtt{H}} + u_i^{\mathtt{L}}) \geq u_i^{\mathtt{L}}$. Then from Eq. (11) and based on the fact that $\overline{u}_i^{\mathtt{L}} \geq u_i^{\mathtt{L}}$, it follows that $1 \geq \theta_i^{\mathtt{L}} \geq u_i^{\mathtt{L}}$ for this case. Now we will show that $1 \geq \theta_i^{\mathtt{H}} \geq u_i^{\mathtt{H}}$ holds. From Eq. (6) we

have

$$\theta_i^{\mathtt{H}} = (u_i^{\mathtt{H}} - u_i^{\mathtt{L}})/(1 - \frac{u_i^{\mathtt{L}}}{\theta_i^{\mathtt{L}}})$$

(From Eq. (11), $u_i^{\mathtt{H}} \geq \theta_i^{\mathtt{L}} \geq \overline{u}_i^{\mathtt{L}}$ for this case)

$$\Rightarrow \ (u_i^{\mathtt{H}} - u_i^{\mathtt{L}})/(1 - \frac{u_i^{\mathtt{L}}}{u_i^{\mathtt{H}}}) \ \leq \ \theta_i^{\mathtt{H}} \ \leq \ (u_i^{\mathtt{H}} - u_i^{\mathtt{L}})/(1 - \frac{u_i^{\mathtt{L}}}{\overline{u}_i^{\mathtt{L}}})$$

$$\Leftrightarrow \ u_i^{\mathtt{H}} \ \leq \ \theta_i^{\mathtt{H}} \ \leq \ (u_i^{\mathtt{H}} - u_i^{\mathtt{L}})/(1 - \frac{u_i^{\mathtt{L}}}{\overline{u}_i^{\mathtt{L}}})$$

(From Eq. (3), we have $\overline{u}_i^{\mathtt{L}} = u_i^{\mathtt{L}}/(1 - u_i^{\mathtt{H}} + u_i^{\mathtt{L}})$)

$$\Leftrightarrow \ u_i^{\mathtt{H}} \ \leq \ \theta_i^{\mathtt{H}} \ \leq \ (u_i^{\mathtt{H}} - u_i^{\mathtt{L}})/(1 - \frac{u_i^{\mathtt{L}}}{u_i^{\mathtt{L}}/(1 - u_i^{\mathtt{H}} + u_i^{\mathtt{L}})})$$

$$\Leftrightarrow \ u_i^{\mathtt{H}} \ \leq \ \theta_i^{\mathtt{H}} \ \leq \ (u_i^{\mathtt{H}} - u_i^{\mathtt{L}})/(1 - \frac{1}{1/(1 - u_i^{\mathtt{H}} + u_i^{\mathtt{L}})})$$

$$\Leftrightarrow \ u_i^{\mathtt{H}} \ \leq \ \theta_i^{\mathtt{H}} \ \leq \ 1 \, .$$

◀

4.1 Algorithm MCFQ: Proof of Correctness

In this subsection, Theorem 8 proves that the MCFQ algorithm presented in Figure 2 is correct. Before presenting Theorem 8, we show in Lemma 7 that the execution rates $\theta_i^{\mathtt{L}}$ and $\theta_i^{\mathtt{H}}$ that are computed by MCFQ in Figure 2 ensure the correctness in HI-critical behavior for both LO- and HI-critical tasks by analyzing the special case in which during runtime it is detected that some job of a HI-critical task has executed beyond its LO-critical execution time (i.e., criticality of the system is switched).

▶ **Lemma 7.** *Assume that the system is schedulable in stable LO-critical behavior. Let t_o denote the first time instant at which some job of a HI-critical task does not signal completion despite having executed for its LO-critical WCET. Any LO or any HI critical task that is active (has been released but not completed execution equal to its HI-critical execution) at time instant t_o receives an amount of execution no smaller than its HI-critical WCET prior to its deadline.*

Proof. We will show that this lemma holds for both LO-critical tasks and HI-critical tasks.

LO-critical task. Suppose a job of a LO-critical task τ_i is active at time t_o. Recall from Step 1 of the algorithm in Figure 2 that the LO- and HI-critical execution rates are set as $\theta_i^{\mathtt{L}} = u_i^{\mathtt{L}}$ and $\theta_i^{\mathtt{H}} = u_i^{\mathtt{H}}$, where $u_i^{\mathtt{H}} \leq u_i^{\mathtt{L}}$ for each LO-critical task τ_i. Therefore, it is guaranteed that each job of τ_i will receive *at least* execution rate $u_i^{\mathtt{H}}$ from its release to its deadline. Consequently, the LO-critical active job at time t_o is guaranteed to complete its $C_i^{\mathtt{H}}$ units of execution by its deadline.

HI-critical task. Suppose a job of a HI-critical task τ_i is active at time t_o. Lemma 1 (originally proved as Lemma 5 in [12]) states that if $u_i^{\mathtt{L}}/\theta_i^{\mathtt{L}} + (u_i^{\mathtt{H}} - u_i^{\mathtt{L}})/\theta_i^{\mathtt{H}} \leq 1$ and the HI-critical task τ_i is schedulable in (stable) LO-critical behavior, then an active job of task τ_i at time t_o also meets its deadline. The MCFQ algorithm in Eq. (6) assigns the HI-critical execution rate $\theta_i^{\mathtt{H}}$ of task τ_i based on the value of $\theta_i^{\mathtt{L}}$ such that $\theta_i^{\mathtt{H}} = (u_i^{\mathtt{H}} - u_i^{\mathtt{L}})/(1 - u_i^{\mathtt{L}}/\theta_i^{\mathtt{L}})$ which implies $u_i^{\mathtt{L}}/\theta_i^{\mathtt{L}} + (u_i^{\mathtt{H}} - u_i^{\mathtt{L}})/\theta_i^{\mathtt{H}} \leq 1$. Therefore, any active job of the HI-critical task τ_i at time t_o meets its deadline under MCFQ. ◀

▶ **Theorem 8.** *If the condition in Eq. (7) of the MCFQ algorithm in Figure 2 is satisfied, then the execution rates assigned to the tasks constitute an MC-correct scheduling strategy.*

Proof. Lemma 5 and Lemma 6 together show that the values of $\theta_i^{\mathtt{L}}$ and $\theta_i^{\mathtt{H}}$ for each task $\tau_i \in \Gamma$ are larger than or equal to $u_i^{\mathtt{L}}$ and $u_i^{\mathtt{H}}$, respectively. In addition, Eq. (7) ensures that the individual sum of the LO- and HI-critical execution rates for all tasks is not larger than the capacity of the platform. Therefore, the system is correct for both stable LO- and HI-critical behaviors (i.e., when the system is not switching its criticality). Finally, Lemma 7 shows the correctness of the system upon transition from LO- to HI-critical behavior. ◀

The MCFQ algorithm has a speedup bound of 4/3 which is optimal for IMC task set. The proof of the speedup bound is given in Theorem 11 in Appendix A.

5 QoS Oriented Scheduling

When the MCFQ algorithm in Figure 2 declares "success", then the system is correct according to Theorem 8 and each LO-critical task has enough execution budget to provide degraded service during the HI-critical behavior of the system. In other words, each job of the LO-critical task τ_i executes for at most $C_i^{\mathtt{H}}$ time units during the HI-critical behavior and contributes a QoS value of $V_i^{\mathtt{H}}$ to the overall QoS of the system, where $V_i^{\mathtt{H}} \leq V_i^{\mathtt{L}}$ since $C_i^{\mathtt{H}} \leq C_i^{\mathtt{L}}$.

The question we investigate in this section is the following: If the system designer is not happy with the degraded service of the LO-critical tasks in HI-critical behavior, how can we make them happier? To answer this question we investigate the possibility of allowing some/all of the LO-critical tasks to provide full service during the HI-critical behavior of the system while ensuring correctness. If $(U_{\mathtt{H}}^{\mathtt{H}} + U_{\mathtt{L}}^{\mathtt{L}}) \leq m$, then each LO- and HI-critical task can be assigned $\theta_i^{\mathtt{L}} = \theta_i^{\mathtt{H}} = \max\{u_i^{\mathtt{L}}, u_i^{\mathtt{H}}\}$ and 100% QoS is achieved in all possible behaviors. Our proposed QoS-oriented scheduling in this section is applicable to task sets even when $(U_{\mathtt{H}}^{\mathtt{H}} + U_{\mathtt{L}}^{\mathtt{L}}) > m$.

If the algorithm in Figure 2 returns success, then slack in utilization during the HI-critical behavior is defined as

$$\mathcal{S} = (m - \sum_{\tau_i \in \Gamma} \theta_i^{\mathtt{H}}) \,. \tag{12}$$

Recall that MCFQ algorithm in Step 1 sets $\theta_i^{\mathtt{H}} = u_i^{\mathtt{H}}$ where $u_i^{\mathtt{H}} \leq u_i^{\mathtt{L}}$ for each LO-critical task $\tau_i \in \Gamma_L$. We will try to distribute the slack $\mathcal{S}$ to guarantee execution budget for some *selected* LO-critical tasks such that these tasks provide full service (i.e., execute $C_i^{\mathtt{L}}$ time units) also during the HI-critical behavior. In other words, the execution rates assigned to the variables $\theta_i^{\mathtt{H}}$ for the selected LO-critical tasks will be set to $u_i^{\mathtt{L}}$ rather than $u_i^{\mathtt{H}}$ so that τ_i provides full service during the HI-critical behavior. If some LO-critical tasks provides full service – due to their execution rates $\theta_i^{\mathtt{H}}$ being upgraded – such that the total increase in HI-critical execution rates in comparison to that is computed by the MCFQ algorithm is not larger than slack $\mathcal{S}$, then the correctness of the system is not compromised. This is because the execution rate of *no* HI-critical task is modified and the sum of HI-critical execution rates is still $\leq m$.

Consider the Example 3 in Section 4. The sum of the HI-critical execution rates is 1.6972 and the slack is 0.3027 where $m = 2$. The LO-critical task τ_3 is assigned execution rate $\theta_3^{\mathtt{H}} = u_3^{\mathtt{H}} = 0.125$ by algorithm MCFQ and provides degraded service during the HI-critical behavior. By increasing $\theta_3^{\mathtt{H}}$ from 0.125 to $u_3^{\mathtt{L}} = 0.2$, we can guarantee that τ_3 provides full service in all behaviors. In such case, the total increase in HI-critical execution rates is $(0.2 - 0.125) = 0.075$. Since we have a slack of 0.3027, the increase in execution rate of $\theta_3^{\mathtt{H}}$ by

0.075 will not violate the correctness. In such case, the QoS of the system is increased by $V_3^{\texttt{L}} - V_3^{\texttt{H}} = 1 - 0.6 = 0.4$ where $V_3^{\texttt{L}} = 1.0$.

Similarly, the LO-critical task τ_4's execution rate $\theta_4^{\texttt{H}} = u_4^{\texttt{H}} = 0.2$ can also be increased to $\theta_4^{\texttt{H}} = u_4^{\texttt{L}} = 0.5$. In such case, the HI-critical execution rate is increased by $(0.5 - 0.2) = 0.3$ which is less than the slack 0.3027 and the correctness of the system is not violated. In such case, the QoS of the system will be increased by $V_4^{\texttt{L}} - V_4^{\texttt{H}} = 1 - 0.4 = 0.6$. However, we cannot increase both $\theta_3^{\texttt{H}}$ and $\theta_4^{\texttt{H}}$ respectively to 0.2 and 0.5 because the total HI-critical execution rate will be increased by $(0.075 + 0.3) = 0.375$, which is larger than the slack 0.3027 and the system is not guaranteed to remain correct. To maximize the increase in overall system's QoS while ensuing correctness, at most one LO-critical task can be selected: task τ_4 is selected since it increases the QoS by 0.6 which is larger than that of task τ_3.

Given a correct system, we formulate an ILP to determine which LO-critical tasks are to be selected to maximize the increase in overall system's QoS while ensuring correctness. Let $x_i \in \{0, 1\}$ denote a decision variable whether the LO-critical task τ_i can be guaranteed to provide full service or not. The solution of the ILP determines the values of x_i for each LO-critical task. If $x_i = 1$, then the increase in HI-critical execution rate of the LO-critical task τ_i is $x_i \cdot (u_i^{\texttt{L}} - u_i^{\texttt{H}})$ and the increase in QoS is $x_i \cdot (V_i^{\texttt{L}} - V_i^{\texttt{H}})$.

The purpose of the ILP is to find the value of x_i for each LO-critical task τ_i such that (i) the total *increase* in QoS is maximized (i.e., $\sum_{\tau_i \in \Gamma_L} x_i \cdot (V_i^{\texttt{L}} - V_i^{\texttt{H}})$ is maximized), and (ii) the total *increase* in the HI-critical execution rates for all the LO-critical tasks is not larger than the slack $\mathcal{S}$ to ensure correctness (i.e., $\sum_{\tau_i \in \Gamma_L} x_i \cdot (u_i^{\texttt{L}} - u_i^{\texttt{H}}) \leq \mathcal{S}$). The LO-critical task τ_i for which $x_i = 1$ provides full service in all possible criticality behaviors. The value of the decision variable x_i for each $\tau_i \in \Gamma_L$ is determined using the following ILP:

$$\begin{aligned} \underset{x_i}{\text{maximize}} \quad & \sum_{\tau_i \in \Gamma_L} x_i \cdot (V_i^{\texttt{L}} - V_i^{\texttt{H}}) \\ \text{subject to} \quad & \sum_{\tau_i \in \Gamma_L} x_i \cdot (u_i^{\texttt{L}} - u_i^{\texttt{H}}) \leq \mathcal{S} \\ & \text{and } x_i = 0 \text{ or } x_i = 1 \end{aligned} \tag{13}$$

Given the values of x_i for all the LO-critical tasks in Γ_L, the total increase in QoS of the system is normalized by the number of LO-critical tasks. The normalized QoS, denoted by V_{task}^{norm}, is given in Eq. (14). We set $V_{task}^{norm} = 0$ if $|\Gamma_L| = 0$. Note that $0 \leq V_{task}^{norm} \leq 1$.

$$V_{task}^{norm} = \frac{\sum_{\tau_i \in \Gamma_L} x_i \cdot (V_i^{\texttt{L}} - V_i^{\texttt{H}})}{|\Gamma_L|} \tag{14}$$

6 Empirical Results

This section presents the results on the effectiveness of MCFQ algorithm both in terms of schedulability and improving the system-level QoS using randomly generated implicit-deadline sporadic IMC task sets. The proposed MCFQ algorithm is compared against the MC-Fluid [12] and MCF [3] algorithms. However, the MC-Fluid and MCF algorithms do not consider IMC task models (i.e., all the LO-critical tasks are aborted once the system switches to HI-critical behavior). Baruah et al. [2] extended the MC-Fluid algorithm for uniprocessor considering IMC task model. We extended the MC-Fluid and MCF algorithms for multiprocessors based on a similar approach in [2] for IMC tasks (details of this extension are in Appendix B). Before we present our results, we present the task set generation algorithm.

6.1 Task Set Generation Algorithm

Random implicit-deadline sporadic MC tasksets are generated using an approach similar to those in [12, 3, 13]. Let $U_B = max\{(U_{\mathtt{H}}^{\mathtt{H}} + U_{\mathtt{L}}^{\mathtt{H}})/m, (U_{\mathtt{H}}^{\mathtt{L}} + U_{\mathtt{L}}^{\mathtt{L}})/m\}$ denotes the upper bound on normalized total system utilization in both LO- and HI-critical behaviors. The number of tasks in a randomly generated task set is controlled using an upper bound on the individual task's utilization (u_{max}). The proportion of HI-critical tasks is controlled using probability (p_h). The ratio of HI- and LO-critical utilizations of each task τ_i is controlled using a parameter (R_{max}) such that $1 \leq u_i^{\mathtt{L}} / u_i^{\mathtt{H}} \leq R_{max}$ for a LO-critical task and $1 \leq u_i^{\mathtt{H}} / u_i^{\mathtt{L}} \leq R_{max}$ for a HI-critical task. The following set of values are considered in our experiments for the task set parameters:

- Number of processors: $m \in \{2, 4, 8, 16\}$.
- Normalized utilization bound: $U_B \in \{0.1, 0.15, \ldots 1.0\}$.
- Probability of tasks to be of HI-critical: $p_h \in \{0.0, 0.1, 0.2, \ldots 1.0\}$.
- Maximum individual task utilization: $u_{max} \in \{0.1, 0.2, 0.3, \ldots 1.0\}$.
- Maximum ratio of individual task's utilizations: $R_{max} \in \{1.0, 1.4, 1.8, \ldots 4.0\}$.

We consider 75,240 different combinations of the above parameters to generate the tasksets. For each combination, we generate 1000 task sets where each task set is generated as follows where each parameter is selected from an uniform distribution.

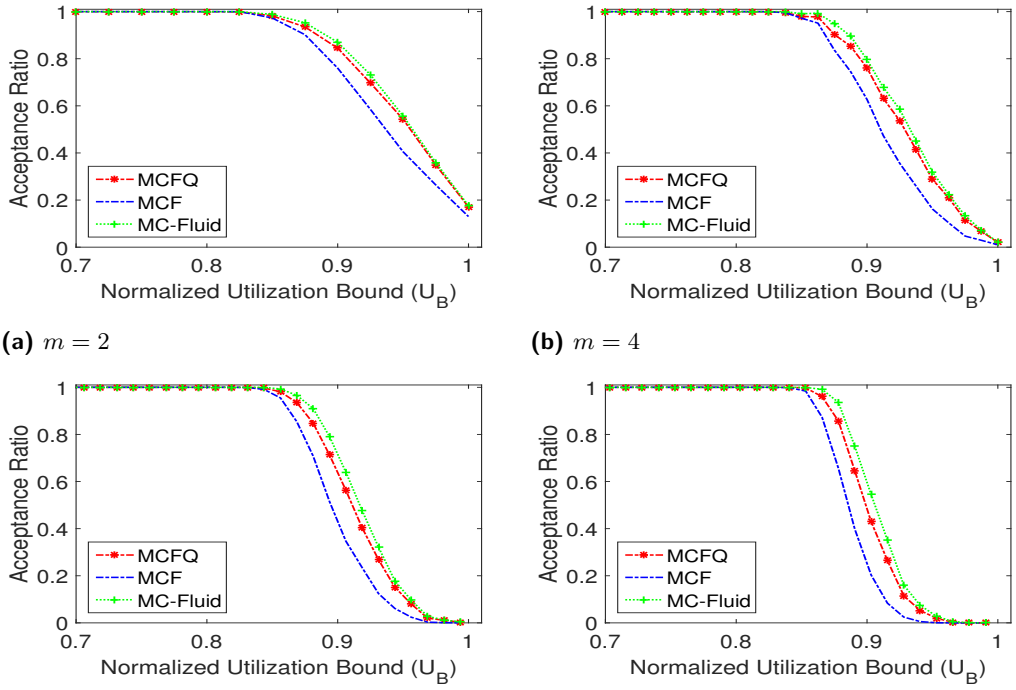
- A real number P_i is drawn from the range $[0, 1]$. If $P_i < p_h$, then $L_i = \mathtt{HI}$; otherwise $L_i = \mathtt{LO}$.
- Task period T_i is drawn from the range $[10, 1000]$.
- Task utilization u_i is drawn from the range $[0.02, u_{max}]$.
- A real number R_i is drawn from the range $[1, R_{max}]$.
- If $L_i = \mathtt{LO}$, then $u_i^{\mathtt{L}} = u_i$ and $u_i^{\mathtt{H}} = (u_i/R_i)$. Otherwise, $u_i^{\mathtt{H}} = u_i$ and $u_i^{\mathtt{L}} = (u_i/R_i)$. The value of $C_i^{\mathtt{L}} = \lceil u_i^{\mathtt{L}} \cdot T_i \rceil$ and $C_i^{\mathtt{H}} = \lceil u_i^{\mathtt{H}} \cdot T_i \rceil$.
- If $L_i = \mathtt{LO}$, then $V_i^{\mathtt{L}} = 1.0$ and $V_i^{\mathtt{H}} = C_i^{\mathtt{H}}/C_i^{\mathtt{L}}$, i.e., the QoS value is set by the system designer based on imprecise computation model [15, 16].
- Repeat the above steps as long as $max\{(U_{\mathtt{H}}^{\mathtt{H}} + U_{\mathtt{L}}^{\mathtt{H}})/m, (U_{\mathtt{H}}^{\mathtt{L}} + U_{\mathtt{L}}^{\mathtt{L}})/m\} \leq U_B$. Once the condition is violated, discard the task that was generated the last.
- If the resulting task set satisfies the condition $max\{(U_{\mathtt{H}}^{\mathtt{H}} + U_{\mathtt{L}}^{\mathtt{H}})/m, (U_{\mathtt{H}}^{\mathtt{L}} + U_{\mathtt{L}}^{\mathtt{L}})/m\} > U_B - 0.05$, then accept the task set and stop the procedure. Otherwise, discard the taskset and the repeat the above steps.

6.2 Results: Schedulability Tests

We compare the effectiveness of the MCFQ algorithm in terms of schedulability of randomly generated task sets with the (extended) MC-Fluid and MCF algorithms applicable to IMC task sets. For a specific scheduling algorithm, and m, U_B, p_h, R_{max} values, let the *acceptance ratio* denote the fraction of task sets out of 1000 task sets that are deemed schedulable by the algorithm.

The HI-critical tasks are considered in increasing order of $u_i^{\mathtt{H}}/\overline{u}_i^{\mathtt{L}}$ when assigning the execution rates based on the MCFQ algorithm. Figure 3 presents the acceptance ratio for each scheduling algorithm against various values of m and U_B with $p_h = 0.5$, $u_{max} = 0.9$ and $R_{max} = 2$. All the algorithms have acceptance ratio 100% when $U_B < 0.70$ and we plot results for $U_B > 0.7$.

We have the following observations. The MCFQ algorithm outperforms the MCF algorithm for task sets with large utilization. The performance of the MCFQ algorithm is very close to



(a) $m = 2$ **(b)** $m = 4$

(c) $m = 8$ **(d)** $m = 16$

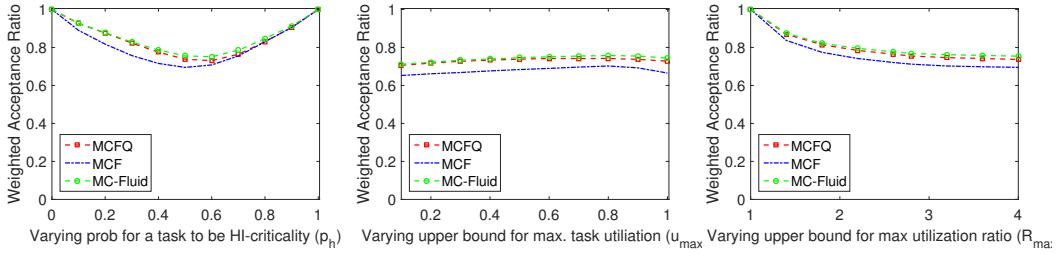
Figure 3 Comparison of acceptance ratios for different number of processors.

the optimal MC-Fluid algorithm. For larger U_B, the acceptance ratio of MCF drops sharply while the performance of `MCFQ` remains very close to the optimal MC-Fluid algorithm.

For comparison of the acceptance ratios of different algorithms for varying values of p_h, u_{max} and R_{max}, we also computed the *weighted acceptance ratios* in Figure 4. This metric denotes the fraction of schedulable task sets weighted by the normalized utilization bound U_B. If $AR(U_B)$ denotes the acceptance ratio of a scheduling algorithm for normalized utilization bound U_B for some given values of p_h, u_{max}, R_{max} and m, then the weighted acceptance ratio for a set S of U_B values is given as $W(S) = \left(\sum_{U_B \in S}(AR(U_B) \times U_B)\right) / \sum_{U_B \in S} U_B$.

In Figure 4a, we plot the weighted acceptance ratio of the algorithms for different values of p_h for $u_{max} = 0.9$ and $R_{max} = 2$. The performance of the algorithms is better when the value of p_h is either very small or very large since at these extremes the task sets behaves more like non-`MC` task systems and the effect of switching the criticality behavior has less impact on schedulability. In Figure 4b, we plot the weighted acceptance ratio of the algorithms for different values of u_{max} for $p_h = 0.5$ and $R_{max} = 2$. The performance of the algorithms is independent of the variation in u_{max} (which is also observed in [3] for non-IMC task sets).

In Figure 4c, we plot the weighted acceptance ratio of the algorithms for different values of R_{max} for $p_h = 0.5$ and $u_{max} = 0.9$. The performance of the algorithms decreases with increasing values of R_{max}. When R_{max} increases, the total `HI`-critical utilization of the `HI`-critical tasks also increases and the total `HI`-critical utilization of the `LO`-critical tasks decreases. As it is already shown in [3] for non-IMC tasks (i.e., `LO`-critical tasks are dropped at criticality switch), the weighted acceptance ratio decreases with larger R_{max}. For IMC task sets in which the `LO`-critical tasks execute in `HI`-critical behavior with degraded service, it is even more difficult to schedule such task sets as R_{max} increases.



(a) Varying p_h **(b)** varying u_{max} **(c)** Varying R_{max}

Figure 4 Comparison of weighted acceptance ratios for different number of processors.

Considering the plots in Figure 4a–4c, it is evident that the performance of `MCFQ` algorithm is much better than the MCF algorithm and is very close to the optimal MC-Fluid algorithm for varying values of u_{max}, p_h, and R_{max} for IMC task sets.

6.3 Results: System's QoS

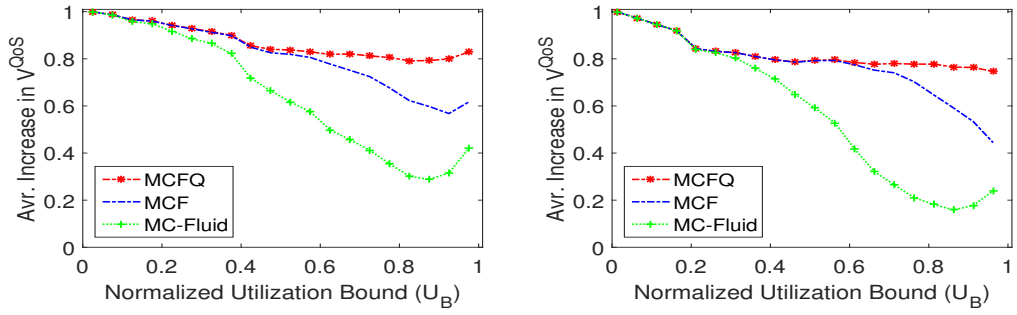
In this section, we present the effectiveness of `MCFQ` algorithm in increasing the overall QoS value of the system (defined as V_{task}^{norm} in Eq. (14)) in comparison to MC-Fluid and MCF algorithms.

If a task set is not schedulable using a particular algorithm, then such a task set is not subjected to QoS evaluation. This is because the system designer's first concern is ensuring `MC`-correctness (i.e., schedulability). If more than one algorithm guarantee the `MC`-correctness of a given task system, then the system designer's second concern is which algorithm maximizes the QoS of the system. Therefore, we consider only those task sets that are schedulable using all the three algorithms for QoS evaluation based on the approach presented in Section 5. For each such randomly-generated task set and each particular algorithm, we determine V_{task}^{norm} by solving the ILP given in Eq. (13) using Matlab's `intlinprog` function. If $\mathcal{K}$ out of 1000 tasksets for a particular configuration are deemed to be schedulable using all three algorithms, then the average V_{task}^{norm}, denoted by V^{QoS}, of these $\mathcal{K}$ task sets is computed for each algorithm.

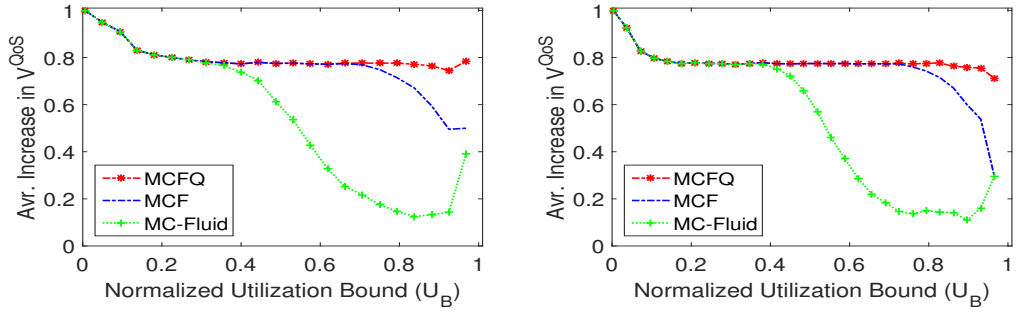
Figure 5 presents the average increase in overall QoS of the system (i.e., value of V^{QoS}) for each scheduling algorithm against various values of m and U_B with $p_h = 0.5$, $u_{max} = 0.9$ and $R_{max} = 2$. The `MCFQ` algorithm outperforms *both* MC-Fluid and MCF. This is because the amount of slack available during the `HI`-critical behavior, based on the execution rates determined by algorithm `MCFQ` in Figure 2, is much larger than that of both MC-Fluid and MCF algorithms. Such slack allows more `LO`-critical tasks to provide full service. When the utilization of the system is very large, the `MCFQ` algorithm provides much higher QoS than both MC-Fluid and MCF.

Figure 6 presents the average number of `LO`-critical tasks (among all the `LO`-critical tasks of all the $\mathcal{K}$ schedulable task sets at each utilization point) that provide full service during the `HI`-critical behavior of the system for various values of m and U_B with $p_h = 0.5$, $u_{max} = 0.9$, and $R_{max} = 2$.

It is evident from Figure 6 that when the utilization of the system is low, then all the algorithms allow almost all the `LO`-critical tasks to provide full service in all behaviors. Earlier approaches do not consider such QoS improvement for systems with $(U_{\mathtt{H}}^{\mathtt{H}} + U_{\mathtt{L}}^{\mathtt{L}}) > m$, i.e., the `LO`-critical tasks are either aborted (non-IMC task model) or only guaranteed to provide degraded (IMC task model) service.

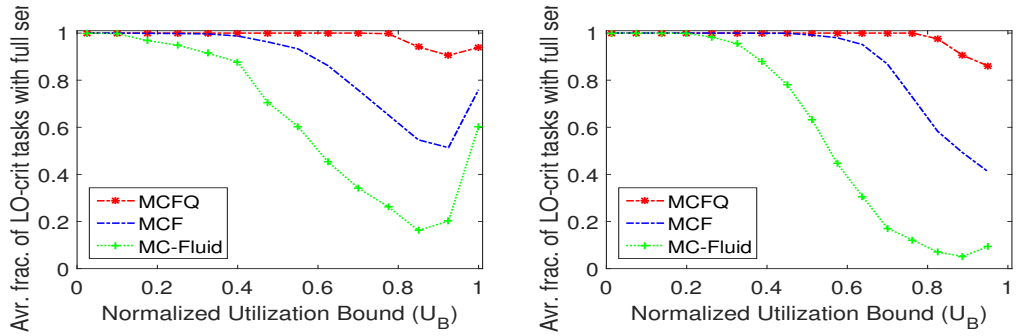


(a) m=2 **(b)** m=4

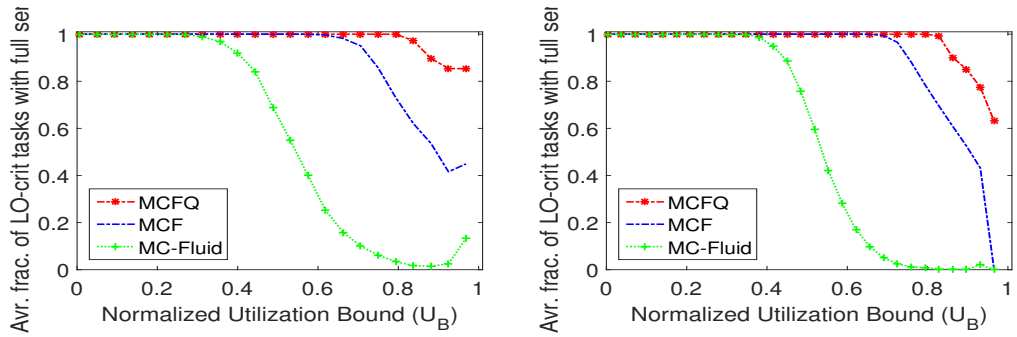


(c) m=8 **(d)** m=16

Figure 5 Average increase in QoS of the system (V^{QoS})



(a) m=2 **(b)** m=4



(c) m=8 **(d)** m=16

Figure 6 Average fraction of LO-critical tasks providing full service in both criticality behaviors

The MCFQ algorithm allows almost 100% of *all* the LO-critical tasks to provide full service up to very high utilization ($U_B \approx 0.8$) in comparison to both MC-Fluid and MCF algorithms for different number of processors. The MCFQ algorithm allows much larger fraction of the LO-critical tasks to provide full service in comparison to both MC-Fluid and MCF when the utilization is higher than 0.8.

7 Conclusion

This paper proposes the MCFQ algorithm based on the fluid scheduling model and determines the execution rates for a set of implicit-deadline IMC sporadic tasks considering multiprocessor platform. The recently proposed IMC task model is extended with two QoS values for each LO-critical task. The system designer can assign these QoS values and determine the QoS of the overall system.

The design of the execution rate assignment algorithm of MCFQ ensures that the system is fully utilized during the LO-critical behavior so that the system may have some slack capacity during the HI-critical behavior. The slack, if available, is distributed to the LO-critical tasks such that some of these tasks continue to provide full service after the system switches to HI-critical behavior. The LO-critical tasks that provide full service improve the QoS of the system – making the system designers happier.

References

1. S. Baruah, V. Bonifaci, G. DAngelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie. The Preemptive Uniprocessor Scheduling of Mixed-Criticality Implicit-Deadline Sporadic Task Systems. In *Proc. of ECRTS*, 2012. doi:10.1109/ECRTS.2012.42.
2. S. Baruah, A. Burns, and Z. Guo. Scheduling mixed-criticality systems to guarantee some service under all non-erroneous behaviors. In *Proc. of ECRTS*, 2016. doi:10.1109/ECRTS.2016.12.
3. S. Baruah, A. Eswaran, and Z. Guo. MC-Fluid: Simplified and Optimally Quantified. In *Proc. of RTSS*, 2015. doi:10.1109/RTSS.2015.38.
4. S. Baruah, Haohan Li, and L. Stougie. Towards the Design of Certifiable Mixed-criticality Systems. In *Proc. of RTAS*, 2010. doi:10.1109/RTAS.2010.10.
5. S. Baruah and S. Vestal. Schedulability Analysis of Sporadic Tasks with Multiple Criticality Specifications. In *Proc. of ECRTS*, 2008. doi:10.1109/ECRTS.2008.26.
6. Sanjoy Baruah, Alan Burns, and Robert Davis. Response-time analysis for mixed criticality systems. In *Proc. of RTSS*, 2011. doi:10.1109/RTSS.2011.12.
7. A. Burns and S. Baruah. Towards a more practical model for mixed criticality systems. In *Proc. of WMC, RTSS*, 2013. http://www-users.cs.york.ac.uk/~robdavis/wmc2013/paper3.pdf.
8. A. Burns and R. Davis. Mixed-criticality systems: A review. In *(available online), Eighth Edition*, July, 2016. http://www-users.cs.york.ac.uk/~burns/review.pdf.
9. Oliver Gettings, Sophie Quinton, and Robert I. Davis. Mixed criticality systems with weakly-hard constraints. In *Proc. of RTNS*, 2015. doi:10.1145/2834848.2834850.
10. Nan Guan, Pontus Ekberg, Martin Stigge, and Wang Yi. Effective and Efficient Scheduling of Certifiable Mixed-Criticality Sporadic Task Systems. In *Proc. of RTSS*, 2011. doi:10.1109/RTSS.2011.10.
11. Mathieu Jan, Lilia Zaourar, and Maurice Pitel. Maximizing the execution rate of low-criticality tasks in mixed criticality systems. In *Proc. of WMC, RTSS*, 2013. http://www-users.cs.york.ac.uk/~robdavis/wmc2013/paper6.pdf.

12 J. Lee, K. M. Phan, X. Gu, J. Lee, A. Easwaran, I. Shin, and I. Lee. MC-Fluid: Fluid Model-Based Mixed-Criticality Scheduling on Multiprocessors. In *Proc. of RTSS*, 2014. doi:10.1109/RTSS.2014.32.

13 Haohan Li and Sanjoy Baruah. Global mixed-criticality scheduling on multiprocessors. In *Proc of ECRTS*, 2012. doi:10.1109/ECRTS.2012.41.

14 Di Liu, Jelena Spasic, Gang Chen, Nan Guan, Songran Liu, Todor Stefanov, and Wang Yi. EDF-VD Scheduling of Mixed-Criticality Systems with Degraded Quality Guarantees. In *Proc. of RTSS*, 2016. doi:10.1109/RTSS.2016.013.

15 Jane W. S. Liu, Kwei-Jay Lin, Wei-Kuan Shih, Albert Chuang-shi Yu, Jen-Yao Chung, and Wei Zhao. Algorithms for scheduling imprecise computations. *Computer*, 24(5):58–68, May 1991. doi:10.1007/978-1-4615-3956-8_8.

16 J. W. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise computations. *Proceedings of the IEEE*, 82(1):83–94, 1994. doi:10.1109/5.259428.

17 Risat Mahmud Pathan. Fault-tolerant and real-time scheduling for mixed-criticality systems. *Real-Time Systems*, 50(4):509–547, 2014. doi:10.1007/s11241-014-9202-z.

18 F. Santy, L. George, P. Thierry, and J. Goossens. Relaxing Mixed-Criticality Scheduling Strictness for Task Sets Scheduled with FP. In *Proc. of ECRTS*, 2012. doi:10.1109/ECRTS.2012.39.

19 H. Su, N. Guan, and D. Zhu. Service guarantee exploration for mixed-criticality systems. In *Proc. of RTCSA*, 2014. doi:10.1109/RTCSA.2014.6910499.

20 H. Su and D. Zhu. An elastic mixed-criticality task model and its scheduling algorithm. In *Proc. of DATE*, 2013. doi:10.7873/DATE.2013.043.

21 S. Vestal. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *Proc. of RTSS*, pages 239–243, 2007. doi:10.1109/RTSS.2007.47.

A Speed Up Bound

The MCFQ algorithm has a speed-up bound of 4/3: if a given dual-criticality implicit-deadline IMC sporadic task system can be scheduled upon a particular multiprocessor platform in an MC-correct manner by any algorithm (including an optimal, clairvoyant, one), then it can be scheduled by MCFQ upon a platform in which each processor is faster by a factor 4/3. It is already shown (in Theorem 5 in [1]) that no non-clairvoyant algorithm for scheduling dual-criticality implicit-deadline non-IMC sporadic task systems can have a speedup factor smaller than 4/3 even on uniprocessor (i.e., for $m = 1$). Therefore, the speed-up bound of 4/3 for MCFQ is optimal since IMC task model is a generalization of non-IMC task model.

To prove the speed-up of 4/3 in Theorem 11, we use Lemma 9 and Lemma 10. Lemma 9 shows that the sum of the LO-critical execution rates that are determined by the MCFQ algorithm in Figure 2 does not exceeds the capacity of the platform.

▶ **Lemma 9.** *The MCFQ algorithm in Figure 2 ensures that* $\sum_{\tau_i \in \Gamma} \theta_i^L \leq m$.

Proof. Each LO-critical task $\tau_i \in \Gamma_L$ is assigned LO-critical execution rate $\theta_i^{\mathtt{L}} = u_i^{\mathtt{L}}$ in Step 1 of the algorithm in Figure 2. Therefore, $\sum_{\tau_i \in \Gamma_L} \theta_i^{\mathtt{L}} = \sum_{\tau_i \in \Gamma_L} u_i^{\mathtt{L}} = U_{\mathtt{L}}^{\mathtt{L}}$. Since $\Gamma = \Gamma_H \cup \Gamma_L$, this lemma is proved by showing that $\sum_{\tau_i \in \Gamma_H} \theta_i^{\mathtt{L}} \leq m - U_{\mathtt{L}}^{\mathtt{L}}$.

Let the tasks in $\Gamma_H = \{\tau_1, \tau_2, \ldots \tau_h\}$ are indexed (in an arbitrary order) such that there are $h = |\Gamma_H|$ tasks in set Γ_H. Since $\theta_i^{\mathtt{L}} = min\{u_i^{\mathtt{H}}, \mathcal{F}_{i-1} \cdot \overline{u}_i^{\mathtt{L}}\}$ in Eq. (5) for $\tau_i \in \Gamma_H$, we have

$$\theta_i^{\mathtt{L}} = min\{u_i^{\mathtt{H}}, \mathcal{F}_{i-1} \cdot \overline{u}_i^{\mathtt{L}}\} \leq u_i^{\mathtt{H}} \tag{15}$$

$$\theta_i^{\mathtt{L}} = min\{u_i^{\mathtt{H}}, \mathcal{F}_{i-1} \cdot \overline{u}_i^{\mathtt{L}}\} \leq \mathcal{F}_{i-1} \cdot \overline{u}_i^{\mathtt{L}} \tag{16}$$

Recall from Eq. (9) that $1 \leq \mathcal{F}_0 \leq \mathcal{F}_1 \leq \ldots \leq \mathcal{F}_{h-1}$. We prove this lemma by considering two cases: case (i) $\mathcal{F}_0 = \mathcal{F}_1 = \ldots = \mathcal{F}_{h-1}$, and case (ii) $\mathcal{F}_{k-1} < \mathcal{F}_k$ for some k, $1 \leq k \leq h-1$.

Case (i): From Eq. (8), we have $\mathcal{F}_0 \cdot \sum_{i=1}^{h} \overline{u}_i^{\mathtt{L}} = \mathcal{F}_0 \cdot \overline{U}_{\mathtt{H}}^{\mathtt{L}} = (m - U_{\mathtt{L}}^{\mathtt{L}})$. From Eq. (16), we have

$$\sum_{\tau_i \in \Gamma_H} \theta_i^{\mathtt{L}} = \sum_{i=1}^{h} \theta_i^{\mathtt{L}} \leq \sum_{i=1}^{h} \mathcal{F}_{i-1} \cdot \overline{u}_i^{\mathtt{L}}$$

(For this case $\mathcal{F}_i = \mathcal{F}_0$ for $i = 0, 1 \ldots (h-1)$)

$$\Leftrightarrow \sum_{\tau_i \in \Gamma_H} \theta_i^{\mathtt{L}} = \sum_{i=1}^{h} \theta_i^{\mathtt{L}} \leq \sum_{i=1}^{h} \mathcal{F}_{i-1} \cdot \overline{u}_i^{\mathtt{L}} = \sum_{i=1}^{h} \mathcal{F}_0 \cdot \overline{u}_i^{\mathtt{L}} = m - U_{\mathtt{L}}^{\mathtt{L}}$$

Case (ii): Let q is the largest index in the range $[1, 2, \ldots (h-1)]$ such that $\mathcal{F}_{q-1} < \mathcal{F}_q$ where $1 \leq q \leq h-1$. Such a q must exist for this case. Since q is largest index, we have based on Eq. (9)

$$\mathcal{F}_{q-1} < \mathcal{F}_q = \mathcal{F}_{q+1} = \ldots = \mathcal{F}_{h-1} \tag{17}$$

Since $\mathcal{F}_q = max\{\mathcal{F}_{q-1}, \frac{m - U_{\mathtt{L}}^{\mathtt{L}} - \sum_{i=1}^{q} u_i^{\mathtt{H}}}{\overline{U}_{\mathtt{H}}^{\mathtt{L}} - \sum_{i=1}^{q} \overline{u}_i^{\mathtt{L}}}\}$ according to Eq. (8) and $\mathcal{F}_{q-1} < \mathcal{F}_q$ for this case, we have

$$\mathcal{F}_q = \frac{m - U_{\mathtt{L}}^{\mathtt{L}} - \sum_{i=1}^{q} u_i^{\mathtt{H}}}{\overline{U}_{\mathtt{H}}^{\mathtt{L}} - \sum_{i=1}^{q} \overline{u}_i^{\mathtt{L}}} > \mathcal{F}_{q-1} \tag{18}$$

To prove this lemma we show that

$$\sum_{\tau_i \in \Gamma_H} \theta_i^{\mathtt{L}} = \sum_{i=1}^{h} \theta_i^{\mathtt{L}} = \sum_{i=1}^{q} \theta_i^{\mathtt{L}} + \sum_{i=q+1}^{h} \theta_i^{\mathtt{L}} \leq m - U_{\mathtt{L}}^{\mathtt{L}}$$

(From Eq. (15) and Eq. (16))

$$\Leftarrow \sum_{i=1}^{q} u_i^{\mathtt{H}} + \sum_{i=q+1}^{h} \mathcal{F}_{i-1} \cdot \overline{u}_i^{\mathtt{L}} \leq m - U_{\mathtt{L}}^{\mathtt{L}}$$

(From Eq. (17), $\mathcal{F}_q = \mathcal{F}_{i-1}$ for $i = q+1, q+2, \ldots h$))

$$\Leftrightarrow \sum_{i=1}^{q} u_i^{\mathtt{H}} + \mathcal{F}_q \cdot \sum_{i=q+1}^{h} \overline{u}_i^{\mathtt{L}} \leq m - U_{\mathtt{L}}^{\mathtt{L}}$$

$$\Leftrightarrow \sum_{i=1}^{q} u_i^{\mathtt{H}} + \mathcal{F}_q \cdot (\overline{U}_{\mathtt{H}}^{\mathtt{L}} - \sum_{i=1}^{q} \overline{u}_i^{\mathtt{L}}) \leq m - U_{\mathtt{L}}^{\mathtt{L}}$$

(From Eq. (18))

$$\Leftrightarrow \sum_{i=1}^{q} u_i^{\mathtt{H}} + \frac{m - U_{\mathtt{L}}^{\mathtt{L}} - \sum_{i=1}^{q} u_i^{\mathtt{H}}}{(\overline{U}_{\mathtt{H}}^{\mathtt{L}} - \sum_{i=1}^{q} \overline{u}_i^{\mathtt{L}})} \cdot (\overline{U}_{\mathtt{H}}^{\mathtt{L}} - \sum_{i=1}^{q} \overline{u}_i^{\mathtt{L}}) \leq m - U_{\mathtt{L}}^{\mathtt{L}}$$

$$\Leftrightarrow m - U_{\mathtt{L}}^{\mathtt{L}} \leq m - U_{\mathtt{L}}^{\mathtt{L}}$$

Therefore, the sum of the LO-critical execution rates of all the tasks is not larger than m. ◀

▶ **Lemma 10.** *Consider the following function $f(x)$ where*

$$f(x) = x \cdot ((2s-1)m - x)$$

for $0 \leq x \leq (2s-1) \cdot m$. The maximum value of function $f(x)$ is $\frac{(2sm-m)^2}{4}$.

Proof. The first derivative of $f(x)$ with respect to x is $f'(x) = ((2s-1)m - 2x)$. The derivative is zero for $x = (2s-1)m/2$. Considering the two end-points of the interval $[0, (2s-1)m]$, function $f(x)$ reaches its maximum for some $x \in \{0, (2s-1)m/2, (2s-1)m\}$. We have $f(x) = 0$ when either $x = 0$ or $x = (2s-1)m$. We have $f(x) = \frac{(2sm-m)^2}{4}$ when $x = (2s-1)m/2$. Therefore, the maximum of $f(x)$ within $[0, (2s-1)m]$ is $\frac{(2sm-m)^2}{4}$. ◀

In addition to Lemma 9–10, we need Eq. (19) to show that MCFQ has a speed-up bound 4/3.

$$\theta_i^{\mathtt{H}} = \frac{(u_i^{\mathtt{H}} - u_i^{\mathtt{L}})}{(1 - \frac{u_i^{\mathtt{L}}}{\theta_i^{\mathtt{L}}})} = u_i^{\mathtt{H}} + \frac{u_i^{\mathtt{H}} - u_i^{\mathtt{L}} - u_i^{\mathtt{H}}(1 - \frac{u_i^{\mathtt{L}}}{\theta_i^{\mathtt{L}}})}{1 - \frac{u_i^{\mathtt{L}}}{\theta_i^{\mathtt{L}}}} = u_i^{\mathtt{H}} + u_i^{\mathtt{L}} \cdot \frac{u_i^{\mathtt{H}} - \theta_i^{\mathtt{L}}}{\theta_i^{\mathtt{L}} - u_i^{\mathtt{L}}} \tag{19}$$

▶ **Theorem 11.** *The speedup bound of MCFQ is 4/3.*

Proof. Consider an IMC task set that is feasible using an algorithm (including an optimal, clairvoyant, one) on m speed-s processors where $s \leq 3/4$. We will now show that the MCFQ algorithm in Figure 2 also declares success (i.e., the condition in Eq. (7) is satisfied) for this task set for m speed-1 processors. Since $1/(3/4) = 4/3$, the MCFQ algorithm has speedup bound 4/3.

If a task system is feasible upon m speed-s processors, it is necessary that the following three conditions hold:

$$(U_{\mathtt{H}}^{\mathtt{H}} + U_{\mathtt{L}}^{\mathtt{H}}) \leq m \cdot s \tag{20}$$

$$(U_{\mathtt{L}}^{\mathtt{L}} + \overline{U}_{\mathtt{H}}^{\mathtt{L}}) \leq m \cdot s \tag{21}$$

$$\forall_i : \quad max\{u_i^{\mathtt{L}}, u_i^{\mathtt{H}}\} \leq s \tag{22}$$

We assume that $U_{\mathtt{H}}^{\mathtt{H}} + U_{\mathtt{L}}^{\mathtt{L}} > m$ since a task set with $U_{\mathtt{H}}^{\mathtt{H}} + U_{\mathtt{L}}^{\mathtt{L}} \leq m$ is trivially schedulable by setting $\theta_i^{\mathtt{L}} = \theta_i^{\mathtt{H}} = \max\{u_i^{\mathtt{L}}, u_i^{\mathtt{H}}\}$ for each task $\tau_i \in \Gamma$. From Eq. (20) and Eq. (21), we have

$$\begin{aligned} & 0 \leq (U_{\mathtt{L}}^{\mathtt{L}} + U_{\mathtt{H}}^{\mathtt{H}}) + (U_{\mathtt{L}}^{\mathtt{H}} + \overline{U}_{\mathtt{H}}^{\mathtt{L}}) \leq 2 \cdot m \cdot s \\ \Rightarrow\ & 0 \leq (U_{\mathtt{L}}^{\mathtt{L}} + U_{\mathtt{H}}^{\mathtt{H}}) \leq 2 \cdot m \cdot s - \overline{U}_{\mathtt{H}}^{\mathtt{L}} \qquad (23) \\ & \quad \text{(Since we assume task sets where } U_{\mathtt{L}}^{\mathtt{L}} + U_{\mathtt{H}}^{\mathtt{H}} > m) \\ \Rightarrow\ & 0 \leq \overline{U}_{\mathtt{H}}^{\mathtt{L}} \leq 2 \cdot m \cdot s - m = (2s-1) \cdot m \qquad (24) \end{aligned}$$

Since Eq. (20)–(22) hold, the assumptions for applying the MCFQ algorithm in Figure 2 are true. Therefore, it follows from Lemma 9 that $\sum_{\tau_i \in \Gamma} \theta_i^{\mathtt{L}} \leq m$ for MCFQ algorithm. To prove this theorem, we now show that $\sum_{\tau_i \in \Gamma} \theta_i^{\mathtt{H}} \leq m$, which implies that the condition in Step 3 in Figure 2 will be true..

From Step 1 in Figure 2, we have $\sum_{\tau_i \in \Gamma_L} \theta_i^{\mathtt{H}} = \sum_{\tau_i \in \Gamma_L} u_i^{\mathtt{H}} = U_{\mathtt{L}}^{\mathtt{H}}$. Since $\Gamma = \Gamma_H \cup \Gamma_L$, we only need to show that $\sum_{\tau_i \in \Gamma_H} \theta_i^{\mathtt{H}} \leq m - U_{\mathtt{L}}^{\mathtt{H}}$ to prove this theorem. From Eq. (6), the value $\theta_i^{\mathtt{H}}$ for $\tau_i \in \Gamma_H$ is

$$\theta_i^{\mathtt{H}} = (u_i^{\mathtt{H}} - u_i^{\mathtt{L}})/(1 - \frac{u_i^{\mathtt{L}}}{\theta_i^{\mathtt{L}}}) \,.$$

It is evident from the above equation that $\theta_i^{\mathtt{H}}$ increases as $\theta_i^{\mathtt{L}}$ decreases. Recall from Eq. (5) that the maximum value of $\theta_i^{\mathtt{L}}$ is $u_i^{\mathtt{H}}$ for which the value of $\theta_i^{\mathtt{H}}$ is *minimized*. According to Eq. (10), the minimum value of $\theta_i^{\mathtt{H}}$ is $u_i^{\mathtt{H}}$ given that $\theta_i^{\mathtt{L}}$ is also equal to $u_i^{\mathtt{H}}$.

For any feasible task set, the value of $\theta_i^{\mathtt{H}}$ must be at least equal to $u_i^{\mathtt{H}}$ for each HI-critical task τ_i in order to ensure that the system is correct during stable HI-critical behavior. The

value of $\theta_i^{\mathtt{H}}$ is larger than $u_i^{\mathtt{H}}$ when $\theta_i^{\mathtt{L}}$ is smaller than $u_i^{\mathtt{H}}$. Therefore, the sum of the HI-critical execution rates of the HI-critical tasks is maximized when each of the LO-critical execution rate $\theta_i^{\mathtt{L}}$ for $\tau_i \in \Gamma_H$ is smaller than $u_i^{\mathtt{H}}$. According to Eq. (5), the value of $\theta_i^{\mathtt{L}}$ is smaller than $u_i^{\mathtt{H}}$ only if $u_i^{\mathtt{H}} \geq \mathcal{F}_{i-1} \cdot \overline{u}_i^{\mathtt{L}}$. Therefore, the sum of $\theta_i^{\mathtt{H}}$ for all the HI-critical tasks is maximized when $\theta_i^{\mathtt{L}} = \mathcal{F}_{i-1} \cdot \overline{u}_i^{\mathtt{L}} \leq u_i^{\mathtt{H}}$ for all $\tau_i \in \Gamma_H$. To prove this theorem we only need to consider the worst-case where $\theta_i^{\mathtt{L}} = \mathcal{F}_{i-1} \cdot \overline{u}_i^{\mathtt{L}}$ for each task $\tau_i \in \Gamma_H$ because the sum of the HI-critical execution rates of the HI-critical tasks is maximized under this worst-case.

We will show that if $s \leq 4/3$, then the following holds even under the worst-case assumption that $\theta_i^{\mathtt{L}} = \mathcal{F}_{i-1} \cdot \overline{u}_i^{\mathtt{L}} \leq u_i^{\mathtt{H}}$ for all $\tau_i \in \Gamma_H$:

$$\sum_{\tau_i \in \Gamma_H} \theta_i^{\mathtt{H}} \leq m - U_{\mathtt{L}}^{\mathtt{H}}$$

(Since $\theta_i^{\mathtt{H}} = u_i^{\mathtt{H}} + u_i^{\mathtt{L}} \cdot \dfrac{u_i^{\mathtt{H}} - \theta_i^{\mathtt{L}}}{\theta_i^{\mathtt{L}} - u_i^{\mathtt{L}}}$ from Eq. (19))

$$\Leftrightarrow \sum_{\tau_i \in \Gamma_H} (u_i^{\mathtt{H}} + u_i^{\mathtt{L}} \cdot \frac{u_i^{\mathtt{H}} - \theta_i^{\mathtt{L}}}{\theta_i^{\mathtt{L}} - u_i^{\mathtt{L}}}) \leq m - U_{\mathtt{L}}^{\mathtt{H}}$$

$$\Leftrightarrow U_{\mathtt{H}}^{\mathtt{H}} + \sum_{\tau_i \in \Gamma_H} u_i^{\mathtt{L}} \cdot \frac{u_i^{\mathtt{H}} - \theta_i^{\mathtt{L}}}{\theta_i^{\mathtt{L}} - u_i^{\mathtt{L}}} \leq m - U_{\mathtt{L}}^{\mathtt{H}}$$

(Since $\theta_i^{\mathtt{L}} = \mathcal{F}_{i-1} \cdot \overline{u}_i^{\mathtt{L}}$ under the worst-case and $\mathcal{F}_{i-1} \geq \mathcal{F}_0$ from Eq. (9) and $\overline{u}_i^{\mathtt{L}} \geq u_i^{\mathtt{L}}$ from Eq. (2), we have $\theta_i^{\mathtt{L}} = \mathcal{F}_{i-1} \cdot \overline{u}_i^{\mathtt{L}} \geq \mathcal{F}_0 \cdot \overline{u}_i^{\mathtt{L}} \geq \mathcal{F}_0 \cdot u_i^{\mathtt{L}}$)

$$\Leftarrow U_{\mathtt{H}}^{\mathtt{H}} + \sum_{\tau_i \in \Gamma_H} u_i^{\mathtt{L}} \cdot \frac{u_i^{\mathtt{H}} - \mathcal{F}_0 \cdot \overline{u}_i^{\mathtt{L}}}{\mathcal{F}_0 \cdot u_i^{\mathtt{L}} - u_i^{\mathtt{L}}} \leq m - U_{\mathtt{L}}^{\mathtt{H}}$$

$$\Leftrightarrow \frac{\sum_{\tau_i \in \Gamma_H} u_i^{\mathtt{H}} - \mathcal{F}_0 \cdot \sum_{\tau_i \in \Gamma_H} \overline{u}_i^{\mathtt{L}}}{\mathcal{F}_0 - 1} \leq m - U_{\mathtt{L}}^{\mathtt{H}} - U_{\mathtt{H}}^{\mathtt{H}}$$

$$\Leftrightarrow \frac{U_{\mathtt{H}}^{\mathtt{H}} - \mathcal{F}_0 \cdot \overline{U}_{\mathtt{H}}^{\mathtt{L}}}{\mathcal{F}_0 - 1} \leq m - U_{\mathtt{L}}^{\mathtt{H}} - U_{\mathtt{H}}^{\mathtt{H}}$$

(Since $\mathcal{F}_0 = \dfrac{m - U_{\mathtt{L}}^{\mathtt{L}}}{\overline{U}_{\mathtt{H}}^{\mathtt{L}}}$ from Eq. (8))

$$\Leftrightarrow \frac{\overline{U}_{\mathtt{H}}^{\mathtt{L}} \cdot U_{\mathtt{H}}^{\mathtt{H}} - (m - U_{\mathtt{L}}^{\mathtt{L}}) \cdot \overline{U}_{\mathtt{H}}^{\mathtt{L}}}{m - U_{\mathtt{L}}^{\mathtt{L}} - \overline{U}_{\mathtt{H}}^{\mathtt{L}}} \leq m - U_{\mathtt{L}}^{\mathtt{H}} - U_{\mathtt{H}}^{\mathtt{H}}$$

$$\Leftrightarrow \frac{\overline{U}_{\mathtt{H}}^{\mathtt{L}} \cdot (U_{\mathtt{H}}^{\mathtt{H}} + U_{\mathtt{L}}^{\mathtt{L}} - m)}{m - U_{\mathtt{L}}^{\mathtt{L}} - \overline{U}_{\mathtt{H}}^{\mathtt{L}}} \leq m - U_{\mathtt{L}}^{\mathtt{H}} - U_{\mathtt{H}}^{\mathtt{H}}$$

(Since $ms \geq (U_{\mathtt{H}}^{\mathtt{H}} + U_{\mathtt{L}}^{\mathtt{H}})$ and $ms \geq (U_{\mathtt{L}}^{\mathtt{L}} + \overline{U}_{\mathtt{H}}^{\mathtt{L}})$ from Eq. (20)–(21))

$$\Leftarrow \overline{U}_{\mathtt{H}}^{\mathtt{L}} \cdot (U_{\mathtt{H}}^{\mathtt{H}} + U_{\mathtt{L}}^{\mathtt{L}} - m) \leq (m - ms)^2$$

(Since Eq. (23) holds, we have $(U_{\mathtt{L}}^{\mathtt{L}} + U_{\mathtt{H}}^{\mathtt{H}}) \leq 2 \cdot m \cdot s - \overline{U}_{\mathtt{H}}^{\mathtt{L}}$)

$$\Leftarrow \overline{U}_{\mathtt{H}}^{\mathtt{L}} \cdot (2 \cdot m \cdot s - \overline{U}_{\mathtt{H}}^{\mathtt{L}} - m) \leq (m - ms)^2$$

$$\Leftrightarrow \overline{U}_{\mathtt{H}}^{\mathtt{L}} \cdot ((2s-1)m - \overline{U}_{\mathtt{H}}^{\mathtt{L}}) \leq (m - ms)^2$$

(Since $\overline{U}_{\mathtt{H}}^{\mathtt{L}} \cdot ((2s-1)m - \overline{U}_{\mathtt{H}}^{\mathtt{L}})) \leq ((2s-1)m/2)^2$ from Lemma 10)

$$\Leftarrow ((2s-1)m/2)^2 \leq (m - ms)^2$$

$$\Leftarrow (2s-1)m/2) \leq (m - ms)$$

$$\Leftrightarrow (2s-1)/2) \leq (1 - s)$$

$$\Leftrightarrow (2s-1) \leq (2-2s) \;\Leftrightarrow\; 4s \leq 3 \;\Leftrightarrow\; s \leq 3/4$$

Therefore, if $s \leq 3/4$, the sum of the HI-critical execution rates is not larger than m and MCFQ algorithm in Figure 2 returns success. ◀

B Extension of MC-Fluid and MCF for IMC tasks

The IMC task set Γ is transformed to a non-IMC task set $\overline{\Gamma}$ in which all the original parameters of each of the tasks in Γ remains the same in $\overline{\Gamma}$ except that the LO- and HI-critical execution time $C_i^{\mathtt{L}}$ and $C_i^{\mathtt{H}}$ of each LO-critical task $\tau_i \in \overline{\Gamma}$ is reduced by $C_i^{\mathtt{H}}$. In other words, each LO-critical task in $\overline{\Gamma}$ has $\overline{C_i^{\mathtt{L}}} = C_i^{\mathtt{L}} - C_i^{\mathtt{H}}$ and $\overline{C_i^{\mathtt{H}}} = 0$. We use the MC-Fluid/MCF algorithm to find the execution rates $\overline{\theta}_i^{\mathtt{L}}$ and $\overline{\theta}_i^{\mathtt{H}}$ considering a multiprocessor platform with total capacity $\overline{m} = (m - U_{\mathtt{L}}^{\mathtt{H}})$ using the non-IMC task set $\overline{\Gamma}$.

The execution rates $\theta_i^{\mathtt{L}}$ and $\theta_i^{\mathtt{H}}$ for the original IMC tasks in set Γ are set as follows: if τ_i is a LO-critical task, then $\theta_i^{\mathtt{L}} = \overline{\theta}_i^{\mathtt{L}} + u_i^{\mathtt{H}}$ and $\theta_i^{\mathtt{H}} = \overline{\theta}_i^{\mathtt{H}} + u_i^{\mathtt{H}}$; otherwise, ($\tau_i$ is a HI-critical task) $\theta_i^{\mathtt{L}} = \overline{\theta}_i^{\mathtt{L}}$ and $\theta_i^{\mathtt{H}} = \overline{\theta}_i^{\mathtt{H}}$. It can be proved (based on the same proof technique in [2]) that this extension is correct for scheduling IMC task sets.

Replica-Aware Co-Scheduling for Mixed-Criticality*

Eberle A. Rambo[1] and Rolf Ernst[2]

1 **Technische Universität Braunschweig, Braunschweig, Germany**
`rambo@ida.ing.tu-bs.de`
2 **Technische Universität Braunschweig, Braunschweig, Germany**
`ernst@ida.ing-tu-bs.de`

Abstract

Cross-layer fault-tolerance solutions are the key to effectively and efficiently increase the reliability in future safety-critical real-time systems. Replicated software execution with hardware support for error detection is a cross-layer approach that exploits future many-core platforms to increase reliability without resorting to redundancy in hardware. The performance of such systems, however, strongly depends on the scheduler. Standard schedulers, such as Partitioned Strict Priority Preemptive (SPP) and Time-Division Multiplexing (TDM)-based ones, although widely employed, provide poor performance in face of replicated execution. In this paper, we propose the replica-aware co-scheduling for mixed-critical systems. Experimental results show schedulability improvements of more than 1.5x when compared to TDM and 6.9x when compared to SPP.

1998 ACM Subject Classification C.4 Performance of Systems

Keywords and phrases replicated execution, scheduling, fault tolerance, real-time systems

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.20

1 Introduction

Technology downscaling has increased the hardware's overall susceptibility to errors to the point where they became non-negligible [15]. Hence, current and future computing systems must be appropriately designed to cope with errors in order to provide a reliable service and correct functionality. Specially in the real-time mixed-criticality domain, where applications with different requirements and criticalities co-exist in the system, which must provide *sufficient independence* and prevent error propagation (e.g. timing, data corruption) between criticalities [17, 28]. Recent examples are complex applications such as Flight Management Systems (FMS) and Advanced Driver Assistance Systems (ADAS) in the avionics and automotive domains, respectively [17, 28]. In this paper, we address the timing aspect of software execution protected from soft errors.

Soft errors, more specifically Single Event Effects (SEEs), are transient faults abstracted as *bit-flips* in hardware and can be caused by alpha particles, energetic neutrons from cosmic radiation and process variability [11, 15]. Depending on where and when they occur, their impact on software execution range from masked (no observable effect) to a complete system crash [5, 8, 9]. To handle such errors, the approaches can vary from completely software-based to completely hardware-based. The former are able to cover only part of the errors [9, 8] and the latter result in costly redundant hardware [15], as currently seen in lock-step

* This work was supported in parts by the German Research Foundation (DFG) as part of the priority program "Dependable Embedded Systems" (SPP 1500 – `http://spp1500.itec.kit.edu`).

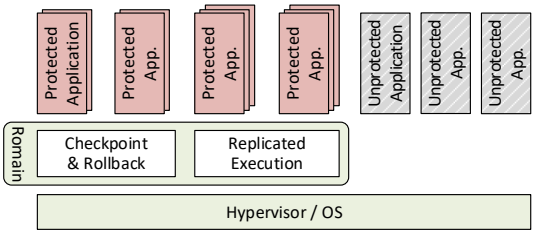


Figure 1 ASTEROID's fault-tolerance architecture: the software side.

dual-core execution [21]. We focus on a more effective and efficient cross-layer approach, which distributes the tasks of detecting errors and recovering from them in different layers of software and hardware [15, 9, 8].

A cross-layer fault-tolerance solution for mixed-criticality has been developed in the ASTEROID project. It increases the reliability at a higher level of abstraction without resorting to hardware redundancy [3, 8]. ASTEROID's architecture is illustrated in Fig. 1. The reliable software execution is realized by the operating system service Romain [8]. Mixed-critical applications may co-exist in the system and are translated into protected and unprotected applications. Romain replicates the protected applications and manage their execution. Error detection is realized by a set o mechanisms whose main feature is the hardware assisted state comparison, which compares the replicas' state at certain points in time [3, 8]. Error recovery strategies can vary depending on whether the application is running in Dual Modular Redundancy (DMR) or Triple Modular Redundancy (TMR) [3, 5].

The performance of replicated execution has been analyzed in [4] and revised in [2]. The work supports Partitioned Strict Priority Preemptive (SPP) scheduling, where tasks are mapped to arbitrary cores, and assumes a single error model. The authors found that SPP, although widely employed in real-time systems, provides very pessimistic response time bounds for replicated tasks. Depending on the interfering workload, replicated tasks executing serially (on the same core) present much better performance than when executing in parallel (on distinct cores). That occurs due to the long time that replicated tasks potentially have to wait on each core to synchronize and compare states before resuming execution. That leads to very low resource utilization and prevents the use of replicated execution in practice.

In this paper, we explore co-scheduling to provide small response times for replicated tasks without hindering the remaining unprotected tasks. Co-scheduling is a technique that schedules interacting tasks/threads to execute simultaneously on different cores [22]. It allows tasks/threads to communicate more efficiently by reducing the time they are blocked during synchronization. In contrast to SPP [4, 2], our approach drastically minimizes delays due to the implicit synchronization found in state comparisons. In contrast to gang scheduling [10], it rules out starvation and distributes the execution of replicas in time to achieve small response times of unprotected tasks. Finally, our approach differs from standard Time-Division Multiplexing (TDM) and TDM with background partition [18] in that all tasks have formal guarantees.

The major **contribution** of this paper is the replica-aware co-scheduling for mixed-critical systems. A formal Worst-Case Response Time (WCRT) analysis under a single error assumption is included. In contrast to related work, it supports different recovery strategies and accounts for the Network-on-Chip (NoC) communication delay and overheads due to replica management and state comparison. Experimental results with benchmark applications show an improvement on taskset schedulability of up to 6.9x when compared to SPP [2], and 1.5x when compared to a TDM-based scheduler.

2 Related Work

L4/Romain [8] is a cross-layer fault-tolerance approach that provides reliable software execution under soft errors. Romain provides protection at the application-level by replicating and managing the applications' executions as an operating system service. The error detection is realized by a set of mechanisms [3, 8, 9] whose main feature is the hardware assisted state comparison, which allows an effective and efficient comparison of the replicas' states. Pipeline fingerprinting [3] provides a checksum of the retired instructions and the pipeline's data path in every processor, detecting errors in the execution flow and data. The state comparison, reduced to comparing checksums instead of data structures, is carried out at certain points in time. It must occur at least when the application is about to externalize its state e.g. in a *syscall* [8]. The replica generated *syscalls* are intercepted by Romain, have their integrity checked and their replicas' states compared before being allowed to externalize the state [8].

Mixed-criticality, in the context of this paper, is supported with different levels of protection for applications with different criticalities and requirements (unprotected, protected with DMR[1]or TMR) and by ensuring that timing constraints are met even in case of errors. For instance, Romain provides different error recovery strategies [3, 5]:

- *DMR with checkpoint and rollback*: to recover, the replicas rollback to their last valid state and re-execute;
- *TMR with state copy*: to recover, the state of the faulty replica is replaced with the state of one of the healthy replicas.

In this work, we focus on the system-level timing aspect of errors affecting the applications. We assume thereby the absence of failures in critical components [9, 24], such as the Operating System (OS), the replica manager/voter (e.g. Romain) and interconnect (e.g. NoC), which can be protected as in [16, 26].

The WCRT of replicated execution has been analyzed in [4], where replicas are modeled as fork-join tasks in a system implementing Partitioned SPP. The work was later revised in [2] due to optimism in the original approach. The revised approach is used in this work. In that approach, with deadline monotonic priority assignment, where the priority of tasks decrease as their deadlines increase, replicated tasks perform worse when mapped in parallel than when mapped to the same core. This is due to the state comparisons during execution, which involves implicit synchronization between cores. With partitioned scheduling, in the worst-case, the synchronization ends up accumulating the interference from all cores to which the replicated task is mapped, resulting in poor performance in higher loads. On the other hand, mapping replicated tasks to the highest priorities results in long response times for lower priority tasks and rules out deadline monotonicity. The latter causes the unschedulability of all tasksets with at least one regular task whose deadline is shorter than the execution time of a replicated task.

Gang scheduling [10] is a co-scheduling variant that schedules groups of interacting tasks/threads simultaneously. It increases performance by reducing the inter-thread communication latency. The authors in [19] present an integration between gang scheduling and Global Earliest Deadline First (EDF), called the Gang EDF. They provide a schedulability analysis derived from the Global EDF's based on the sporadic task model. In another work, [12] shows that SPP Gang schedulers in general are not predictable, for instance, due to

[1] DMR *per se* can be used for system integrity only. However, DMR augmented with checkpointing and rollback enables recovery and can be used to achieve integrity and availability (state rollback followed by re-execution in both replicas) [3, 5].

priority inversions and slack utilization. In the context of real-time systems, gang scheduling has not received much attention.

TDM-based scheduling [18] is widely employed to achieve predictability and ensure temporal-isolation. Tasks are allocated to partitions, which are scheduled to execute in time slots. Partitions can span across several (or all) cores and can be executed at the same time. The downside of TDM is that it is not work-conserving and underutilizes system resources. A TDM variant with background partition [18] tackles this issue by allowing low priority tasks to execute in other partitions whenever no higher priority workload is executing. Yet, in addition to the high cost to switch between partitions, no guarantees can be given to tasks in the background partition.

In this work, we exploit co-scheduling with SPP to improve the performance of the system. Our work differs from [4] in that replicas are treated as gangs and are mapped with highest priorities, and are hence activated simultaneously on different cores. In contrast to gang-scheduling [10, 12] and to [4], the execution of replicas is distributed in time with offsets to compensate for the lack of deadline monotonicity thus allowing the schedulability of tasks with short deadlines. We further provide for the worst-case performance of lower priority tasks by allowing them to execute whenever no higher priority workload is executing. However, in contrast to [18], all tasks have WCRT guarantees. Moreover, we also model the state comparison and the on-Chip communication overheads, and although we use Romain as example, the model can also be applied to other approaches.

3 Preliminaries

In this work, we use the Compositional Performance Analysis (CPA) [14] to provide formal response time bounds. Let us introduce the system, task and error models.

3.1 System Model

The system consists of a standard NoC-based many-core composed of processing elements, simply referred to as cores.

There are two types of tasks in our system, as in [2]:

- *independent* tasks τ_i: regular, unprotected tasks; and
- *fork-join* tasks Γ_i: replicated, protected tasks.

The system implements partitioned scheduling, where the operating system manages tasks statically mapped to cores. The mapping is assumed to be given as input. The scheduling policy is a combination of SPP and gang scheduling. When executing only independent tasks, the system's behavior is identical to Partitioned SPP, where tasks are scheduled independently on each core according to SPP. It differs from SPP, when scheduling fork-join tasks.

Fork-join tasks are mapped with highest priorities, hence do not suffer interference from independent tasks, and execute simultaneously on different cores, as in gang scheduling. Note that deadline monotonicity is therefore only partially possible. To limit the interference to independent tasks, the execution of a fork-join task is divided in smaller intervals called stages, whose executions are distributed in time. At the end of each stage, the states of the replicas are compared. In case of an error, i.e. states differ, recovery is triggered.

Fork-join stages are executed with static offsets [23] in execution slots. One stage is executed per slot. On a core with n fork-join tasks, there are $n + 1$ execution slots: one slot for each fork-join task Γ_i and one slot for recovery. The slots are cyclically scheduled in a cycle Φ. The slot for Γ_i starts at offset $\phi(\Gamma_i)$ relative to the start of Φ and ends after $\varphi(\Gamma_i)$,

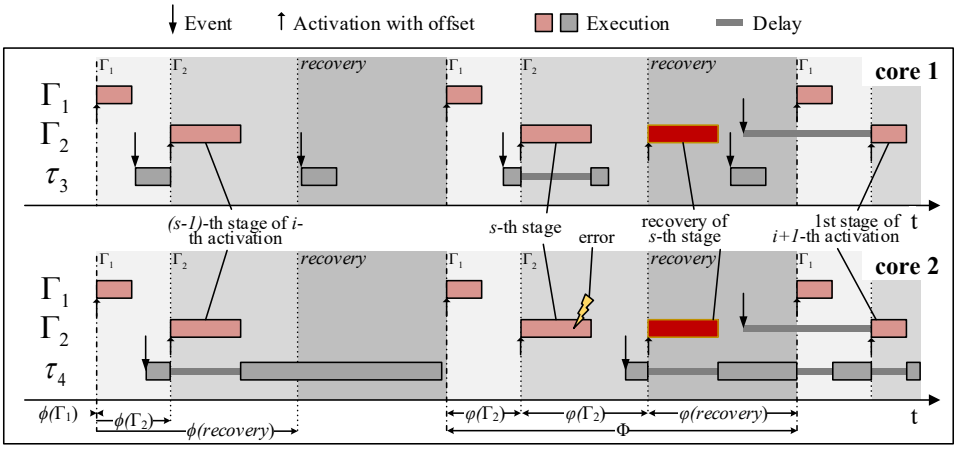


Figure 2 Execution example with two fork-join and two independent tasks on two cores.

the slot length. The recovery slot is shared by all fork-join tasks on that core and is where error recovery may take place under a single error assumption (details in Sec. 3.3 and 4.3). The recovery slot has an offset $\phi(recovery)$ relative to Φ and length $\varphi(recovery)$. Lower priority independent tasks are allowed to execute whenever no higher priority workload is executing.

An example is shown in Fig. 2, where two fork-join tasks Γ_1 and Γ_2 and two independent tasks τ_3 and τ_4 are mapped to two cores. Γ_1 and Γ_2 execute in their respective slots simultaneously in both cores. When an error occurs, the recovery of Γ_2 is scheduled and the recovery of the error-affected stage occurs in the recovery slot. The use of offsets enables the schedulability of independent tasks with short periods and deadlines, such as τ_3 and τ_4. Note that, without the offsets, Γ_1 and Γ_2 would execute back-to-back leading to the unschedulability of τ_3 and τ_4.

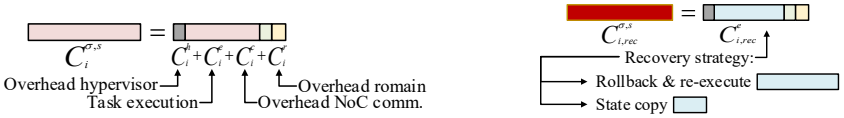
3.2 Task Model

An independent task τ_i is mapped to core σ with a priority p. Once activated, it executes for at most C_i, its Worst-Case Execution Time (WCET). The activations of a task are modeled with arbitrary *event models*. Task activations in an event model are given by arrival curves $\eta^-(\Delta t)$ and $\eta^+(\Delta t)$, which return the minimum and maximum number of events arriving in any time interval Δt. Their pseudo-inverse counterparts $\delta^+(q)$ and $\delta^-(q)$ return the maximum and minimum time interval between the first and last events in any sequence of q event arrivals. Conversion is provided in [27]. Periodic events with jitter, sporadic events and others can be modeled with the minimum distance function $\delta_i^-(q)$ as follows [27]:

$$\delta_i^-(q) = \max((q-1)\cdot d^{min}, (q-1)\cdot \mathcal{P} - \mathcal{J}) \tag{1}$$

where $\mathcal{P}$ is the period, $\mathcal{J}$ is the jitter, d^{min} is the minimum distance between any two events, and the subscript i indicates the association with a task τ_i or Γ_i.

Fork-join tasks are rigid parallel tasks, i.e. the number of processors required by a fork-join task is fixed and specified externally to the scheduler [12], and consist of multiple stages with data dependencies, as in [2, 1]. A fork-join task Γ_i is a Directed Acyclic Graph (DAG) $G(V, E)$, where vertices in V are subtasks and edges in E are precedence dependencies [2]. In the graph, tasks are partitioned in *segments* and *stages*, as illustrated in Fig. 4a. A subtask $\tau_i^{\sigma,s}$ is the s-th stage of the σ-th segment and is annotated with its WCET $C_i^{\sigma,s}$. The WCET



(a) WCET of a fork-join subtask **(b)** WCET of recovery

Figure 3 The composition of WCET of fork-join subtasks.

of a stage is equal across all segments, i.e. $\forall x, y\colon C_i^{x,s}{=}C_i^{y,s}$. Each segment σ of Γ_i is mapped to a distinct core. A fork-join task Γ_i is annotated with the *static offset* $\phi(\Gamma_i)$, which marks the start of its execution slot in Φ. The offset also admits a small positive jitter j_ϕ, to account for a slight desynchronization between cores and context switch overhead.

The activations of a fork-join task are modeled with *event models*. Once Γ_i is activated, its stages are successively activated by the completion of all segments of the previous stage, as in [2, 1]. Our approach differs from them in that it restricts the scheduling of at most one stage of Γ_i in a cycle Φ, and the stage receives service at the offset $\phi(\Gamma_i)$. Note that the event arrival at a fork-join task is not synchronized with its offset. The events at a fork-join task are queued at the first stage and only one event at a time is processed (FIFO) [2]. A queued event is admitted when the previous event leaves the last stage.

The interaction with Romain (the voter) is modeled in the analysis as part of the WCET $C_i^{\sigma,s}$, as depicted in Fig. 3a. The WCET includes the on-Chip communication latency and state comparison overheads, as the Romain instance may be mapped to an arbitrary core. Those can be obtained e.g. with [25] along with task mapping and scheduler properties to avoid over-conservative interference estimation and obtain tighter bounds.

3.3 Error Model

Our model assumes a single error scenario caused by SEEs (cf. Sec. 1). We assume that all errors affecting fork-join tasks can be detected and contained, ensuring integrity. The overhead of error detection mechanisms are modeled as part of the WCET (cf. Fig. 3a). Regarding independent tasks, we assume that an error immediately leads to a task failure and assume also that its failure will not violate the WCRT guarantees of the remaining tasks. Those assumptions are met e.g. by Romain[2]. Moreover, we assume the absence of failures in critical components [9, 24], such as the OS, the replica manager/voter Romain and the interconnect (e.g. the NoC), which can be protected as in [16, 26].

Our model provides recovery[2] for fork-join tasks, ensuring their availability. With a recovery slot in every cycle Φ, our approach is able to handle up to one error per cycle Φ. However, the analysis in Sec. 4.3 assumes at most one error per busy window for the sake of a simpler analysis (the concept will be introduced in Sec. 4). The assumption is reasonable since the probability of a multiple error scenario is very low and can be considered as an acceptable risk [17]. A multiple error scenario occurs only if an error affects more than one replica at a time or if more than one error occurs within the same busy window.

[2] Romain is able to detect and recover from all soft errors affecting user-level applications. For details on the different error impacts and detection strategies, the interested reader can refer to [3, 8].

3.4 Offsets

The execution of fork-join tasks in our approach is based on static offsets, which are assumed to be provided as input to the scheduler. The offsets form execution slots whose size do not vary during runtime, as seen in Fig. 2. Varying the slots sizes would substantially increase the timing analysis complexity without a justifiable performance gain. The offsets must satisfy two constraints:

▶ **Constraint 1.** *A slot for a fork-join task Γ_i must be large enough to fit the largest stage of Γ_i. That is, $\forall s, \sigma$: $\varphi(\Gamma_i) \geq C_i^{\sigma,s} + j_\phi$.*

▶ **Constraint 2.** *The recovery slot must be large enough to fit the recovery of the largest stage of any fork-join task mapped to that core. That is, $\forall i, s, \sigma$: $\varphi(recovery) \geq C_{i,rec}^{\sigma,s} + j_\phi$.*

where a one error scenario per cycle is assumed and $C_{i,rec}^{\sigma,s}$ is the recovery WCET of subtask $\tau_i^{\sigma,s}$ (cf. Sec. 4.3).

We provide basic offsets that satisfy Constraints 1 and 2. The calculation must consider only overlapping fork-join tasks, i.e. fork-join tasks mapped to at least one core in common. Offsets for non-overlapping fork-join tasks are computed separately as they do not interfere directly with each other. The indirect interference, e.g. in the NoC, are accounted for in the WCETs. First we determine the smallest slots that satisfy Constraint 1:

$$\forall \Gamma_i : \varphi(\Gamma_i) = \max_{\forall \sigma, s} \{C_i^{\sigma,s}\} + j_\phi \tag{2}$$

and the smallest recovery slot that satisfies Constraint 2:

$$\varphi(recovery) = \max_{\forall \Gamma_i, \tau_j^{\sigma,s} \in \Gamma_i} \left\{C_{i,rec}^{\sigma,s}\right\} + j_\phi \,. \tag{3}$$

The cycle Φ is then the sum of all slots:

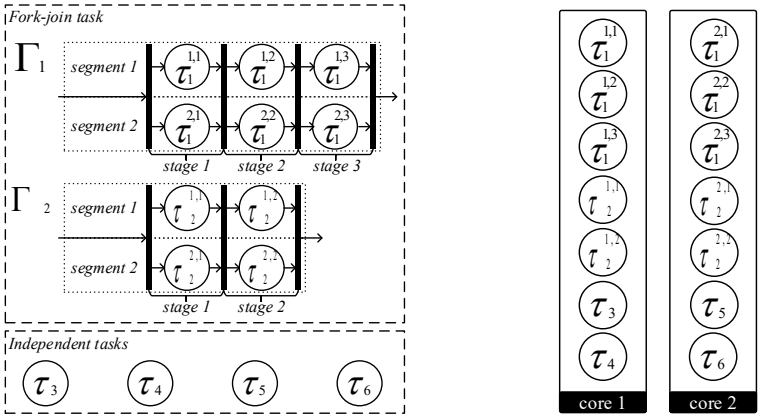
$$\Phi = \sum_{\forall \Gamma_i} \{\varphi(\Gamma_i)\} + \varphi(recovery) \,. \tag{4}$$

The offsets then depend on the order in which the slots are placed inside Φ. Assuming that the slots $\phi(\Gamma_i)$ are sorted in ascending order on i and that the recovery slot is the last one, the offsets are obtained by:

$$\phi(x) = \begin{cases} 0 & \text{if } x = \Gamma_1 \\ \phi(\Gamma_{i-1}) + \varphi(\Gamma_{i-1}) & \text{if } x = \Gamma_i \text{ and } i > 1 \\ \Phi - \varphi(recovery) & \text{if } x = recovery \end{cases} \tag{5}$$

4 Response-Time Analysis

The analysis is based on CPA and inspired by [2, 23]. In CPA, the WCRT is calculated with the busy window approach [29]. The response time of an event of a task τ_i (resp. Γ_i) is the time interval between the event arrival and the completion of its execution. In the busy window approach [29], the event with the WCRT can be found inside the busy window. The busy window w_i of a task τ_i (resp. Γ_i) is the time interval where all response times of the task depend on the execution of at least one previous event in the same busy window, except for the task's first event. The busy window starts at a critical instant corresponding to the worst-case scheduling scenario. Since the worst-case scheduling scenario depends on the type of task, it will be derived individually in the sequel.



(a) Taskset **(b)** Mapping

Figure 4 A taskset with 4 independent tasks and 2 fork-join tasks, and its mapping to 2 cores. Highest priority at the top, lowest at the bottom.

Before we derive the analysis for fork-join and for independent tasks, let us introduce the example in Fig. 4 used throughout the section. The taskset consists of 4 independent tasks and 2 fork-join tasks, mapped to two cores. The task priority on each core decreases from top to bottom (e.g. $\tau_1^{1,1}$ has the highest priority and τ_4 the lowest).

4.1 Fork-Join Tasks

We now derive the WCRT for an arbitrary fork-join task Γ_i. Therefor, we need to identify the critical instant leading to the worst-case scheduling scenario. In case of SPP, the critical instant is when all tasks are activated at the same time and the tasks' subsequent events arrive as early as possible [29]. In our case, the critical instant must also account for the use of static offsets [23].

The worst-case scheduling scenario for Γ_2 on core 1 is illustrated in Fig. 5. Γ_2 is activated and executed at the same time on cores 1 and 2 (omitted). Note that, by design, fork-join tasks do not dynamically interfere with each other. The *critical instant* occurs when the first event of Γ_2 arrives just after missing Γ_2's offset. The event has to wait until the next cycle to be served, which takes time $\Phi + j_\phi$ when the activation with offset is delayed by a jitter j_ϕ. Notice that the WCETs of fork-join tasks already account for the inter-core communication and synchronization overhead (cf. Fig. 3a).

▶ **Lemma 1.** *The critical instant leading to the worst-case scheduling scenario of a fork-join task Γ_i is when the first event of Γ_i arrives just after missing Γ_i's offset $\phi(\Gamma_i)$.*

Proof. A fork-join task Γ_i does not suffer interference from independent tasks or other fork-join tasks. The former holds since independent tasks always have lower priority. The latter holds due to three reasons: an arbitrary fork-join task Γ_j always receives service in its slot $\phi(\Gamma_j)$; the slot $\phi(\Gamma_j)$ is large enough to fit Γ_j's largest subtask (Constraint 1); and the slots in a cycle Φ are disjoint. Thus, the critical instant can only be influenced by Γ_i itself.

We prove by contradiction. Suppose that there is another scenario worse than Lemma 1. That means that the first event can arrive at a time that causes a delay to Γ_i larger than $\Phi + j_\phi$. However, if the delay is larger than $\Phi + j_\phi$, then the event arrived before a previous slot $\phi(\Gamma_i)$ and Γ_i did not receive service. Since that can only happen if there is a pending

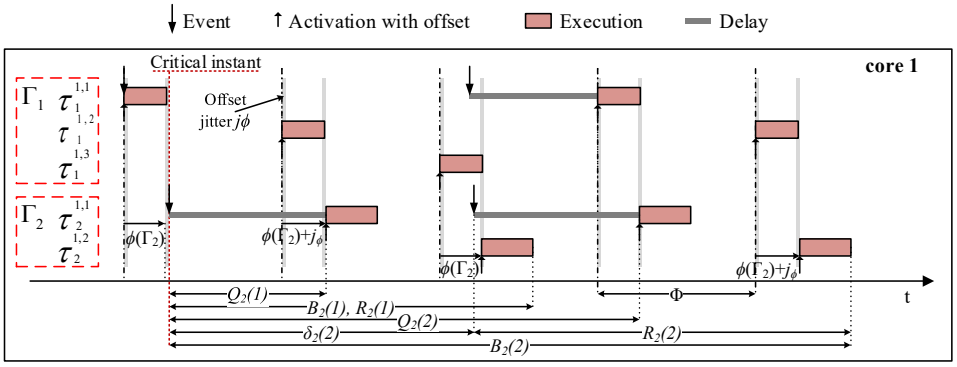


Figure 5 Worst-case schedule for fork-join gang Γ_2 on core 1 (cf. Fig. 4).

activation of Γ_i and thus violates the definition of a busy window, the hypothesis must be rejected. ◀

Let us now derive the Multiple-Event Queueing Delay $Q_i(q)$ and Multiple-Event Busy Time $B_i(q)$ on which the busy window relies. $Q_i(q)$ is the longest time interval between the arrival of Γ_i's first activation and the first time its q-th activation receives service, considering that all events belong to the same busy window [2, 20]. For Γ_i, the q-th activation can receive service at the next cycle Φ after the execution of $q-1$ activations of Γ_i lasting $s_i \cdot \Phi$ each, a delay Φ (cf. Lemma 1) and a jitter j_ϕ. This is given by:

$$Q_i(q) = (q-1) \cdot s_i \cdot \Phi + \Phi + j_\phi \tag{6}$$

where s_i is the number of stages of Γ_i and Φ is the cycle.

▶ **Lemma 2.** *The Multiple-Event Queueing Delay $Q_i(q)$ given by Eq. 6 is an upper bound.*

Proof. The proof is by induction. When $q=1$, Γ_i has to wait for service at most until the next cycle Φ plus an offset jitter j_ϕ to get service for its first stage, considering that the event arrives just after its offset (Lemma 1). In a subsequent $q+1$-th activation in the same busy window, Eq. 6 must also consider q entire executions of Γ_i. Since Γ_i has s_i stages and only one stage can be activated and executed per cycle Φ, it takes additional $s_i \cdot \Phi$ for each activation of Γ_i, resulting in Eq. 6. ◀

The Multiple-Event Busy Time $B_i(q)$ is the longest time interval between the arrival of Γ_i's first activation and the completion of its q-th activation, considering that all events belong to the same busy window [2, 20]. The q-th activation of Γ_i completes after a delay Φ (cf. Lemma 1), a jitter j_ϕ and the execution of q activations of Γ_i. This is given by:

$$B_i(q) = q \cdot s_i \cdot \Phi + j_\phi + C_i^{\sigma,s} \tag{7}$$

where $C_i^{\sigma,s}$ is the WCET of Γ_i's last stage.

▶ **Lemma 3.** *The Multiple-Event Busy Time $B_i(q)$ given by Eq. 7 is an upper bound.*

Proof. The proof is by induction. When $q=1$, Γ_i has to wait for service at most until the next cycle Φ plus an offset jitter j_ϕ to get service for its first stage (Lemma 1), plus the completion of the last stage of the activation lasting $(s_i-1) \cdot \Phi + C_i^{\sigma,s}$. This is given by:

$$\begin{aligned} B_i(1) &= (s_i - 1) \cdot \Phi + \Phi + j_\phi + C_i^{\sigma,s} \\ &= s_i \cdot \Phi + j_\phi + C_i^{\sigma,s} \end{aligned} \tag{8}$$

In a subsequent $q+1$-th activation in the same busy window, Eq. 7 must consider q additional executions of Γ_i. Since Γ_i has s_i stages and only one stage can be activated and executed per cycle Φ, it takes additional $s_i \cdot \Phi$ for each activation of Γ_i. Thus, Eq. 7. ◀

Now we can calculate the busy window and WCRT of Γ_i. The busy window w_i of a fork-join task Γ_i is given by:

$$w_i = \max_{q \geq 1,\, q \in \mathbb{N}} \{B_i(q) \,|\, Q_i(q+1) \geq \delta_i^-(q+1)\}\,. \tag{9}$$

▶ **Lemma 4.** *The busy window is upper bounded by Eq. 9.*

Proof. The proof is by contradiction. Suppose there is a busy window $\breve{w}_i$ longer than w_i. In that case, $\breve{w}_i$ must contain at least one activation more than w_i, i.e. $\breve{q} \geq q+1$. From Eq. 9, we have that $Q_i(\breve{q}) < \delta_i^-(\breve{q})$, i.e. $\breve{q}$ is not delayed by the previous activation. Since that violates the definition of a busy window, the hypothesis must be rejected. ◀

The response time $R_i(q)$ of the q-th activation of Γ_i in the busy window is given by:

$$R_i(q) = B_i(q) - \delta_i^-(q)\,. \tag{10}$$

The worst-case response time R_i^+ is the longest response time of any activation of Γ_i observed in the busy window.

$$R_i^+ = \max_{1 \leq q \leq \eta_i^+(w_i)} R_i(q)\,. \tag{11}$$

▶ **Theorem 5.** *R_i^+ (Eq. 11) provides an upper bound on the worst-case response time of an arbitrary fork-join task Γ_i.*

Proof. The WCRT of a fork-join task Γ_i is obtained with the busy window approach [29]. It remains to prove that the critical instant leads to the worst-case scheduling scenario, that the interference captured in Eqs. 6 and 7 are upper bounds, and that the busy window is correctly captured by Eq. 9. These are proved in Lemmas 1, 2, 3, and 4, respectively. ◀

4.2 Independent Tasks

We now derive the WCRT analysis of an arbitrary independent task τ_i. Two types of interference affect independent tasks: interference caused by higher priority independent tasks and by fork-join tasks. Let us first identify the critical instant leading to the worst-case scheduling scenario where τ_i suffers the most interference.

▶ **Lemma 6.** *The critical instant of τ_i is when the first event of higher priority independent tasks arrive simultaneously with τ_i's event at the offset of a fork-join task.*

Proof. The worst-case interference caused by a higher priority (independent) task τ_j under SPP is when its first event arrives simultaneously with τ_i's and continue arriving as early as possible [29].

The interference caused by a fork-join task Γ_j on τ_i depends on Γ_j's offset $\phi(\Gamma_j)$ and subtasks $\tau_j^{\sigma,s}$, whose execution times vary for different stages s. Assume a critical instant that occurs at a time other than at the offset $\phi(\Gamma_j)$. Since a task Γ_j starts receiving service at its offset, an event of τ_i arriving at time $t > \phi(\Gamma_j)$ can only suffer less interference from Γ_j's subtask than when arriving at $t = 0$. ◀

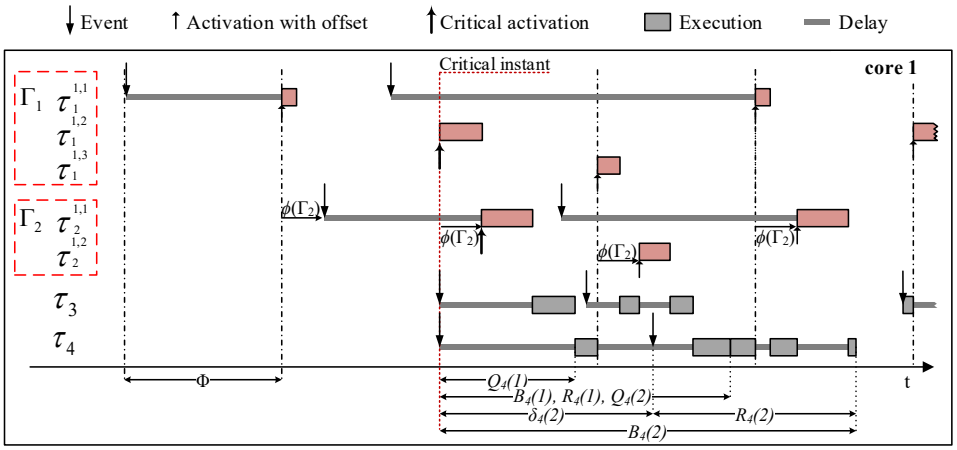


Figure 6 The worst-case schedule for independent task τ_4 on core 1 (cf. Fig. 4).

Fork-join subtasks have different execution times for different stages, which leads to a number of scheduling scenarios that must be evaluated [23]. Each scenario is defined by the fork-join subtasks that will receive service in the cycle Φ and the offset at which the critical instant supposedly occurs. The scenario is called a critical instant candidate S. Since independent tasks participate in all critical instant candidates, they are omitted in S for the sake of simplicity.

▶ **Definition 7.** Critical Instant Candidate S: the critical instant candidate S is an ordered pair (a, b) where a is a critical offset and b is a tuple containing one subtask $\tau_j^{\sigma,s}$ of every interfering fork-join task Γ_j.

Let us also define the set of candidates that must be evaluated.

▶ **Definition 8.** Critical Instant Candidate Set $\mathcal{S}$: the set containing all possible different critical instant candidates S.

The worst-case schedule of the independent task τ_4 from the example in Fig. 4 is illustrated in Fig. 6. In fact, the critical instant leading to τ_4's WCRT is at $\phi(\Gamma_1)$ when $\tau_1^{1,2}$ and $\tau_2^{1,1}$ receive service at the same cycle Φ, i.e. $S = (\phi(\Gamma_1), (\tau_1^{1,2}, \tau_2^{1,1}))$. Events of the independent task τ_3 start arriving at the critical instant and continue arriving as early as possible.

Let us now bound the interference $I_i^I(\Delta t)$ caused by equal or higher priority independent tasks in any time interval Δt. The interference $I_i^I(\Delta t)$ can be upper bounded as follows [20]:

$$I_i^I(\Delta t) = \sum_{\forall \tau_j \in hp_I(i)} \eta_j^+(\Delta t) \cdot C_j \tag{12}$$

where $hp_I(i)$ is the set of equal or higher priority independent tasks mapped to the same core as τ_i.

To derive the interference caused by fork-join tasks we need to define the Critical Instant Event Model. The critical instant event model $\check{\eta}_i^{\sigma,s}(\Delta t, S)$ of a subtask $\tau_i^{\sigma,s} \in \Gamma_i$ returns the maximum number of activations observable in any time interval Δt, assuming the critical instant S. It can be derived from Γ_i's input event model $\eta_i^+(\Delta t)$ as follows:

$$\check{\eta}_i^{\sigma,s}(\Delta t, S) = \min\left\{\eta_i^+\left(\Delta t_S + \Phi - \phi(\Gamma_i)\right), \psi\right\} - gt\left(s^S, s, \phi^S, \phi(\Gamma_i)\right) \tag{13}$$

$$\psi = \left\lfloor \frac{\Delta t_S}{\Phi \cdot s_i} \right\rfloor + ge\big(\Delta t_S \bmod (\Phi \cdot s_i)\,,\, \Phi \cdot (s-1)\big) \tag{14}$$

$$\Delta t_S = \Delta t + \underbrace{\Phi \cdot (s^S - 1)}_{\substack{\text{critical instant} \\ \text{stage}}} + \underbrace{\phi^S}_{\substack{\text{critical instant} \\ \text{offset}}} \tag{15}$$

where s is the stage of subtask $\tau_i^{\sigma,s}$; s_i is the number of stages in Γ_i; ϕ^S is the offset in S; s^S is the stage of Γ_i in S; $gt(a,b,c,d)$ is a function that returns 1 when $(a > b) \vee (a = b \wedge c > d)$, 0 otherwise; and $ge(a,b)$ is a function that returns 1 when $a \geq b$, 0 otherwise.

▶ **Lemma 9.** *$\check{\eta}_i^{\sigma,s}(\Delta t, S)$ (Eq. 13) provides a valid upper bound on the number of activations of $\tau_i^{\sigma,s}$ observable in any time interval Δt, assuming the critical instant S.*

For the sake of readability, the proof is presented in Appendix A.

The interference $I_i^{FJ}(\Delta t, S)$ caused by fork-join tasks on the same core in any time interval Δt, assuming a critical instant candidate S, can then be upper bounded as follows:

$$I_i^{FJ}(\Delta t, S) = \sum_{\forall \tau_j^{\sigma,s} \in hp_{FJ}(i)} \check{\eta}_j^{\sigma,s}(\Delta t, S) \cdot C_j^{\sigma,s} \tag{16}$$

where $hp_{FJ}(i)$ is the set of fork-join subtasks mapped to the same core as τ_i.

The Multiple-Event Queueing Delay $Q_i(q,S)$ and Multiple-Event Busy Time $B_i(q,S)$ for an independent task τ_i, assuming a critical instant candidate S, can be derived as follows.

$$Q_i(q,S) = (q-1) \cdot C_i + I_i^I(Q_i(q,S)) + I_i^{FJ}(Q_i(q,S), S) \tag{17}$$

$$B_i(q,S) = q \cdot C_i + I_i^I(B_i(q,S)) + I_i^{FJ}(B_i(q,S), S) \tag{18}$$

where $q \cdot C_i$ is the time required to execute q activations of task τ_i.

Eqs. 17 and 18 result in fixed-point problems, similar to the well known busy window equation (Eq. 9). They can be solved iteratively, starting with a very small, positive ϵ.

▶ **Lemma 10.** *The Multiple-Event Queueing Delay $Q_i(q,S)$ given by Eq. 17 is an upper bound, assuming the critical instant S.*

Proof. The proof is by induction. When $q=1$, τ_i has to wait for service until the interfering workload is served. The interfering workload is given by Eqs. 12 and 16. Since $\eta_j^+(\Delta t)$ and C_j are upper bounds by definition, Eq. 12 is also an upper bound. Similarly, since $\check{\eta}_j^{\sigma,s}(\Delta t, S)$ is an upper bound (cf. Lemma 9) and $C_j^{\sigma,s}$ is an upper bound by definition, 16 is an upper bound for a given S. Therefore, $Q_i(1,S)$ is also an upper bound, for a given S.

In a subsequent $q+1$-th activation in the same busy window, $Q_i(q,S)$ also must consider q executions of τ_i. This is captured in Eq. 17 by the first term, which is, by definition, an upper bound on the execution time. From that, Lemma 10 follows. ◀

▶ **Lemma 11.** *The Multiple-Event Busy Time $B_i(q,S)$ given by Eq. 18 is an upper bound, assuming the critical instant S.*

Proof. The proof is similar to Lemma 10, except that $B_i(q, S)$ in Eq. 18 also captures the completion of the q-th activation. It takes additional C_i, which is an upper bound by definition. Thus Eq. 18 is an upper bound, for a given S. ◀

The busy window $w_i(q, S)$ of an independent task τ_i is given by:

$$w_i(S) = \max_{q \geq 1,\, q \in \mathbb{N}} \{B_i(q, S) \mid Q_i(q{+}1, S) \geq \delta_i^-(q{+}1)\} \tag{19}$$

▶ **Lemma 12.** *The busy window is upper bounded by Eq. 19.*

Proof. The proof is by contradiction. Suppose there is a busy window $\breve{w}_i(S)$ longer than $w_i(S)$. In that case, $\breve{w}_i(S)$ must contain at least one activation more than $w_i(S)$, i.e. $\breve{q} \geq q{+}1$. From Eq. 19, we have that $Q_i(\breve{q}, S) < \delta_i^-(\breve{q})$, i.e. $\breve{q}$ is not delayed by the previous activation. Since that violates the definition of a busy window, the hypothesis must be rejected. ◀

The response time R_i of the q-th activation of a task in a busy window is given by:

$$R_i(q, S) = B_i(q, S) - \delta_i^-(q) \tag{20}$$

Finally, the worst-case response time R_i^+ is found inside the busy window and must be evaluated for all possible critical instant candidates $S \in \mathcal{S}$. The worst-case response time R_i^+ is given by:

$$R_i^+ = \max_{S \in \mathcal{S}} \left\{ \max_{1 \leq q \leq \eta_i^+(w_i(S))} \{R_i(q, S)\} \right\} \tag{21}$$

where the set $\mathcal{S}$ is given by the following Cartesian products:

$$\mathcal{S} = \big\{\phi(\Gamma_j), \phi(\Gamma_k), \dots\big\} \times \big\{\sigma_i(\Gamma_j) \times \sigma_i(\Gamma_k) \times \dots\big\} \tag{22}$$

where $\Gamma_j, \Gamma_k, \dots$ are all fork-join tasks mapped to the same core as τ_i and $\sigma_i(\Gamma_j)$ is the set of subtasks of Γ_j that are mapped to that core. When no fork-join tasks interfere with τ_i, $\mathcal{S} = \{(0, ())\}$.

▶ **Theorem 13.** *R_i^+ (Eq. 21) returns an upper bound on the worst-case response time of an independent task τ_i.*

Proof. We must first prove that, for a given S, R_i^+ is an upper bound. R_i^+ is obtained with the busy window approach [29]. It returns the maximum response time $R_i(q, S)$ among all activations inside the busy window. From Lemmas 10 and 11 we have that Eqs. 17 and 18 are upper bounds for a given S. From Lemma 12 we have that the busy window is captured by Eq. 19. Since the first term of Eq. 20 is an upper bound and the second term is a lower bound by definition, $R_i(q, S)$ is an upper bound. Thus R_i^+ is an upper bound for a given S. Since Eq. 21 evaluates the maximum response time over all $S \in \mathcal{S}$, R_i^+ is an upper bound on the response time of τ_i. ◀

4.3 Error Recovery

Designed for mixed-criticality, our approach supports different recovery strategies for different fork-join tasks (cf. Sec. 2). For instance, in DMR augmented with checkpointing and rollback, recovery consists in reverting the state and re-executing the error-affected stage in both replicas. In TMR, recovery consists in copying and replacing the state of the faulty replica with the state of a healthy one. The different strategies are captured in the analysis by the

recovery execution time, which depends on the strategy and the stage to be recovered. The recovery WCET $C^{\sigma,s}_{i,rec}$ of a fork-join subtask $\tau^{\sigma,s}_i$ accounts for the adopted recovery strategy as illustrated in Fig. 3b. Once an error is detected, error recovery is triggered and executed in the recovery slot of the same cycle Φ. Fig. 2 illustrates the recovery of the s-th stage of Γ_2's i-th activation.

Let us incorporate the error recovery into the analysis. For a fork-join task Γ_i, we must only adapt the Multiple-Event Busy Time $B_i(q)$ (Eq. 7) to account for the execution of the recovery:

$$B^{rec}_i(q) = q \cdot s_i \cdot \Phi + j_\phi + \phi(recovery) - \phi(\Gamma_i) + C^{\sigma,s}_{i,rec} \tag{23}$$

where $C^{\sigma,s}_{i,rec}$ is the WCET of the recovery of last subtask of Γ_i. The recovery of another task Γ_j does not interfere with Γ_i's WCRT. Only the recovery of one of Γ_i's subtasks can interfere with Γ_i's WCRT. Moreover, since the recovery of a subtask occurs in the recovery slot of the same cycle Φ and does not interfere with the next subtask, only the recovery of the last stage of Γ_i actually has an impact on its response time. This is captured by the three last terms of Eq. 23.

For an independent task τ_i, the worst-case impact of recovery of a fork-join task Γ_j is modelled as an additional fork-join task Γ_{rec} with one subtask $\tau^{\sigma,1}_{rec}$ mapped to the same core as τ_i and that executes in the *recovery* slot. The WCET $C^{\sigma,1}_{rec}$ of $\tau^{\sigma,1}_{rec}$ is chosen as the maximum recovery time among the subtasks of all fork-join tasks mapped to that core:

$$C^{\sigma,1}_{rec} = \max_{\forall \tau^{\sigma,s}_j \in hp_{FJ}(i)} \left\{ C^{\sigma,s}_{i,rec} \right\} \tag{24}$$

With Γ_{rec} mapped, Eq. 21 finds the critical instant where the recovery $C^{\sigma,1}_{rec}$ has the worst impact on the response time of τ_i.

5 Experimental Evaluation

In our experiments we evaluate our approach with real as well as synthetic workload, focusing on the performance of the scheduler. First we characterize MiBench applications [13] and evaluate them as fork-join (replicated) tasks in the system. Then we evaluate the performance of independent (regular) tasks. Finally we evaluate the approach with synthetic workload when varying parameters of fork-join tasks.

5.1 Evaluation with benchmark applications

5.1.1 Characterization

First we extract execution times and number of stages from MiBench automotive and security applications [13]. They were executed with small input on an ARMv7@1GHz and a DDR3-1600 [7]. Table 1 summarizes the total WCET, *observed* number of stages and WCET of the longest stage (max). A stage is delimited by *syscalls* (cf. Sec. 2). We report the observed execution times as WCETs. As pointed out in [2], stages vary on number and execution time depending on the application and on the current activity in that stage (computation/IO). This is seen e.g. in *susan*, where 99% of the WCET is concentrated in one stage (computation) while the other stages perform mostly IO and are on average 3.34us long.

In our approach, the optimal is when all stages of a fork-join task have the same WCET. There are two possibilities to achieve that: to *split* very long stages in smaller ones or to *group* small subsequent stages together. We exploit the latter as it does not require changes

Table 1 MiBench applications' profile.

	WCET [ms]	Observed stages		Grouped stages	
		#stages	max WCET [ms]	#stages	max WCET [ms]
basicmath	32.48	19738	0.02	5	6.50
bitcount	24.42	30	15.16	3	15.16
susan	9.63	12	9.59	1	9.63
blowfish	0.11	7	0.09	1	0.11
rijndael	13.17	93	0.37	3	5.91
sha	3.49	51	0.11	2	1.90

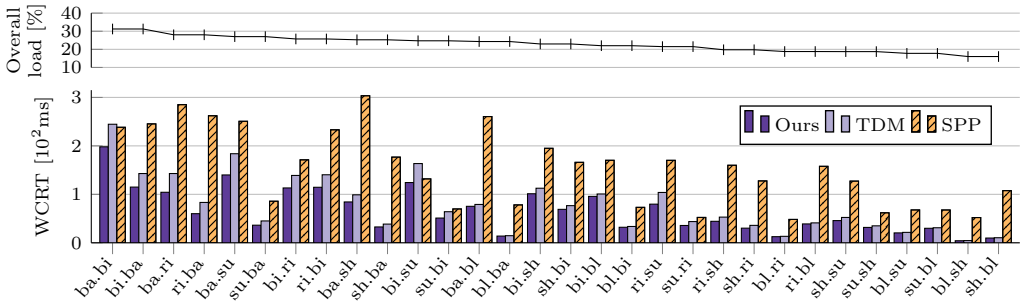


Figure 7 WCRT of fork-join tasks with two segments derived from MiBench.

to the error detection mechanism or to our model. The results with *grouped* stages are shown on the right-hand side of Table 1. We have first grouped stages without increasing the maximum stage length. The largest improvement is seen in *bitcount*, where the number of stages reduces in one order of magnitude. In cases where all stages are very short, we increase the maximum stage length. When increasing the maximum stage length in two orders of magnitude, the number of stages of *basicmath* reduces in four orders of magnitude. We have manually chosen the maximum stage length. Alternatively the problem of finding the maximum stage length can be formulated as an optimization problem that e.g. minimizes the overall WCRT or maximizes the slack. Next, we map the applications as fork-join tasks and evaluate their WCRTs.

5.1.2 Evaluation of fork-join tasks

Two applications at a time are mapped as fork-join tasks with two segments (i.e. replicas in DMR) to two cores (cf. Fig. 4). On each core, 15% load is introduced by ten independent tasks generated with UUniFast [6]. We compare our approach with a TDM-based scheduler and Axer's Partitioned SPP [2]. In TDM, each fork-join task executes (and recovers) in its own slot. Independent tasks execute in a third slot, which replaces the recovery slot of our approach. The size of the slots are derived from our offsets. For all approaches, the priority assignment for independent tasks is deadline monotonic and considers that deadline equals period. In SPP, the deadline monotonic priority assignment also includes fork-join tasks.

The results are plotted in Fig. 7, where *ba.bi* gives the WCRT of *basicmath* when mapped together with *bitcount*. Despite the low system load, our approach also outperforms SPP in all cases, with bounds 58.2% lower, on average. Better results with SPP cannot be obtained unless the interfering workload is removed or highest priority is given to the fork-join tasks [2], which violates DM. Despite the similarity of how our approach handles fork-join tasks with TDM, the proposed approach outperforms TDM in all cases, achieving,

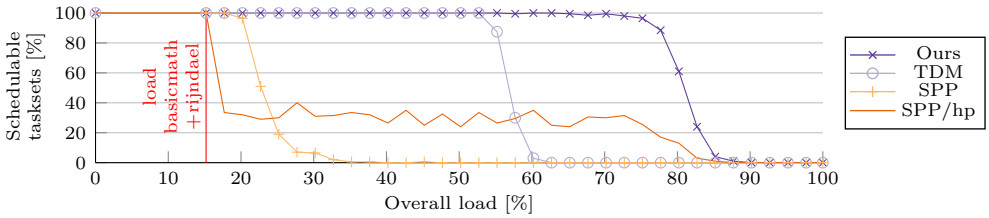
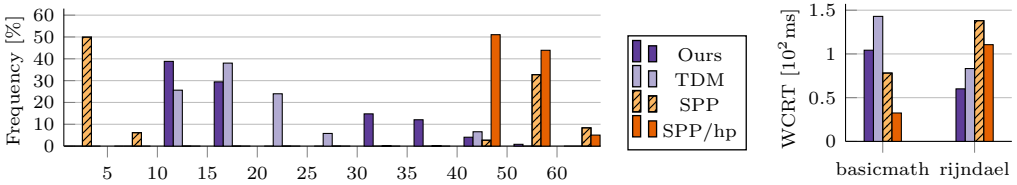


Figure 8 Schedulability as a function of the load of the system. *Basicmath* and *rijndael* as fork-join tasks with two segments.



(a) WCRT of independent tasks [ms] **(b)** WCRT of FJ tasks

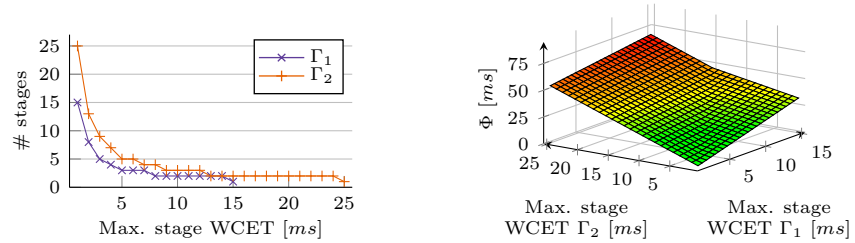
Figure 9 *Basicmath* and *rijndael* as replicated tasks in DMR running on a dual-core configuration with 20.2% load (5% load from independent tasks).

on average, bounds 13.9% lower. This minor difference is because TDM slots must be slightly longer than our offsets to fit an eventual recovery. Nonetheless, not only our approach can guarantee small WCRT for replicated tasks but also provides for the worst-case performance of independent tasks.

5.1.3 Evaluation of independent tasks

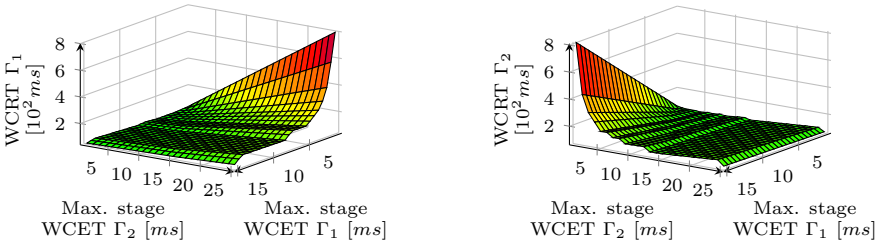
In a second experiment we fix *bitcount* and *rijndael* as fork-join tasks and vary the load on both cores. The generated task periods are in the range [20,500] ms, larger than the longest stage of the fork-join tasks. The schedulability of the system as the load increases is shown in Fig. 8. Our approach outperforms TDM and SPP in all cases, scheduling 1.55x and 6.96x more tasksets, respectively. Due to its non-work conserving characteristic, TDM's schedulability is limited to medium loads. SPP provides very small response times with lower loads but, as the load increases, the schedulability drops fast due to high interference (and thus high WCRT) suffered by fork-join tasks. For reference purposes, we also plot the schedulability of SPP when assigning the highest priorities to the fork-join tasks (SPP/hp). The schedulability in higher loads improves but losing deadline monotonicity guarantees renders the systems unusable in practice. Moreover, when increasing the jitter to 20% (relative to period), schedulability decreases 14.2% but shows the same trends for all schedulers.

Fig. 9 details the tasks' WCRTs when the system load is 20.2%. Indeed, when schedulable, SPP provides some of the smallest WCRTs for independent tasks, and SPP/hp improves the response times of fork-join tasks at the expense of the independent tasks'. Our approach provides a balanced trade-off between the performance of independent tasks and of fork-join tasks, and achieves high schedulability even in higher loads.



(a) Stages of Γ_1 and Γ_2 **(b)** Cycle Φ

Figure 10 Parameters of two fork-join tasks Γ_1 and Γ_2 with two segments running on a dual-core configuration.



(a) WCRT of Γ_1 **(b)** WCRT of Γ_2

Figure 11 Performance of fork-join tasks Γ_1 and Γ_2 as a function of the maximum stage WCET.

5.2 Evaluation with synthetic workload

We now evaluate the performance of our approach when varying parameters such as stage length and cycle Φ.

5.2.1 Evaluation of fork-join tasks

Two fork-join tasks Γ_1 and Γ_2 with two segments each (i.e. replicas in DMR) are mapped to two cores. The total WCETs[3] of Γ_1 and Γ_2 are 15 and 25ms, respectively. Both tasks are sporadic, with a minimum distance of 1s between activations. The number of stages of Γ_1 and Γ_2 is varied as a function of the maximum stage WCET, as depicted in Fig. 10a. The length of the cycle Φ, depicted in Fig. 10b, varies with the maximum stage WCET since it is derived from them (cf. Sec. 3.4).

The system performance as the maximum stage lengths of Γ_1 and Γ_2 increase is reported in Fig. 11. The WCRT of Γ_1 increases with the stage length (Fig. 11a) as it depends on the number of stages and Φ's length. In fact, the WCRT of Γ_1 is longest when the stages of Γ_1 are the shortest and the stages of the interfering fork-join task (Γ_2) are the longest. Conversely, WCRT of Γ_1 is shortest when its stages are the longest and the stages of the interfering fork-join task are the shortest. The same occurs to Γ_2 in Fig. 11b. Thus, there is a trade-off between the response times of interfering fork-join tasks. This is plotted in Fig. 12 as the sum of the WCRTs of Γ_1 and Γ_2. As can be seen in Fig. 12, low response times can be obtained next and above to the line segment between the origin $(0, 0, 0)$ and the point $(15, 25, 0)$, the total WCETs[1] of Γ_1 and Γ_2 respectively.

[3] The sum of the WCET of all stages of a fork-join task.

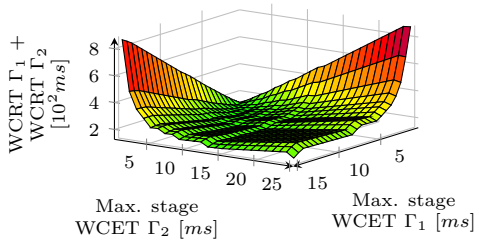
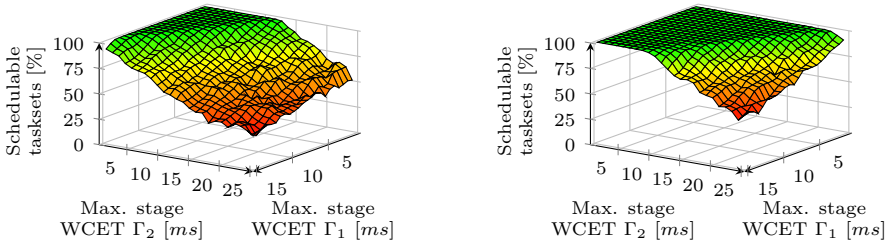


Figure 12 WCRT trade-off between interfering fork-join tasks.



(a) Task period interval [15-500] ms

(b) Task period interval [25-500] ms

Figure 13 Schedulable tasksets as a function of the maximum stage WCET of fork-join tasks Γ_1 and Γ_2 with 25% load from independent tasks.

5.2.2 Evaluation of independent tasks

To evaluate the impact of the parameters on independent tasks, we extend the previous scenario introducing 25% load on each core with ten independent tasks generated with UUniFast [6]. The task periods are within the interval [15,500] ms for the first experiment, and the interval [25,500] ms for the second. The priority assignment is deadline monotonic and considers that the deadline is equal to the period.

The schedulability as a function of the stage lengths is shown in Fig. 13. Sufficiently long stages cause the schedulability to decrease as independent tasks with short periods start missing their deadlines. This is seen in Fig. 13a when the stage length of either fork-join task reaches 15ms, the minimum period for the generated tasksets. Thus, when increasing the minimum period of generated tasks to 25ms, the number of schedulable tasksets also increases (Fig. 13b).

The maximum stage length of a fork-join task has direct impact on the response times and schedulability of the system. For the sake of performance, shorter stage lengths are preferred. However, that is not always possible because it would result in a large number of stages or because of the application, which restricts the minimum stage length (cf. Sec. 5.1.1). Nonetheless, fork-join tasks still are able to perform well with appropriate parameter choices. Additionally, one can formulate the problem of finding the stage lengths according to an objective function, such as minimize the overall response time or maximize the slack. The offsets can also be included in the formulation, as long as Constraints 1 and 2 are met.

6 Conclusion

In this paper, we presented the replica-aware co-scheduling for mixed-critical systems, where applications with different requirements and criticalities co-exist. The work includes a formal WCRT analysis supporting different recovery strategies and accounting for the NoC

communication delay and overheads due to replica management and state comparison. Our approach provides for high worst-case performance of replicated software execution on many-core architectures without impairing the remaining tasks in the system. Experimental results with benchmark applications showed an improvement on taskset schedulability of up to 6.9x when compared to Partitioned SPP and 1.5x when compared to a TDM-based scheduler.

Naturally, there is always the possibility of critical components failing due to errors (replica manager or voter and the OS). In that case, either enough time for a reboot must be allocated or the critical components must be hardened.

Acknowledgement. We would like to thank Adam Lackorzynski and Tobias Stumpf for the helpful discussions.

References

1. Björn Andersson and Dionisio de Niz. Analyzing Global-EDF for multiprocessor scheduling of parallel tasks. In *International Conference On Principles Of Distributed Systems*, pages 16–30. Springer, 2012.
2. Philip Axer. *Performance of Time-Critical Embedded Systems under the Influence of Errors and Error Handling Protocols*. PhD thesis, TU Braunschweig, 2015.
3. Philip Axer, Rolf Ernst, Björn Döbel, and Hermann Härtig. Designing an analyzable and resilient embedded operating system. In *Proc. on Software-Based Methods for Robust Embedded Systems*, Braunschweig, Germany, 2012.
4. Philip Axer, Sophie Quinton, Moritz Neukirchner, Rolf Ernst, Bjorn Dobel, and Hermann Hartig. Response-time analysis of parallel fork-join workloads with real-time constraints. In *Proc. of ECRTS'13*, 2013.
5. Philip Axer, Maurice Sebastian, and Rolf Ernst. Reliability analysis for mpsocs with mixed-critical, hard real-time constraints. In *Proc. Intl. Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2011.
6. Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.
7. Nathan Binkert, Bradford Beckmann, Gabriel Black, et al. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2), August 2011.
8. Björn Döbel, Hermann Härtig, and Michael Engel. Operating system support for redundant multithreading. In *Proc. of EMSOFT'12*, 2012.
9. Michael Engel and Björn Döbel. The reliable computing base-a paradigm for software-based reliability. In *GI-Jahrestagung*, pages 480–493, 2012.
10. Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306 – 318, 1992. doi:10.1016/0743-7315(92)90014-E.
11. Rémi Gaillard. Single event effects: Mechanisms and classification. In Michael Nicolaidis, editor, *Soft Errors in Modern Electronic Systems*. Springer US, 2011.
12. Joël Goossens and Vandy Berten. Gang ftp scheduling of periodic and parallel rigid real-time tasks. *arXiv preprint arXiv:1006.2617*, 2010.
13. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC-4. 2001*, Dec 2001.
14. R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System Level Performance Analysis–the SymTA/S Approach. *IEE Proceedings-Computers and Digital Techniques*, 152, 2005.

15 Andreas Herkersdorf et al. Resilience Articulation Point (RAP): Cross-layer dependability modeling for nanometer system-on-chip resilience. *Microelectronics Reliability*, 54(6–7):1066–1074, 2014. doi:10.1016/j.microrel.2013.12.012.

16 M. Hoffmann, F. Lukas, C. Dietrich, and D. Lohmann. dOSEK: the design and implementation of a dependability-oriented static embedded kernel. In *Proc. of RTAS'15*, pages 259–270, 2015. doi:10.1109/RTAS.2015.7108449.

17 International Standards Organization. *ISO 26262: Road Vehicles – Functional Safety*, 2011.

18 Robert Kaiser and Stephan Wagner. Evolution of the PikeOS microkernel. In *First International Workshop on Microkernels for Embedded Systems*, 2007.

19 Shinpei Kato and Yutaka Ishikawa. Gang EDF scheduling of parallel task systems. In *Proc. of RTSS'09*, 2009.

20 J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proc. of RTSS'90*, 1990.

21 NXP MPC577xK Ultra-Reliable MCU Family. [online]. Available: http://www.nxp.com/assets/documents/data/en/fact-sheets/MPC577xKFS.pdf, 2017.

22 John K. Ousterhout. Scheduling techniques for concurrent systems. In *ICDCS*, volume 82, pages 22–30, 1982.

23 J. C. Palencia and M. Gonzalez Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proc. of RTSS'98*, 1998. doi:10.1109/REAL.1998.739728.

24 Eberle A. Rambo and Rolf Ernst. Providing flexible and reliable on-chip network communication with real-time constraints. In *1st International Workshop on Resiliency in Embedded Electronic Systems (REES)*, 2015.

25 Eberle A. Rambo, Selma Saidi, and Rolf Ernst. Providing formal latency guarantees for ARQ-based protocols in networks-on-chip. In *Proc. of DATE'16*, 2016.

26 Eberle A. Rambo, Christoph Seitz, Selma Saidi, and Rolf Ernst. Designing networks-on-chip for high assurance real-time systems. In *Proc. of PRDC'17*, 2017.

27 K. Richter. *Compositional Scheduling Analysis Using Standard Event Models*. PhD thesis, TU Braunschweig, 2005.

28 RTCA Incorporated. *DO-254: Design Assurance Guidance For Airborne Electronic Hardware*, 2000.

29 K. W. Tindell, A. Burns, and A. J. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2), 1994.

A Additional Proofs

For the sake of readability, the proof of Lemma 9 (Sec. 4.2) is presented here.

Proof. The proof is by induction, in two parts. First let us assume $s^S = 1$ and $\phi^S = 0$, neutral values resulting in $\Delta t_S = \Delta t$ and $gt(s^S, s, \phi^S, \phi(\Gamma_i)) = 0$. The maximum number of activations of $\tau_i^{\sigma,s}$ seen in the interval Δt is limited by the maximum number of activations of the fork-join task Γ_i because a subtask $\tau_i^{\sigma,s}$ is activated once per Γ_i's activation, and limited by the maximum number of times that $\tau_i^{\sigma,s}$ can actually be scheduled and served in Δt. This is ensured in Eq. 13 by the minimum function and its first and second terms, respectively.

When $s^S > 1$ and/or $\phi^S > 0$, the time interval $[0, \Delta t)$ must be moved forward so that it starts at stage s^S and offset ϕ^S. This is captured by Δt_S in Eq. 15 and by the last term of Eq. 13. The former extends the end of the time interval by the time it takes to reach the stage s^S and the offset ϕ^S, i.e. $[0, \Delta t_S)$. The latter pushes the start of the interval forward by subtracting an activation of $\tau_i^{\sigma,s}$ if it occurs before the stage s^S and the offset ϕ^S, resulting in the interval $[\Delta t_S - \Delta t, \Delta t_S)$. Thus Eq. 13. ◀

Thermal Implications of Energy-Saving Schedulers

Sandeep M. D'souza[1] and Ragunathan (Raj) Rajkumar[2]

1 **Carnegie Mellon University, Pittsburgh, PA, USA**
`sandeepd@andrew.cmu.edu`
2 **Carnegie Mellon University, Pittsburgh, PA, USA**
`rajkumar@andrew.cmu.edu`

Abstract

In many real-time systems, continuous operation can raise processor temperature, potentially leading to system failure, bodily harm to users, or a reduction in the functional lifetime of a system. Static power dominates the total power consumption, and is also directly proportional to the operating temperature. This reduces the effectiveness of frequency scaling and necessitates the use of sleep states. In this work, we explore the relationship between energy savings and system temperature in the context of fixed-priority *energy-saving* schedulers, which utilize a processor's *deep-sleep* state to save energy. We derive insights from a well-known thermal model, and are able to identify *proactive* design choices which are *independent* of system constants and can be used to reduce processor temperature. Our observations indicate that, while energy savings are key to lower temperatures, not all energy-efficient solutions yield low temperatures. Based on these insights, we propose the *SysSleep* and *ThermoSleep* algorithms, which enable a thermally-effective sleep schedule. We also derive a lower bound on the optimal temperature achievable by energy-saving schedulers. Additionally, we discuss partitioning and task phasing techniques for multi-core processors, which require all cores to synchronously transition into deep sleep, as well as those which support independent deep-sleep transitions. We observe that, while energy optimization is straightforward in some cases, the dependence of temperature on partitioning and task phasing makes temperature minimization non-trivial. Evaluations show that compared to the existing purely energy-efficient design methodology, our proposed techniques yield lower temperatures along with significant energy savings.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

Keywords and phrases thermal analysis, real-time scheduling

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.21

1 Introduction

Computationally-intensive real-time applications are becoming ubiquitous. Autonomous vehicles are a prime example where, computational requirements are driven by the need to process streams of data from multiple sensors. Advancements in semiconductor technology have enabled such applications by increasing the number of transistors available to system designers. However, the side effects of rising transistor density include increased power and heat dissipation [23]. Hence, continuous operation may cause the temperature of a processor to exceed its operating limits, forcing it to reduce its frequency or shut down. This in turn can lead to missed deadlines, and possibly catastrophic failure. Similarly, violating thermal constraints in implantable medical devices can cause bodily harm [8]. Moreover, it is critical that the components used in such systems perform reliably over their lifetime. System temperature is one of the key factors which influence reliability. High temperatures degrade

the system reliability over a period of time [31][32], and a 10-15°C difference in operating temperature can result in a 2x difference in the lifespan of a device [32].

Energy savings and system temperature are intricately tied together. Modern processors are equipped with energy-management features such as Dynamic Voltage and Frequency Scaling (DVFS) [35], and the use of low-power sleep states [28]. DVFS enables the processor to change its operating frequency and voltage, thereby reducing *dynamic switching power*, while low-power sleep states use power gating and/or clock gating [3] to reduce *static leakage power* dissipation when the processor is idle. As transistor geometries get smaller, the dominance of static power as a contributor to total power consumption is only expected to increase [22]. Since static power is also directly dependent on operating temperature, scheduling techniques will increasingly need to take advantage of processor sleep states.

1.1 Contributions of the Paper

In this work, we analyze the thermal properties of Energy-Saving (ES) Schedulers [12], which utilize the processor's deep-sleep state. Our contributions are as follows:

1. We analyze the thermal performance of ES Schedulers using the well-known thermal model based on Fourier's Law, and derive design choices to pro-actively (i.e. *a priori*) minimize the maximum temperature for both uni-core and multi-core processors.
2. We present the *SysSleep* algorithm to maximize the time the processor can be in deep sleep, and the *ThermoSleep* heuristic that yields a thermally-effective sleep schedule.
3. We derive a lower bound on the optimal maximum temperature achievable by ES Schedulers.
4. We propose task-partitioning heuristics that significantly reduce the maximum temperature for multi-core processors using ES Schedulers.
5. We analyze the impact of phasing each core's forced-sleep task on temperature, in the context of multi-core processors where cores can independently transition into deep sleep.

1.2 Background

We now introduce the background material and notation relevant to our work. Consider a task set Γ consisting of n independent[1] periodic real-time tasks $\tau_1, \tau_2, ..., \tau_n$. Each task $\tau_i \in \Gamma$ is characterized by $\{C_i, T_i, D_i\}$, where C_i is the worst-case execution time, T_i is the period, and D_i is the relative deadline from its arrival time. We assume that for each task $D_i = T_i$, i.e., deadlines are implicit. The utilization of a task τ_i is given by $U_i = C_i/T_i$ and task priorities are assigned using the rate-monotonic scheduling policy [27]. The task set is listed in non-increasing order of task priorities such that $T_1 \leq T_2 \leq .. \leq T_n$. Each task has an initial arrival time (or phase) of ϕ_i, such that its arrival times are $\phi_i, \phi_i + T_i, \phi_i + 2T_i, ...$ Without loss of generality, we assume that the initial arrival time of task $\tau_1, \phi_1 = 0$.

The following Energy-Saving (ES) Schedulers have been defined in [28] and [12]: Energy-Saving Rate-Harmonized Scheduling+ [28][12] (ES-RHS+), Energy-Saving Rate-Monotonic Scheduling [12] (ES-RMS) and Energy-Saving Deadline-Monotonic Scheduling [12] (ES-DMS). These techniques are characterized by a high-priority periodic Energy-Saver task (also referred to as an ES-task or forced-sleep task) τ_{sleep}, which puts the processor into an uninterrupted deep sleep for a duration $C_{sleep} \geq C_{SleepMin}$ every period $T_{sleep} \leq T_1$. This ensures that the ES-task executes at the highest priority in accordance with the Rate-Monotonic (RM) [27]

[1] Task release jitter and task dependence can be incorporated using the frameworks proposed in [6] and [30], and are beyond the scope of this work.

priority assignment. If any idle durations precede and are contiguous with the ES-task, they can be used to put the processor into deep sleep [12]. $C_{SleepMin}$ is a system constraint that represents the minimum round-trip time required for the processor to go into the deep-sleep state and return back to the active state. We assume that $C_{SleepMin}$ captures the overhead involved in transitioning to deep sleep. While using ES Schedulers, the processor can be in one of the following states:

- *Busy:* The processor is executing a task $\tau_i \in \Gamma$.
- *Forced Sleep:* The processor is forced into *deep sleep* by the *Energy-Saver* task τ_{sleep}.
- *Idle:* The processor is neither *busy* nor in *forced sleep*.

For ES Schedulers, the generalized worst-case response time test for a task τ_i is given by the following recurrence relation:

$$W_0 = C_i, W_{k+1} = C_i + \left\lceil \frac{W_k}{T_{sleep}} \right\rceil C_{sleep} + \sum_{j=1}^{i-1} \left\lceil \frac{W_k}{T_j} \right\rceil C_j \tag{1}$$

where, W_{k+1} is the worst-case response time of the task τ_i. If $W_{k+1} \leq D'_i$, then τ_i will be schedulable, otherwise τ_i will miss its deadline, where, D'_i is the *generalized deadline* of a task τ_i and depends on the type of ES Scheduler used. Based on this notation, we briefly describe each of the ES Schedulers:

(1) *ES-RMS*: Tasks execute as per rate-monotonic priorities and deadlines are assumed to be implicit ($D_i = T_i$). Here, the generalized deadline, $D'_i = T_i$.

(2) *ES-DMS*: Tasks execute as per deadline-monotonic priorities. This implies that the generalized deadline, $D'_i = D_i$.

(3) *ES-RHS+*: Tasks execute as per rate-monotonic priorities, and deadlines are implicit. However, tasks become eligible to execute based on the principle of *harmonization*: A task is eligible to execute only when the processor is *busy* or a *Harmonizing Period* boundary has been reached [12]. The use of harmonization enables every *idle* duration in the ES-RHS+ schedule to precede and be contiguous with the ES-task. Hence, all the processor's idle durations can be utilized to put it into deep sleep, thereby providing maximal energy savings [12]. Due to harmonization, each task can be delayed by at most $T_{sleep} - C_{sleep}$ [12]. This implies that the generalized deadline, $D'_i = T_i - (T_{sleep} - C_{sleep})$, and provides a tight schedulability test compared to the slightly looser one proposed in [12]. An example schedule for ES-RHS+ and ES-RMS using a taskset with 3 tasks is illustrated in Figure 1.

Multi-core processors also support a number of low-power states called C-states. In some processors, individual cores can transition to intermediary *idle* states. However, in many processors, cores cannot individually transition into deep sleep. Based on the ability to transition into deep sleep, two types of problems were defined in [12] for ES Schedulers:

1. *Synchronized-Sleep* or *SyncSleep Scheduling* where, all cores transition *synchronously* into deep sleep. Example processors include Intel Core^2 Duo [13] and AMD Opteron [15].
2. *Independent-Sleep* or *IndSleep Scheduling* where, each core can independently transition into deep sleep. Example processors with this flexibility include Samsung Exynos 5800 [2] and the 4th generation Intel Core processors [1].

In the SyncSleep context, only for the idle durations that overlap across all cores and exceed $C_{SleepMin}$ can the processor be put into deep sleep. Given the same T_{sleep}, ES-RMS can guarantee higher forced-sleep utilization U_{sleep} than ES-RHS+ [12]. This makes ES-RMS a better choice for SyncSleep [12]. For IndSleep, it was proved that using ES-RHS+ can yield an energy-optimal schedule *for all* feasible partitions [12].

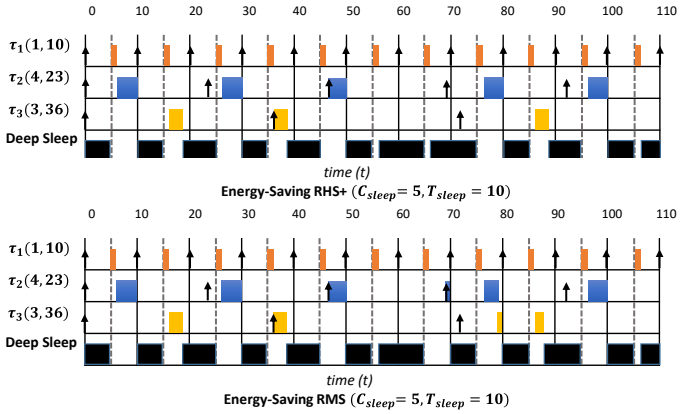


Figure 1 Energy-Saving Schedulers: ES-RHS+ & ES-RMS (C_{sleep} = 5, $C_{SleepMin}$ = 5, T_{sleep} = 10).

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 introduces the thermal model used in the paper. Section 4 introduces the *SysSleep* algorithm, and discusses utilizing ES Schedulers for reducing temperature in uni-core processors. Section 5 discusses utilizing ES Schedulers for reducing temperature in multi-core processors. Section 6 presents comparative evaluations, and Section 7 provides concluding remarks.

2 Related Work

Thermal Management can be done reactively at runtime [8, 17, 11, 36] or proactively at design time [20, 5, 16, 33, 10, 9, 4]. In the scope of reactive techniques, Fu. et al. [17] proposed a control-theoretic algorithm to meet the desired temperature requirement on a multi-core processor, subject to timing constraints. Yun et al. [36] used a machine-learning technique (SVM) to predict the temperature profile of a multi-processor system. Based on the predicted value, a dynamic temperature management scheme is used. In [8], Chandarli et al. proposed an optimal reactive scheduler for fixed-priority uniprocessor sleep scheduling along with an associated response-time based analysis framework. However, reactive schedulers require temperature sensors, which may not always be present in real platforms.

In the scope of proactive techniques, [10] describes a real-time scheduling algorithm for uniprocessors, based on a thermal model approximated by Fourier's Law. The algorithm derives a speed schedule by minimizing temperature under both timing and thermal constraints. In [9], an assignment and scheduling technique for an MPSoC was proposed, which utilizes a mixed-integer linear program solver to optimize the peak temperature. In [16], an optimal speed schedule is derived for a multi-core platform, based on a thermal model given at design time. In [4], Masud et al. proposed the use of a thermal-aware periodic resource to minimize peak temperature, in the context of uniprocessor Earliest Deadline First (EDF) scheduling. The processor slack is utilized to put the processor into a sleep state.

Most of the pieces of work stated [17, 11, 36, 16, 20, 5] have focused on the use of DVFS to optimize the processor temperature. However, the dominance of static power makes it necessary to investigate techniques which utilize sleep states. Additionally, many low-powered devices often lack DVFS, but support sleep states [28]. The work in [8] and [4] propose thermal-aware techniques which utilize processor sleep states. However, [4] assumes dynamic-priority EDF scheduling. On the other hand, [8] presents a reactive framework for

uniprocessor fixed-priority scheduling. To the best of our knowledge, no thermal analysis framework for proactive fixed-priority sleep scheduling exists in the literature.

Fixed-priority energy-saving schedulers, which periodically utilize the processor's deep-sleep state, were proposed in [28][12]. For these schedulers, the work in [12] proposed various techniques to design energy-efficient schedules in both the uni-core and multi-core processor contexts. In this paper, we analyze the thermal implications of ES Schedulers in light of their energy-saving properties. Based on a well-known thermal model, we derive practical insights and algorithms. Our proposed techniques focus on minimizing the maximum temperature, rather than optimizing to meet a set of thermal constraints.

3 Thermal Modeling of ES Schedulers

In this section, we introduce the thermal model used in the paper, and derive insights in the context of ES Schedulers. The temperature of a processor is dependent on the power consumption, and the variation in power consumption over time. Therefore, we can broadly define three factors responsible for a processor's thermal profile: (i) Heat generation by a core (due to power consumption). (ii) Heat dissipation to the environment (using heat sinks). (iii) Heat dissipation between adjacent cores (due to difference in power consumption patterns).

3.1 Power and Thermal Model

The power consumption of a CMOS circuit is modeled as a combination of two components:

1. *Dynamic Switching Power* is dependent on the processor operating frequency, and is consumed when the processor is *busy*. The dynamic power consumption, P_D, can be modeled as a convex function of the operating frequency s as [8]: $P_D = \kappa_0 s^\alpha$ where, α and κ are system constants which depend on the semiconductor technology used.
2. *Static Leakage Power* is due to leakage current, which depends on the semiconductor technology and the operating temperature. Static power is consumed even when the processor is *idle*, but can be nearly eliminated by putting the processor into *deep sleep*. Static power, P_S, can be conservatively modeled as a linear function of temperature [8]: $P_S = \kappa_1 \Theta + \kappa_2$ where, κ_1 and κ_2 are technology-dependent system constants, and Θ is the operating temperature.

Hence, the total power consumption P, as a function of time t, can be modeled as: $P(t) = P_D(t) + P_S(t)$. This model can be used to derive the thermal model for a uniprocessor. As OS schedulers control task execution at the granularity of a processor core, each core can be treated as a single unit producing heat and can be modeled as an RC circuit [8] [37]. When a core is *busy*, it generates heat. Using the RC thermal model, Fourier's Law [8] can be used to state the differential equation of the temperature, Θ^* with respect to time:

$$d\Theta^*(t)/dt = [P(t)/C] - [(\Theta^*(t) - \Theta_A)/RC] \tag{2}$$

where, Θ_A is the ambient temperature of the environment. By substituting P_D and P_S in Equation 2, we can rewrite Equation 2 as a classical linear differential equation [8]:

$$d\Theta(t)/dt = a - b\Theta(t) \tag{3}$$

where, $a = \kappa_0 s^\alpha / C$, $b = (1 - \kappa_1 R)/RC$ and the temperature has been offset from $\Theta^*(t) - [(\kappa_2 R + \Theta_A)/(1 - \kappa_1 R)]$ to $\Theta(t)$. Solving Equation 3 gives the temperature at time t as:

$$\Theta(t) = a/b + (\Theta(t_0) - a/b)e^{-b(t-t_0)}\,. \tag{4}$$

When the processor is in deep sleep, the power consumption can be assumed to be negligible. This is a valid assumption as the difference in power consumption between the busy and deep-sleep states is different by several orders of magnitude [28]. Hence, the processor can be deemed to be cooling when in the deep-sleep state. Using this assumption, one can set $a = 0$ in Equation 4 to obtain the model for cooling:

$$\Theta(t) = \Theta(t_0)e^{-b(t-t_0)}. \tag{5}$$

3.2 Thermal-Aware ES Scheduler Design

Consider a uni-core processor. For ES Schedulers, the processor is *guaranteed* to be in deep sleep atleast for a duration C_{sleep} every T_{sleep}. Hence, in the worst case, a core is *busy* for a duration of $T_{sleep} - C_{sleep}$ every T_{sleep}. Therefore, in the worst case, a processor core heats up from kT_{sleep} to $kT_{sleep} + C_{sleep}$ and cools down from $kT_{sleep} + C_{sleep}$ to $(k+1)T_{sleep}$, where k is a non-negative integer. As the heating function is monotonic in the period T_{sleep}, the temperature would be maximum at the end of the heating duration. We call this temperature Θ_{max}. Similarly, as the cooling function is monotonic in the period T_{sleep}, the temperature would be minimum at the end of the cooling duration. We call this temperature Θ_{min}. Applying the heating and cooling models from Equations 4 and 5 in the duration $[kT_{sleep}, (k+1)T_{sleep})$, we can write Θ_{max} and Θ_{min} as recurrent equations:

$$\Theta^k_{max} = a/b + (\Theta^{k-1}_{min} - a/b)e^{-b(Tsleep-Csleep)},\ \Theta^k_{min} = \Theta^k_{max}e^{-bC_{sleep}}. \tag{6}$$

At steady state, as $k \to \infty$, then $\Theta^k_{min} = \Theta^{k-1}_{min}$ and $\Theta^k_{max} = \Theta^{k-1}_{max}$. Hence, the steady state worst-case values of Θ_{max} and Θ_{min} are given by:

$$\Theta_{min} = (a/b) * [(e^{bT_{sleep}(1-U_{sleep})} - 1)/(e^{bT_{sleep}} - 1)], \Theta_{max} = \Theta_{min}e^{bU_{sleep}T_{sleep}} \tag{7}$$

where, $U_{sleep} = C_{sleep}/T_{sleep}$ denotes the *guaranteed* utilization of the ES-task. Based on the steady state temperatures, we can draw the following conclusions:

- Increasing U_{sleep}, keeping T_{sleep} constant, decreases the maximum temperature Θ_{max}.
- Decreasing T_{sleep}, keeping U_{sleep} constant, decreases the maximum temperature Θ_{max}.

Hence, minimizing T_{sleep}, while maximizing U_{sleep}, leads to a low maximum temperature. Thus, while it is advantageous to increase the total fraction of time the processor cools, i.e. $U_{sleep} \uparrow$ (also increases guaranteed energy savings), the cooling durations should be smaller but more frequent, i.e. $T_{sleep} \downarrow$. Note that these statements hold regardless of the system's thermal constants. Hence, using these principles, we can design techniques which can be used to minimize the temperature across a range of different systems.

In prior work [28][12], it was assumed that the period of the ES-task is a sub-harmonic of the highest-priority task. In the following section, we relax this constraint and provide techniques to design a thermally-effective ES schedule. Additionally, we show how choosing a proper T_{sleep} can maximize energy savings and improve schedulability.

4 SysSleep Algorithm

Consider a uni-core processor. To lower the worst-case maximum temperature for a taskset, we need to find an ES-task with a small period T_{sleep}, which also maximizes U_{sleep}. Maximizing U_{sleep} corresponds to finding the maximum *highest-priority* workload that can be added to a taskset without making it unschedulable. In [29], Saewong et al. proposed the SysClock algorithm which calculates the lowest processor frequency at which all tasks (with RM/DM

priority assignment) meet their deadlines. SysClock calculates the slack at all scheduling points in the critical zone [24] to determine the optimal operating frequency. We extend that algorithm in the context of ES Schedulers, and use it to compute the set of T_{sleep} values which maximize U_{sleep}. Our algorithm is called *SysSleep*, and its pseudo-code is presented in Algorithm 1. We illustrate the working of *SysSleep* by proving its optimality.

▶ **Theorem 1.** *For a taskset* Γ *using ES-RMS, SysSleep yields the maximum possible forced-sleep utilization* U_{sleep}^{max}.

Proof. Consider the critical zone theorem [24] where, in the worst case, the requests of all tasks arrive simultaneously. In order to be schedulable, a task τ_i must complete before its deadline D_i, i.e., its worst-cast response time $R_i \leq D_i$. If an ES-task is added to the system, all tasks will now complete at a later time, which should still be less than D_i for the task to remain schedulable. Since the workload changes at every scheduling point, *SysSleep* determines the maximum workload α_i^t, that can be added to the system, such that a task τ_i completes exactly at the end of each *idle* period t between R_i and D_i. This maximum workload corresponds to the slack utilization in the schedule up to time t. While calculating α_i^t, we consider a task's execution as well as all other higher-priority tasks. For a task, the maximum workload that can be added is chosen to be the *maximum* of these candidate values. We refer to this as the *maximum additional workload*, $\rho_i^{max} = max_t(\alpha_i^t)$ for a task τ_i.

For a taskset Γ, the maximum highest-priority workload that can be added also corresponds to the maximum possible forced sleep U_{sleep}^{max}, which is the *minimum* of the *maximum additional workload* of all the tasks, i.e., $U_{sleep}^{max} = min_{\tau_i \epsilon \Gamma}(\rho_i^{max})$. Hence, U_{sleep}^{max} corresponds to the task, τ_c with the lowest *maximum additional workload*, i.e. $min_{\tau_i \epsilon \Gamma}(\rho_i^{max})$. If the added workload exceeds U_{sleep}^{max}, then τ_c will miss its deadline and the taskset will become unschedulable. ◀

▶ **Example 2.** Consider a taskset Γ consisting of two tasks $\tau_1 = (1, 5)$ and $\tau_2 = (1, 7)$. For τ_1, the only end-of-idle period to consider is 5.

$$\alpha_1^5 = (t - C_1)/t = 0.8, \rho_1^{max} = max(\alpha_1^5) = 0.8$$

For τ_2, the end-of-idle periods to consider are 5 and 7.

$$\alpha_2^5 = [t - (C_1 + C_2)]/t = 0.6, \alpha_2^7 = [t - (2C_1 + C_2)]/t = 0.57, \rho_2^{max} = max(\alpha_2^5, \alpha_2^7) = 0.6$$

Hence, the maximum workload U_{sleep}^{max} that can be added is: $U_{sleep}^{max} = min(\rho_1^{max}, \rho_2^{max}) = 0.6$

We now need to find the set of T_{sleep} values which yield the maximum forced-sleep utilization U_{sleep}^{max}. For each task τ_i, let the end-of-idle period to which ρ_i^{max} corresponds be its *critical deadline*, $t_i^{critical}$. Using this notation, we can state the following lemma:

▶ **Lemma 3.** *If* T_{sleep} *is a sub-harmonic of* $t_i^{critical}$*, then the ES-task* τ_{sleep} *can utilize all the slack* ρ_i^{max} *till* $t_i^{critical}$*, such that* τ_i *completes at* $t_i^{critical}$.

Proof. If T_{sleep} is a sub-harmonic of $t_i^{critical}$, the effective utilization [34] of τ_{sleep} in the duration $[0, t_i^{critical}]$ is equal to its utilization U_{sleep}. The effective utilization of a task in a duration $[0, t]$ is the fraction of processor time used by a task in that duration. The actual utilization of a task cannot exceed its effective utilization in *any* duration. Hence, τ_{sleep} can optimally utilize all the slack ρ_i^{max} in the duration $[0, t_i^{critical}]$, such that its effective and actual utilizations are equal in the duration, i.e. $U_{sleep} = \rho_i^{max}$. ◀

The calculated U_{sleep}^{max} corresponds to the task with the minimum ρ_i^{max}. Let us call this the *critical task* τ_c, and let the end-of-idle period to which ρ_c^{max} corresponds be its *critical deadline*, $t_c^{critical}$. Applying Lemma 2 in the context of τ_c, we can state the following corollary:

Algorithm 1 SysSleep Algorithm

procedure SysSleep(Γ)
  for $\tau_i \in \Gamma$ **do**
    $(\rho_i^{max}, t_i^{critical})$ = CalculateMaxSlack(τ_i, Γ)
  $U_{sleep}^{max} = min(\rho_i^{max}, \tau_i \in \Gamma)$ ▷ Max Sleep Utilization
  $t^{critical} = t_{argmin(\rho_i^{max})}^{critical}$ ▷ Critical Deadline
  return $U_{sleep}^{max}, t^{critical}$
procedure CalculateMaxSlack(τ_i, Γ)
  /* S = slack, I = idle duration, BusyFlag is set if core *busy*, β = workload */
  $S = I = \beta = \Delta = 0, \mu = 1$, BusyFlag=TRUE
  $\omega = C_i, \omega' = 0$
  while $\omega < D_i$ **do**
    if BusyFlag == TRUE **then** ▷ Start of a busy period
      $\Delta = D_i - \omega$
      while $\omega < D_i$ AND $\Delta > 0$ **do**
        $\omega' = \sum_{j=0}^{i} \lceil C_j * (\lfloor \omega / T_j \rfloor + 1) \rceil + S$ ▷ Workload Calculation
        $\Delta = \omega' - \omega, \omega = \omega'$
      BusyFlag = FALSE
    else ▷ Start of an idle period
      $I = min_{\forall j<i}[(T_j * \lceil \omega / T_j \rceil - \omega), D_i - \omega]$ ▷ Slack Computation
      $S = S + I, \omega = \omega + I, t = \omega, \beta = \omega - S$
      if $\beta / t < \mu$ **then**
        $\mu = \beta / t, t^{critical} = t, \rho = 1 - \mu$ ▷ Update the maximum additional workload
      BusyFlag = TRUE
  return $\rho, t_{critical}$

▶ **Corollary 4.** *If T_{sleep} is a sub-harmonic of $t_c^{critical}$, then the ES-task, τ_{sleep}, optimally utilizes all the slack, such that the critical task τ_c completes at $t_c^{critical}$.*

Unfortunately, choosing any sub-harmonic of $t_c^{critical}$ may not guarantee schedulability for other tasks in Γ. If the effective utilization of τ_{sleep} exceeds ρ_k^{max} in the duration $[0, t_k^{critical}]$, for another task $\tau_k \in \Gamma$, then τ_k will become unschedulable. Hence, we need to choose T_{sleep} such that the effective utilization of τ_{sleep} is always less than $\rho_i^{max} \forall \tau_i \in \Gamma$.

▶ **Theorem 5.** *Choosing T_{sleep} as a common divisor of all $t_i^{critical} \forall \tau_i \in \Gamma$ such that $T_{sleep} \leq T_1$, always yields a schedule with the optimal forced-sleep utilization U_{sleep}^{max}.*

Proof. From Lemma 2, choosing T_{sleep} as a *common divisor* of all $t_i^{critical}$ ensures that the effective utilization U_{sleep}^{eff} of the energy-saver task τ_{sleep} is equal to its maximum utilization U_{sleep}^{max} in all the critical durations $[0, t_i^{critical}] \forall \tau_i \in \Gamma$. The optimal forced-sleep utilization is given by, $U_{sleep}^{max} = min_{\tau_i \epsilon \Gamma}(\rho_i^{max})$. Hence, $U_{sleep}^{eff} = U_{sleep}^{max} \leq \rho_i^{max} \forall \tau_i \in \Gamma$. ◀

It is very important to note that, in practice, the choice of T_{sleep} is constrained by the system constraint $C_{SleepMin}$ on the lower side and the period of the highest-priority task T_1 (τ_{sleep} must execute at the highest priority) on the higher side. Given this system constraint, we can state the following theorem:

▶ **Theorem 6.** *Consider a taskset Γ, schedulable by an ES scheduler, running on a system with the minimum deep sleep round-trip duration $C_{SleepMin}$. Then for Γ, the lower bound on*

Algorithm 2 ThermoSleep Heuristic

```
1:  procedure ThermoSleep(Γ, C_SleepMin, num_core)
2:      while True do
3:          U^max_sleep, t^critical_j = SysSleep(Γ)                ▷ Invoke SysSleep
4:          if t^critical_j ≤ D'_j then break   ▷ If critical deadline is within generalized deadline
5:      if C_SleepMin/U^max_sleep < T_1 then                       ▷ Check if feasible solution exists
6:          μ = ⌊U^max_sleep * t^critical/C_SleepMin⌋, ν = ⌈t^critical/T_1⌉   ▷ Range of divisors
7:          if μ < ν then
8:              μ = ν
9:          Θ_best = ∞
10:         for k = μ to ν do
11:             T^k_sleep = t^critical/k
12:             Θ^best_k = CalcTemperature(U^max_sleep, T^k_sleep)   ▷ Lowest temperature for T^k_sleep
13:             if Θ_best < Θ^best_k then
14:                 break
15:             else
16:                 U^best_sleep = FindSleepUtil(Γ, T^k_sleep, num_core)  ▷ Find best U_sleep for T^k_sleep
17:                 Θ_max = CalcTemperature(U^best_sleep, T^k_sleep)
18:                 if Θ_max < Θ_best then
19:                     T_sleep = T^k_sleep, U_sleep = U^best_sleep, Θ_best = Θ_max   ▷ Best Solution found
20:     else
21:         return NotSchedulable                                  ▷ No feasible solution exists
22:     return T_sleep, U_sleep
23: procedure FindSleepUtil(Γ, T_sleep, m)
24:     /* m = num_cores, Γ_i = tasks allocated to core i */
25:     for i = 1 to m do
26:         U^i_sleep = FindBestSleep(Γ_i, T_sleep)                ▷ Invoke FindBestSleep
27:     return min_i(U^i_sleep)
```

the optimal worst-case maximum temperature Θ^{best}_{max} *achievable by ES schedulers is:*

$$\Theta^{best}_{max} = (a/b)[(e^{bT^{min}_{sleep}(1-U^{max}_{sleep})} - 1)/(e^{bT^{min}_{sleep}} - 1)] * e^{bU^{max}_{sleep}T^{min}_{sleep}}. \quad (8)$$

Proof. For a taskset Γ, *SysSleep* returns the maximum possible forced-sleep utilization U^{max}_{sleep}. Hence, given the system constraint $C_{SleepMin}$, the smallest feasible ES-task period is $T^{min}_{sleep} = C_{SleepMin}/U^{max}_{sleep}$. From Equation 7, the worst-case maximum temperature Θ_{max} is minimized by simultaneously minimizing T_{sleep} and maximizing U_{sleep}. Hence, substituting the smallest feasible ES-task period, T^{min}_{sleep}, and the largest schedulable forced-sleep utilization U^{max}_{sleep} in Equation 7 yields the lower bound on the *optimal* worst-case maximum temperature Θ^{best}_{max} achievable by ES Schedulers, corresponding to the taskset Γ. ◀

From a thermal perspective, for a fixed U_{sleep}, a smaller T_{sleep} yields a lower worst-case maximum temperature. Hence, a possible thermally-effective solution with optimal forced-sleep utilization can be the smallest *common divisor* of all $t^{critical}_i \forall \tau_i \in \Gamma$ that lies in the range $[C_{SleepMin}/U^{max}_{Sleep}, T_1]$. If $C_{SleepMin}/U^{max}_{sleep} > T_1$, then no feasible solution exists. Note that, choosing T_{sleep} as *any* common divisor of $t^{critical}_i \forall \tau_i \in \Gamma$ that lies in the range

$[C_{SleepMin}/U^{max}_{Sleep}, T_1]$ would yield solutions with equivalent energy consumption. However, the dependence of temperature on T_{sleep} would yield different thermal profiles.

Unfortunately, in many cases, no common divisor of the critical deadlines may lie in $[C_{SleepMin}/U^{max}_{Sleep}, T_1]$. Hence, we present the *ThermoSleep* heuristic. *ThermoSleep* invokes *SysSleep* to compute U^{max}_{sleep}, along with the critical deadline $t_c^{critical}$ corresponding to U^{max}_{sleep}. *ThermoSleep* uses these values to return the smallest possible sub-harmonic of the critical deadline $t_c^{critical}$ corresponding to the critical task τ_c, that yields a thermal and energy-efficient schedule. The pseudo-code for *ThermoSleep* is presented in Algorithm 2.

Given an ES-task period $T_{sleep} \leq T_1$, *ThermoSleep* uses the *FindBestSleep* (FBS) algorithm to compute the optimal C_{sleep} for a core, which allows a taskset Γ to be schedulable. The pseudo-code for FBS is provided in Algorithm 3. We now prove the optimality of FBS.

▶ **Theorem 7.** *For a taskset Γ schedulable by ES-RMS, with an ES-task τ_{sleep} having a period T_{sleep}, FindBestSleep returns the optimal forced-sleep utilization U'_{sleep}.*

Proof. Consider the critical zone theorem [24] where, in the worst case, the requests of all tasks arrive simultaneously. In order to be schedulable, a task τ_i must complete before its deadline D_i. Given that a new job of τ_{sleep} is dispatched every T_{sleep}, for each task $\tau_i \in \Gamma$, FBS determines the maximum workload that can be added to the taskset, such that τ_i completes by t where, t is an integer multiple of T_{sleep}, i.e., $(k * T_{sleep} \leq D_i)$ or D_i. This gives the effective slack, α_i^t, that τ_{sleep} can utilize, if τ_i and all higher-priority tasks complete by t. For a task τ_i, the maximum highest-priority workload with period T_{sleep} that can be added is the *maximum* of these calculated values $\rho_i^{max} = max_t(\alpha_i^t)$. For a taskset Γ with an ES-task period T_{sleep}, this workload corresponds to the maximum possible forced sleep, U'_{sleep}, which is the minimum of the ρ_i^{max} of all the tasks. Hence, $U'_{sleep} = min_{\tau_i \epsilon \Gamma}(\rho_i^{max})$, which corresponds to the task, $\tau_c \mid c = argmin_{\tau_i \epsilon \Gamma}(\rho_i^{max})$. If the added workload exceeds U'_{sleep}, then τ_c will miss its deadline and Γ will become unschedulable. ◀

For ES-RHS+, the total deep-sleep utilization $U_{SleepTotal}$ is given by $1 - \sum_{\tau_i \epsilon \Gamma}(C_i/T_i)$. Hence, for a schedulable taskset, ES-RHS+ guarantees a sleep schedule with *optimal* energy savings. However, this deep-sleep utilization is not uniformly distributed over each period. To reduce the worst-case maximum temperature, the ES-task utilization U_{sleep} must be increased, and its period T_{sleep} must be decreased. In Section 1.2 the schedulability test for ES-RHS+ was discussed, and for each task the *generalized deadline* $D'_i = T_i - (T_{sleep} - C_{sleep})$, is a function of both C_{sleep} and T_{sleep}. Hence, *ThermoSleep* invokes *SysSleep* multiple times to compute U^{max}_{sleep} until the *critical deadline* of the *critical task* lies within its *generalized deadline*. To calculate the generalized deadline, we choose T_{sleep} to be the smallest sub-harmonic of the critical deadline in the feasible range $[C_{SleepMin}/U^{max}_{sleep}, T_1]$, and $C_{sleep} = U^{max}_{sleep} * T_{sleep}$.

Given a forced-sleep period, T_{sleep}, ES-RMS can provide a higher forced-sleep utilization, U_{sleep}, than ES-RHS+ [12]. Hence, for a taskset Γ, in most cases, ES-RMS will yield a lower worst-case maximum temperature compared to ES-RHS+. In practice, ES-RHS+ can yield lower temperatures, as it utilizes *all* idle durations to put the processor into deep sleep.

5 Thermal-Aware Multi-Core ES Scheduling

Consider a task set Γ consisting of n periodic real-time tasks $\tau_1, \tau_2, ..., \tau_n$ that need to be scheduled on a homogeneous multi-core processor with m cores, $M_1, M_2, ..., M_m$. Each core M_k has an ES-task, $\tau_{sleep,k}$, which has a forced-sleep duration of $C_{sleep,k} \geq C_{SleepMin}$ every $T_{sleep,k}$. As mentioned in Section 1.2, two types of multi-core ES scheduling problems were

Algorithm 3 FindBestSleep Algorithm

```
1:  procedure FindBestSleep(Γ, C_SleepMin, T_sleep)
2:      for τ_i ∈ Γ do
3:          (ρ_i^max, t_i^critical) = CalculateSlack(τ_i, Γ, T_sleep)
4:      U_sleep = min(ρ_i^max, τ_i ∈ Γ)                    ▷ Max Sleep Utilization
5:      if U_sleep * T_sleep ≥ C_SleepMin then              ▷ Check if feasible solution exists
6:          return U_sleep^max * T_sleep
7:      else
8:          return NotSchedulable
9:  procedure CalculateSlack(τ_i, Γ, T_s)
10:     /* S = slack, I = idle duration, BusyFlag is set if core busy, β = workload */
11:     S = I = β = Δ = 0, μ = 1, BusyFlag=TRUE, ω = C_i, ω' = 0
12:     while ω < D_i do
13:         if BusyFlag == TRUE then                        ▷ Start of a busy period
14:             Δ = D_i − ω
15:             while ω < D_i AND Δ > 0 do
16:                 ω' = Σ_{j=0}^{i} [C_j * (⌊ω/T_j⌋ + 1)] + S   ▷ Workload Calculation
17:                 Δ = ω' − ω, ω = ω'
18:             BusyFlag = FALSE
19:         else                                            ▷ Start of an idle period
20:             Ω = {j ∈ Z^+ | (j − 1) * T_s ≤ D_i < j * T_s}
21:             I = min_{∀j∈Ω}[(j * T_s * ⌈ω/j * T_s⌉ − ω)]   ▷ Slack computation
22:             S = S + I, ω = ω + I, t = ω, β = ω − S
23:             if β/t < μ then
24:                 μ = β/t, ρ = 1 − μ                      ▷ Update the maximum additional workload
25:             BusyFlag = TRUE
26:     return ρ
```

defined in [12]. In this section, we analyze the thermal implications of *SyncSleep* and *Indsleep* scheduling, and propose techniques to derive thermally-effective partitioned schedules.

In multi-core processors, heat also dissipates between adjcacent cores, and the rate of dissipation depends on the temperature differences between them. Hence, each core can be modeled using the RC model with the addition of thermal resistances between adjacent cores [16]. Let the instantaneous temperature on each core be Θ_j, for $j = 1, 2, ..., m$. Using Fourier's Law, the differential equation for each core's temperature can be given by:

$$\frac{d\Theta_j(t)}{dt} = \frac{P_j(t)}{C} - \frac{\Theta_j(t) - \Theta_A}{RC} - \sum_{k=1}^{m} \frac{\Theta_j(t) - \Theta_k(t)}{R_{jk}C} \tag{9}$$

where, P_j is the instantaneous power dissipated by the core, and R_{jk} is the thermal resistance between the cores j and k. For *non-adjacent* cores one can reasonably assume there is no heat dissipation between them and hence, $R_{jk} = \infty$ [16].

5.1 SyncSleep Scheduling

For *SyncSleep* scheduling, the forced-sleep task must be synchronized across all cores [12]. As the sleep transition is synchronous, for all cores $T_{sleep,k} = T_{sleep}$, and the initial ES-task phase can be taken as $\phi_{sleep,k} = 0$ [12]. Additionally, the minimum amount of time for which the

Algorithm 4 SyncSleep Partitioning Heuristic

1: **procedure** PARTITIONTASKSET($\Gamma, C_{SleepMin}, m$)
2: /* m = number of cores, Γ_i = tasks allocated to core i */
3: $T_s = 1$ ▷ Set forced-sleep period to 1
4: $\Gamma_i \forall i \in 1$ to m = MaxSyncSleep($\Gamma, C_{SleepMin}, T_s, m$) ▷ from [12]
5: U_s, T_s = ThermoSleep($\Gamma, C_{SleepMin}, m$) ▷ Invoke ThermoSleep
6: **return** U_s, T_s ▷ SyncSleep task parameters

system can be in deep sleep is dictated by the core which has the least forced-sleep duration [12]. Hence, if the system synchronous sleep $C_{SyncSleep} = \min_{k=1}^{m}(C_{sleep,k}) \geq C_{SleepMin}$, then the minimum guaranteed deep-sleep utilization is given by $\min_{k=1}^{m}(C_{sleep,k})/T_{sleep}$.

Based on the synchronous-sleep constraint, in the worst case, we can assume that all the cores are in deep sleep for the durations $[kT_{sleep}, kT_{sleep} + C_{SyncSleep})$, and busy from $[kT_{sleep} + C_{SyncSleep}, (k+1)T_{sleep})$. Hence, all cores will have the same worst-case execution profile (as illustrated in Figure 2(a)), and we can assume that in the *worst case*, at any time instant, all cores share the *same* temperature. Thus, the worst-case inter-core temperature difference is always zero, and the model reduces to the uniprocessor thermal model. Hence, from a worst-case perspective, we can consider the entire system as one thermal unit. Applying these assumptions in Equation 9, the worst-case SyncSleep temperature model is given by:

$$d\Theta_j(t)/dt = P_j(t)/C - (\Theta_j(t) - \Theta_A)/RC\,. \tag{10}$$

Figure 2(b) presents an example using SyncSleep ES-RMS for a quad-core system with cores $M_i, i = \{1,2,3,4\}$. The taskset $\Gamma = \{\tau_1(6,10), \tau_2(7,10), \tau_3(5,10), \tau_4(4,10)\}$ is used, such that, during partitioning, each core receives one task (τ_i is assigned to M_i). Due to the synchronous nature of forced sleep, all cores have similar temperature profiles, making the heat dissipation between cores negligible. Hence, like the uniprocessor case, the problem reduces to finding a forced-sleep task τ_{sleep} which minimizes T_{sleep} while maximizing U_{sleep}. However, given that we have multiple cores, partitioning the tasks among them also plays a major role in determining the thermally-effective τ_{sleep}. The temperature minimization problem can be stated as the following task-partitioning problem: "Find a partition that has a synchronized ES-task which minimizes the worst-case maximum temperature, such that the workload allocated to each core can be scheduled feasibly by an ES Scheduler."

The stated partitioning problem is a more constrained form of the feasibility problem in multi-core processor scheduling, which is known to be NP-hard in the strong sense [18][25]. Hence, the thermal-aware *SyncSleep* scheduling problem is also NP-hard. Consider the trivial case where all tasks have the same periods, with different computation times. In this case, choosing the optimal T_{sleep} is trivial (from Theorem 4, it is a sub-harmonic of the task period). Given T_{sleep}, the temperature across all cores will be minimized if all cores have the same load. Hence, the problem reduces to calculating the optimal balanced partition for independent tasks with known computation times, which is known to be equivalent to the Partition problem [21] which is NP-Complete [21].

We now present a two-stage heuristic for the partitioning problem:

Partitioning for Thermal Performance: In the first stage, we choose the best possible hypothetical $T_{sleep} = 1$ to find the best synchronous forced sleep that a partitioning heuristic can achieve. Theorem 4 states that, on a single core, the optimal U_{sleep} is achieved when T_{sleep} is a common divisor of the critical deadline. Since 1 is a divisor of all integers, choosing

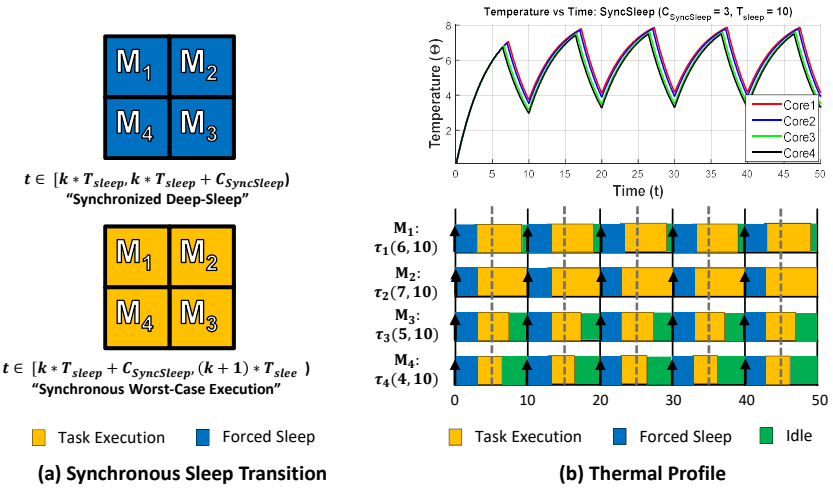


Figure 2 *SyncSleep* Scheduling for a quad-core system with cores M_i.

T_{sleep} = 1 enables a heuristic to achieve its best possible forced-sleep utilization. If a taskset cannot be scheduled when T_{sleep} = 1, we consider it unschedulable. Setting T_{sleep} = 1 and maximizing the forced-sleep utilization is similar to the energy minimization problem for SyncSleep Scheduling [12]. To realize energy savings and minimize temperature in multi-core systems, load balancing is often used [12]. Worst-Fit Decreasing (also referred to as WFD or List Scheduling when the number of cores is fixed *a priori*) is commonly used to obtain a load-balanced partition. WFD allocates tasks to the core with the least utilization, one by one in non-increasing order of their utilization. For ES Schedulers, the period ratios also play an important role in dictating the forced-sleep utilization, something that WFD does not take into account. In [12], the *MaxSyncSleep* (MSS) partitioning heuristic was proposed. Instead of using utilization to allocate tasks to cores, MSS measures the impact of a task's allocation on the synchronous forced-sleep duration.

Choosing the SyncSleep Period: In the second stage, we find a thermally-effective T_{sleep}. For an m-core system, let the best possible synchronous forced-sleep utilization (setting T_{sleep} = 1) obtained by a partitioning heuristic A be $U^{max}_{SyncSleep}$, which corresponds to the core k with the minimum forced-sleep utilization. The feasible range for T_{sleep} can now be given by $[C_{SleepMin}/U^{max}_{SyncSleep}, T_1]$. To find a good value for T_{sleep}, we run *ThermoSleep* on the partition. The proposed partitioning technique is described in Algorithm 4.

5.2 IndSleep Scheduling

Some processors allow each core to individually transition into deep sleep, enabling better energy savings. Hence, each core M_k has a forced-sleep task, $\tau_{sleep,k}$, which has a forced-sleep duration of $C_{sleep,k} \geq C_{SleepMin}$ every $T_{sleep,k}$, with a phasing $\phi_{sleep,k}$. Note that, compared to *SyncSleep* scheduling, each core's forced-sleep task can have a different $C_{sleep,k}$, as well as a different phasing $\phi_{sleep,k}$. Hence, we need to consider heat dissipation between cores. Thus, the *IndSleep* thermal model is given by Equation 9, and takes into account both heat dissipation to the environment, as well as between cores.

For *IndSleep* scheduling, the thermal-aware scheduling problem can be defined as follows: "Find a partition and forced-sleep task parameters (including phasing) on each core, that

minimizes the maximum temperature of the system, under the constraint that the workload allocated to each core can be scheduled by an ES Scheduler."

In [12], it was proved that using ES-RHS+ can yield an energy-optimal schedule *for all* feasible partitions. A partition is feasible if the tasks allocated to each core are schedulable. However, unlike the energy-minimization problem, all the feasible partitions are *not* optimal from a thermal perspective. This is due to the dependence of temperature on the ES-task period, as well as the execution pattern between cores, i.e. relative ES-task phasing.

The heat flow between two objects is primarily dependent on their thermal properties as well as the temperature difference between them. At any instant, the temperature difference between two adjacent cores will always be less than the temperature difference between a core and the environment. This is based on the practical assumption that the environmental temperature is *always* lower than that of any core. Thus, we can safely assume that heat dissipation to the environment is the dominant factor for cooling. Hence, from an optimization standpoint, we first optimize the schedule on each core to reduce its own temperature, and then optimize the schedule between cores to ensure maximal heat dissipation between them. Based on this practical assumption, we propose a two-stage solution:

Partitioning for Thermal Performance: The objective of partitioning is to ensure that the worst-case maximum temperature of the system is minimized. If there were no heat dissipation between cores, then the worst-case maximum temperature Θ^k_{max} on a core k is a function of $T_{sleep,k}$ and $U_{sleep,k}$. A balanced partition helps ensure that all cores have similar Θ^k_{max}. In an unbalanced partition, a core with a significantly lower $U_{sleep,k}$ would yield a higher temperature, thus raising the maximum temperature of the system. This is similar to the SyncSleep problem, and hence is also NP-Hard. Hence, like SyncSleep, we initially set $T_{sleep,k} = 1$ on each core, and use *MaxSyncSleep* [12] (or WFD) to create a balanced partition. Applying *ThermoSleep* to all the cores together gives a single T_{sleep} that is suitable for all the cores. We refer to this as *uniform sleep*. However, since each core can independently transition into deep sleep, each core's ES-task can have a different period, that we refer to as *non-uniform sleep*. These non-uniform sleep periods $T_{sleep,k}$ can be calculated by applying *ThermoSleep* to each core *individually*. *FindBestSleep* is then used to obtain each $C_{sleep,k}$ using the corresponding $T_{sleep,k}$. While *uniform sleep* ensures that all cores have a similar temperature profile, *non-uniform sleep* can allow each core to attain a lower temperature.

Forced-Sleep Phasing: The phasing between ES-tasks plays an important role in the heat dissipation between cores. In the worst case, we can assume that each core $M_j, j = 1$ to m is in *deep sleep* for the durations $[\phi_{sleep,j} + kT_{sleep,j}, \phi_{sleep,j} + kT_{sleep,j} + C_{sleep,j})$, and *busy* from $[\phi_{sleep,j} + kT_{sleep,j} + C_{sleep,j}, \phi_{sleep,j} + (k+1)T_{sleep,j})$. To ensure maximal heat dissipation between adjacent cores, the temperature difference between them needs to be maximal. For two adjacent cores i and j, the largest temperature difference between them occurs when core i is at the start of its forced-sleep period and core j is at the end of its forced-sleep period. Hence, if $\tau_{sleep,i}$ starts exactly after $\tau_{sleep,j}$ ends, then the instantaneous temperature difference between the cores can be maximized. This leads to an execution pattern where core i is busy while core j is in deep sleep and vice versa.

Figure 3(b) presents an example using IndSleep ES-RMS with uniform periods for a quad-core system with cores $M_i, i = \{1, 2, 3, 4\}$. The taskset $\Gamma = \{\tau_1(6, 10), \tau_2(7, 10), \tau_3(5, 10), \tau_4(4, 10)\}$ is used, such that, during partitioning, each core receives one task (τ_i is assigned to M_i). Note that, each core has its own distinct thermal profile. Additionally, phasing the ES-task on each core, to minimize execution overlap can yield thermal benefits. For the

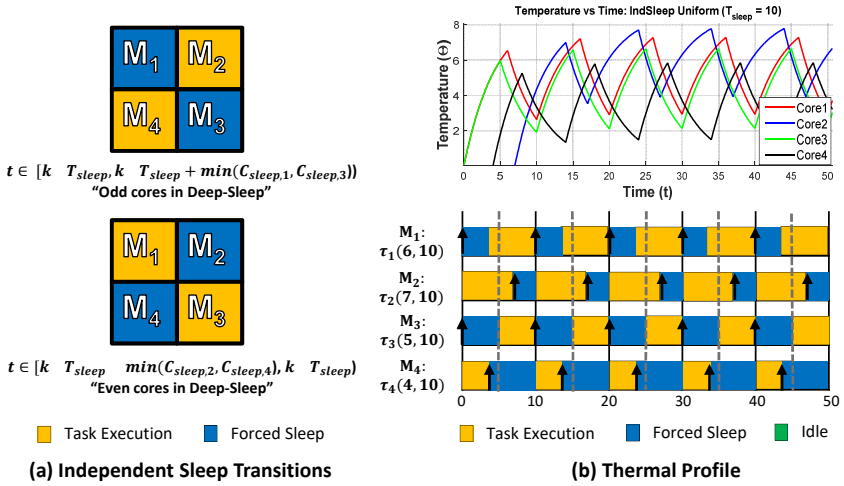


Figure 3 *IndSleep* Scheduling with uniform sleep periods for a quad-core system with cores M_i.

IndSleep example, the *odd-even* execution pattern illustrated in Figure 3(a) is noteworthy, where execution overlap is minimized by ensuring that odd-numbered cores are *busy* (i.e. execute tasks), while even-numbered cores are in deep sleep, and vice-versa. From the thermal profile, observe that this phasing causes the temperature difference between adjacent cores to be maximized, thus yielding better heat dissipation between adjacent cores.

As a simplification, we formulate the phasing problem as one of: "minimizing the execution (or forced-sleep) overlap between adjacent cores". By considering busy durations as *hot* and forced-sleep durations as *cool*, the execution overlap metric captures the durations where *hot* regions overlap, hence acting as a proxy for temperature difference. In most processor designs, cores are rectilinear, and adjacent cores are of the same size. Hence, to compute a thermally-effective phasing, the overlap between every pair of adjacent cores needs to be minimized. This execution overlap (also referred to as *overlap*) needs to be calculated over the *relative hyperperiod*, T_R, of all the cores. We define the *relative hyperperiod* as the least common multiple of all the cores' forced-sleep periods. In the simplest case, consider a dual-core system, with two adjacent cores. Let the cores be M_1 and M_2, and their forced-sleep tasks be $\tau_{sleep,i} = (C_{sleep,i}, T_{sleep,i})$ with phasing $\phi_{sleep,i}$ where, $i = 1, 2$. Assume that all the terms are integers, which is reasonable as we can convert timescales to arbitrarily small units (like nanoseconds). We have four possible cases:

1. $T_{sleep,1} = T_{sleep,2}$, i.e. uniform sleep. The phasing with the minimum overlap is computed over $T_R = T_{sleep,1} = T_{sleep,2}$. The minimum overlap possible is $T_R - C_{sleep,1} - C_{sleep,2}$. Then, $\phi_{sleep,1} = 0$, $\phi_{sleep,2} = C_{sleep,1}$, is one of the phasings which *guarantees* minimum overlap.
2. $T_{sleep,1}$ and $T_{sleep,2}$ are *relatively prime*, i.e. non-uniform sleep whose greatest common divisor is 1. The minimum overlap needs to be computed over $T_R = T_{sleep,1} * T_{sleep,2}$. In this case, any relative *integer phasing* of $\tau_{sleep,1}$ and $\tau_{sleep,2}$ guarantees the same overlap, which is the minimum overlap. This stems from the fact that all possible relative integer phasings between two periods are encountered, before the relative phasing is equal to that at the start.
3. $T_{sleep,1}$ and $T_{sleep,2}$ are *harmonic*, i.e. non-uniform sleep where one is a multiple of the other. Let $T_{sleep,2} = a * T_{sleep,1}, a \in Z^+$. Hence, $T_R = T_{sleep,2}$, and only one iteration of $\tau_{sleep,2}$ occurs in T_R. Then $\phi_{sleep,1} = 0$, $\phi_{sleep,2} = C_{sleep,1}$ can *guarantee* the minimum overlap.

4. $T_{sleep,1}$ and $T_{sleep,2}$ are not *relatively prime* and not *harmonic*, i.e. non-uniform sleep which share a common divisor, but one is non-divisible by the other. Here, $T_R < T_{sleep,1} * T_{sleep,2}$. In this case, no property can be stated on the relative phasing which guarantees minimum overlap.

Based on the above properties, we see that while simple approaches work for phasing uniform sleep, using non-uniform sleep requires more complex optimization techniques.

However, using *uniform sleep* does not always guarantee lower execution overlap than using *non-uniform sleep*. This can be seen from the following 3 cases:

Case 1: *Uniform Sleep* performs better than *Non-Uniform Sleep*. Consider a taskset with two tasks, $\tau_1 = (6, 9)$ and $\tau_2 = (10, 15)$. τ_1 is assigned to core M_1, and τ_2 to core M_2. In the *uniform sleep* case the best ES-task periods in terms of sleep utilization are $\tau_{sleep,1} = (3, 9)$ and $\tau_{sleep,2} = (2.5, 9)$. Using the best possible phasing, achieves a guaranteed minimum execution overlap of 3.5 every 9 (38.89%). In the non-uniform case, the best ES-task periods are $\tau_{sleep,1} = (3, 9)$ and $\tau_{sleep,2} = (5, 15)$. By searching the entire search space of unique relative integer phasings, the minimum execution overlap achievable is 20 every 45 (44.44%). Hence, in this case, using *uniform sleep* provides lower execution overlap.

Case 2: *Uniform Sleep* performs equal to *Non-Uniform Sleep*. Consider a taskset with two tasks, $\tau_1 = (6, 9)$ and $\tau_2 = (9, 12)$. τ_1 is assigned to core M_1, and τ_2 to core M_2. In the *uniform sleep* case, the best ES-task periods in terms of sleep utilization are $\tau_{sleep,1} = (3, 9)$ and $\tau_{sleep,2} = (1.5, 9)$. By using the best phasing, we can achieve a guaranteed minimum execution overlap of 4.5 every 9 (50%). In the non-uniform case, the best ES-task periods are $\tau_{sleep,1} = (3, 9)$ and $\tau_{sleep,2} = (3, 12)$. By searching the entire search space of unique relative integer phasings, the minimum execution overlap achievable is 18 every 36 (50%). Hence, both provide a solution with the same execution overlap.

Case 3: *Uniform Sleep* performs worse than *Non-Uniform Sleep*. Consider a taskset with two tasks, $\tau_1 = (6, 9)$ and $\tau_2 = (9, 11)$. τ_1 is assigned to core M_1, and τ_2 to core M_2. In the *uniform sleep* case, the best ES-task periods are $\tau_{sleep,1} = (3, 9)$ and $\tau_{sleep,2} = (1, 9)$. Using the best phasing, achieves a guaranteed minimum execution overlap of 5 every 9 (55.55%). In the non-uniform case, the best ES-task periods are $\tau_{sleep,1} = (3, 9)$ and $\tau_{sleep,2} = (2, 11)$. By searching the entire search space of unique relative phasings, the minimum execution overlap achievable is 54 every 99 (54.54%). Hence, in this case using *non-uniform sleep* provides lower execution overlap.

Since there is no exact solution for choosing ES-task periods for minimizing execution overlap, we examine the properties of using *uniform sleep* versus *non-uniform sleep*:

Best Phasing: While uniform sleep can be phased easily and optimally (using the odd-even execution pattern from Figure 3(a)) for a rectilinear multi-core processor, no such simple technique can be used for non-uniform sleep.

Temperature Profile: Uniform sleep will ensure that all cores have similar temperatures. However, using non-uniform sleep allows each individual core to choose the best T_{sleep}, to further reduce its temperature, based on the tasks allocated to it.

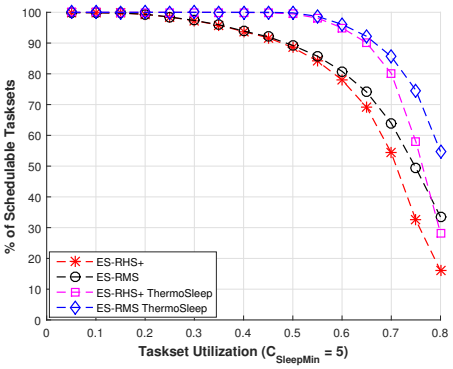


Figure 4 % of task sets schedulable w.r.t. taskset utilization, for uniprocessors

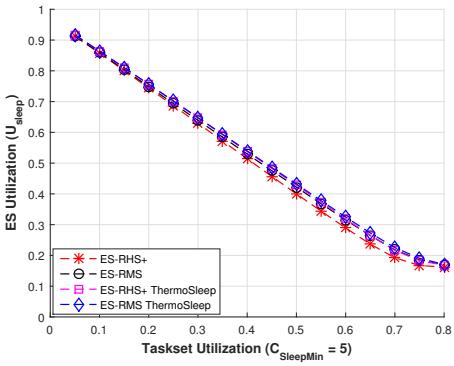


Figure 5 Utilization of the ES-task w.r.t. taskset utilization, for uniprocessors

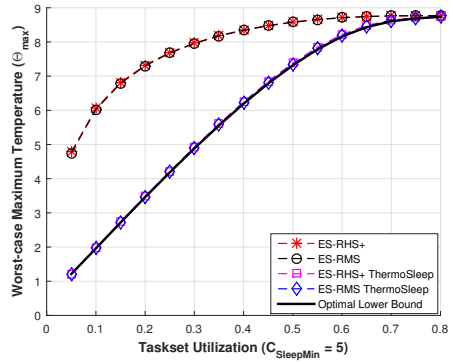


Figure 6 Worst-case maximum temperature w.r.t. taskset utilization, for uniprocessors

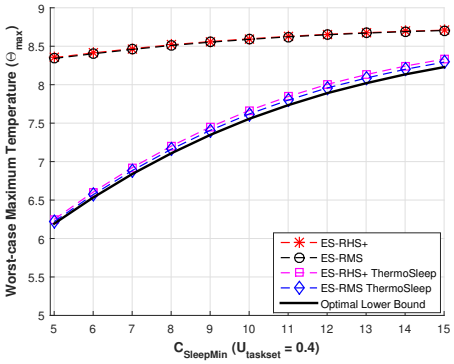


Figure 7 Worst-case maximum temperature w.r.t. $C_{SleepMin}$, for uniprocessors

6 Comparative Evaluation

We now evaluate our proposed techniques on the basis of schedulability and worst-case maximum temperature Θ_{max} with an offset. Results are obtained using both static worst-case analysis as well as dynamic simulations using Hotspot [37]. Static analysis experiments were performed on 100,000 tasksets generated randomly using UUniFast-Discard [14] for each data-point. In a taskset, each task is randomly assigned a period between 15 and 400 time units, and the number of tasks varies from 1 to 20. $C_{SleepMin}$ is set to 5 time units. The system thermal parameters were set to $a = 2$ and $b = 0.228$ [8]. To the best of our knowledge, no other *proactive* techniques exist for designing thermal-aware fixed-priority sleep schedules. Hence, we compare against the purely energy-efficient design methodology proposed in [12].

6.1 Static Worst-Case Analysis

Uniprocessor Comparisons: We compare ES-RMS and ES-RHS+ with and without using *ThermoSleep* on the basis of schedulability, and the worst-case maximum temperature, Θ_{max}. Figure 4 plots schedulability versus taskset utilization. In terms of schedulability: ES-RMS performs better than ES-RHS+. Observe that, using *ThermoSleep*, ES-RMS can schedule up to 62.5% more task sets than before. Figure 5 plots the ES-task utilization versus taskset

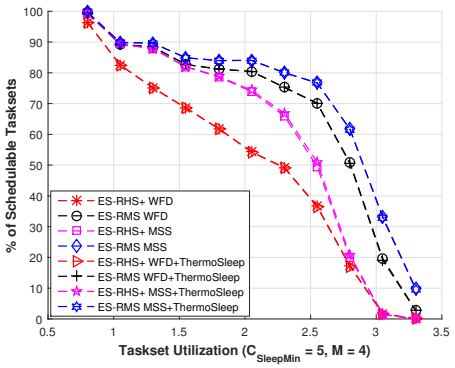


Figure 8 % of task sets schedulable w.r.t. taskset utilization, for multi-core *Sync-Sleep* scheduling ($m = 4$)

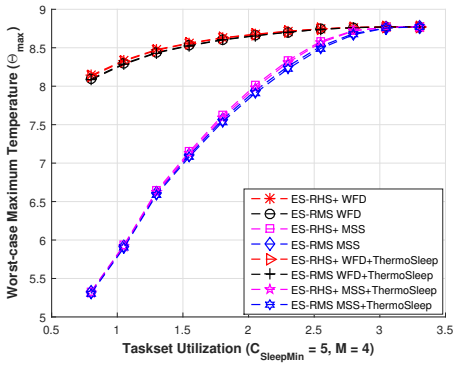


Figure 9 Worst-case maximum temperature w.r.t. taskset utilization, for multi-core *Sync-Sleep* scheduling ($m = 4$)

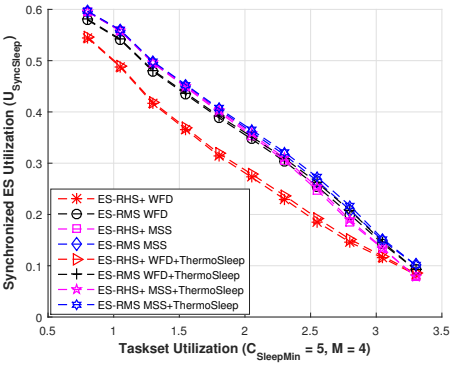


Figure 10 Synchronized ES-task utilization w.r.t. taskset utilization, for multi-core *Sync-Sleep* scheduling ($m = 4$)

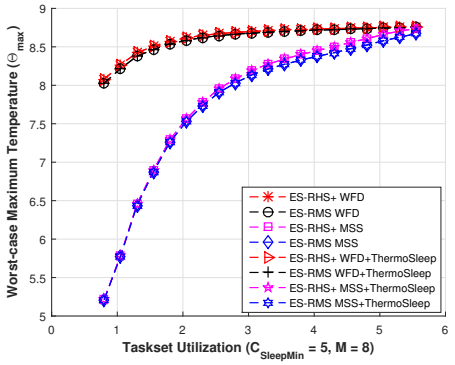


Figure 11 Worst-case maximum temperature w.r.t. taskset utilization, for multi-core *Sync-Sleep* scheduling ($m = 8$)

utilization for tasksets schedulable by all techniques. By using *SysSleep*, *ThermoSleep*-based techniques yield slightly greater ES-task utilization – up to 3.3% greater for ES-RMS. Figure 6 plots Θ_{max} versus taskset utilization for tasksets schedulable by all techniques. Despite the ES-task utilization being similar, by choosing a smaller ES-task period, *ThermoSleep* can achieve significantly lower temperatures – on average up to 4°K lower for ES-RMS, while simultaneously yielding better energy savings. Figure 6 also plots the average of the lower bound on Θ_{max} for the tasksets. On average, the worst-case deviation between the solution provided by ES-RMS and *ThermoSleep*, and the optimal lower bound was 0.028°K. Figure 7 plots Θ_{max} as a function of $C_{SleepMin}$, when taskset utilization $U_{taskset} = 0.4$. Observe that, despite varying $C_{SleepMin}$, our approach yields solutions with a worst-case temperature difference of 0.067°K compared to the optimal lower bound.

Multi-core SyncSleep Comparisons: We compare ES-RMS and ES-RHS+ on the basis of schedulability and the worst-case maximum temperature, Θ_{max}. We consider each technique using both WFD and *Max-SyncSleep* (MSS) for task partitioning, with and without using *ThermoSleep*. For a quad-core ($m = 4$) processor, Figure 8 plots schedulability versus taskset utilization, and Figure 10 plots the utilization of the synchronized ES-task $U_{SyncSleep}$. In

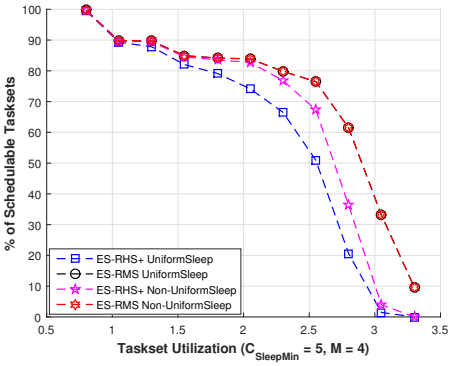


Figure 12 % of task sets schedulable w.r.t. taskset utilization, for multi-core *IndSleep* scheduling ($m = 4$).

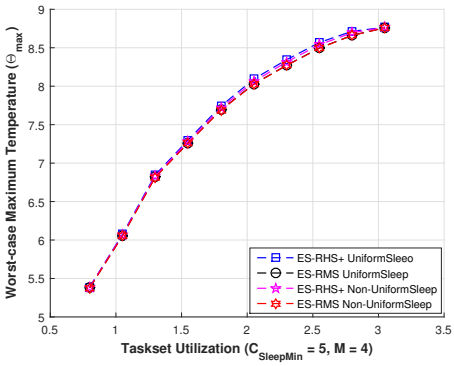


Figure 13 Worst-case maximum temperature w.r.t. taskset utilization, for multi-core *IndSleep* scheduling ($m = 4$).

terms of schedulability and $U_{SyncSleep}$ ES-RMS performs better than ES-RHS+ for all partitioning techniques. For partitioning techniques, MSS marginally dominates WFD in terms of schedulability and $U_{SyncSleep}$, both with and without using *ThermoSleep*. Using *ThermoSleep* provides marginally better $U_{SyncSleep}$ – up to 11.59% greater for ES-RMS with MSS. Figures 9 and 11 plot Θ_{max}, versus taskset utilization, for a quad-core ($m = 4$), and an octa-core ($m = 8$) processor respectively. Using *ThermoSleep* can give significantly lower Θ_{max} – on average up to 2.89°K lower for ES-RMS with MSS for $m = 4$.

Multi-core IndSleep Comparisons: We compare ES-RMS and ES-RHS+ using *Max-SyncSleep* to generate partitions. We consider using both uniform and non-uniform sleep for each core's ES-task. MSS along with *ThermoSleep* is used to determine the sleep periods. Figure 12 plots the percentage of schedulable tasksets. Note that using non-uniform sleep allows for greater schedulability – up to 1.2% greater for ES-RMS. Figure 13 plots Θ_{max} without considering inter-core heat dissipation. Note that, while ES-RHS+ provides maximal energy savings, in all cases ES-RMS yields lower temperatures than ES-RHS+. Additionally, due to better use of each core's *idle* durations, non-uniform sleep provides slightly lower temperatures than uniform sleep – up to 0.08°K.

6.2 Dynamic Simulations

To perform dynamic thermal simulation, we have designed a real-time multi-core scheduling simulation tool called *Inferno* (v1.0). Based on the processor floor-plan, prior temperature, power consumption in the interval and the interval length, *Inferno* uses Hotspot [37] to calculate each core's temperature, in each scheduler-simulation interval. *Inferno* supports fully-partitioned fixed-priority scheduling. Simulation parameters such as the number of cores, simulation cycles, simulation granularity, $C_{SleepMin}$, floorplan and thermal configuration can be specified by the user. The power consumption of each core for different operating frequencies in the *busy*, *idle* and *deep-sleep* states are specified in a look-up table. Based on the taskset and partition provided by the user, *Inferno* provides a trace of the power and temperature values at each simulation instant. The source code for *Inferno* can be found at `https://github.com/sandeepdsouza93/Inferno`.

In order to use realistic power values, we considered the automotive benchmark from the MiBench suite [19]. The benchmark was compiled and executed in the SniperSim [7] cycle-

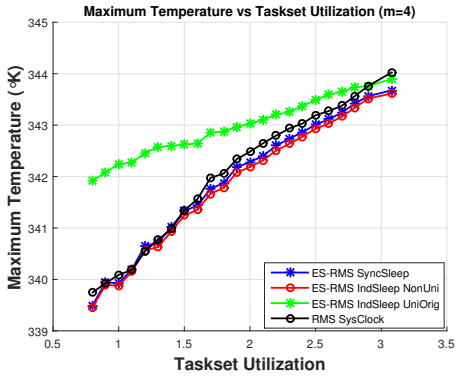


Figure 14 Dynamic simulation: maximum temperature w.r.t. taskset utilization ($m = 4$).

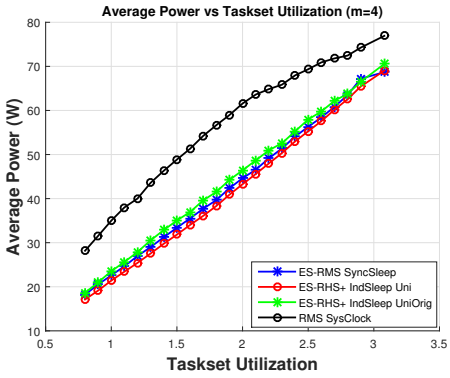


Figure 15 Dynamic simulation: average power w.r.t. taskset utilization ($m = 4$).

accurate x86 emulator (for a Nehalem-class x86 processor) for a range of frequency settings (1.22-2.66 GHz). The execution trace obtained from SniperSim is then fed to the McPAT [26] power simulator, which calculates the power consumption based on an x86 Nehalem power model (45 nm technology node). To model the dependency of static power on temperature, McPAT power calculations were done for the range of temperatures: 300-400°K, and the values were stored in a look-up table. *Inferno* uses these values to compute the core power consumption value, based on the previously calculated core temperature. The scheduling simulation granularity was set to 10μs, and Hotspot's default thermal configuration was used.

We have simulated a quad-core processor, with the floor-plan consisting of cores laid out in a square grid (as shown in Figures 2(a) and 3(a)). 10,000 randomly generated tasksets were considered, each containing 1 to 20 tasks. The taskset utilization varied from 0.8 to 3.2. Each taskset was simulated up to thermal steady state (Hotspot warm-up was considered).

Figure 14 plots maximum temperature versus taskset utilization. Observe that ES-RMS *IndSleep* with non-uniform sleep yields the lowest temperature. We also compared techniques from [12] following a purely energy-efficient design (UniOrig), and it returned the highest temperature – on average up to 3.91°K of difference between ES-RMS IndSleep without thermal considerations (UniOrig), and ES-RMS IndSleep with non-uniform sleep. We also compare our techniques with SysClock, which is the energy-optimal fixed-priority technique for static frequency scaling. We simulate SysClock with RMS where each core could have its own frequency. SysClock yields higher temperatures than IndSleep – up to 1.5°K higher.

Figure 15 plots the average power consumption versus taskset utilization. We find that by better utilizing the idle durations, ES-RHS+ IndSleep yields lower power consumption than ES-RMS SyncSleep – up to 5.04 W lower. ES-RHS+ IndSleep on average yields a power consumption that is 8.52 W lower than SysClock with a maximum difference of 21.74 W. This highlights the importance of energy-saving techniques based on sleep states. Our techniques also provide greater power savings compared to the purely energy-efficient design methodology presented in [12] – up to 8.36 W additional power-savings for ES-RHS+ IndSleep. Note that, although ES-RHS+ IndSleep provides greater energy savings, ES-RMS IndSleep yields lower temperatures. Additionally, even though SysClock consumes significantly more power than ES Schedulers, they both yield similar maximum temperatures. This highlights the fact that energy efficiency does not always imply lower temperatures.

7 Conclusions

In this paper, we analyze the thermal implications of fixed-priority energy-saving schedulers, which utilize the processor's deep-sleep state to save energy. We infer design choices from a well-known thermal model, and present two techniques for designing thermally-effective ES Schedulers: the *SysSleep* algorithm to provide optimal sleep utilization and the *ThermoSleep* heuristic to design a thermally-effective ES-task. Specifically, we derive a lower bound on the optimal maximum temperature, thus quantifying the best thermal performance achievable by ES Schedulers. In the multi-core context, we extend our analysis to two classes of scheduling problems [12]: *SyncSleep*, where cores need to synchronously transition into deep sleep, and *IndSleep*, where cores can independently transition into deep sleep. We consider the impact of both task partitioning and ES-task phasing on temperature. In the SyncSleep context, we observe that the synchronous deep-sleep constraint reduces the temperature-minimization problem to the energy-minimization problem, with the exception of the synchronous ES-task period calculation. On the other hand, while energy minimization is straightforward in the IndSleep context (all feasible partitions are optimal using ES-RHS+[12]), the same cannot be said for temperature minimization. The dependence of temperature on the ES-task periods and relative phasing makes the IndSleep problem non-trivial.

Since we focus on fully-partitioned scheduling, our proposed framework can be extended to heterogeneous multi-core processors. Additionally, our techniques do not require significant knowledge of a system's thermal parameters, and hence are applicable to a range of multi-core platforms. Static analysis and dynamic simulation validate our approach, yielding lower temperatures and better energy savings than both the purely energy-efficient ES Scheduler design [12], and frequency scaling based techniques [29]. Our results show that, while energy savings is key to lower temperatures, not all energy-efficient solutions yield low temperatures.

References

1. Intel Core Processor Family (4th Generation). [online]: `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/4th-gen-core-family-desktop-vol-1-datasheet.pdf`.
2. Samsung Exynos 5800. [online]: `http://www.samsung.com/semiconductor/minisite/Exynos/w/solution/mobile_ap/5420/`.
3. K. Agarwal, K. Nowka, H. Deogun, and D. Sylvester. Power gating with multiple sleep modes. In *International Symposium on Quality Electronic Design*, pages 633–637, 2006. `doi:10.1109/ISQED.2006.102`.
4. M. Ahmed, N. Fisher, S. Wang, and P. Hettiarachchi. Minimizing peak temperature in embedded real-time systems via thermal-aware periodic resources. *Sustainable Computing: Informatics and Systems*, 1(3):226–240, 2011. `doi:doi.org/10.1016/j.suscom.2011.05.006`.
5. R. Ahmed, P. Ramanathan, and K. K. Saluja. On thermal utilization of periodic task sets in uni-processor systems. In *International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 267–276, 2013. `doi:10.1109/RTCSA.2013.6732227`.
6. N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292, 1993. `doi:10.1.1.132.2794`.
7. T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. An evaluation of high-level mechanistic core models. *ACM Trans. Archit. Code Optim.*, 11(3):28:1–28:25, 2014. `doi:10.1145/2629677`.

8 Y. Chandarli, N. Fisher, and D. Masson. Response time analysis for thermal-aware real-time systems under fixed-priority scheduling. In *IEEE International Symposium on Real-Time Distributed Computing*, pages 84–93, 2015. doi:10.1109/ISORC.2015.34.

9 T. Chantem, R. P. Dick, and X. S. Hu. Temperature-Aware Scheduling and Assignment for Hard Real-Time Applications on MPSoCs. In *Design, Automation and Test in Europe*, pages 288–293, 2008. doi:10.1109/DATE.2008.4484694.

10 J. J. Chen, S. Wang, and L. Thiele. Proactive speed scheduling for real-time tasks under thermal constraints. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 141–150, 2009. doi:10.1109/RTAS.2009.30.

11 A. K. Coskun, T. S. Rosing, and K. Whisnant. Temperature aware task scheduling in mpsocs. In *Design, Automation Test in Europe Conference Exhibition*, pages 1–6, 2007. doi:10.1109/DATE.2007.364540.

12 S. D'souza, A. Bhat, and R. Rajkumar. Sleep scheduling for energy-savings in multicore processors. In *Euromicro Conference on Real-Time Systems*, pages 226–236, 2016. doi:10.1109/ECRTS.2016.16.

13 Intel Core2 Duo. [online]: http://download.intel.com/design/processor/datashts/320390.pdf.

14 P. Emberson, R. Stafford, and R. Davis. Techniques for the synthesis of multiprocessor tasksets. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems*, pages 6–11, 2010.

15 AMD Opteron Family. [online]: http://support.amd.com/TechDocs/40036.pdf.

16 N. Fisher, J. J. Chen, S. Wang, and L. Thiele. Thermal-aware global real-time scheduling on multicore systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 131–140, 2009. doi:10.1109/RTAS.2009.34.

17 Y. Fu, N. Kottenstette, C. Lu, and X. D. Koutsoukos. Feedback thermal control of real-time systems on multicore processors. In *ACM International Conference on Embedded Software*, pages 113–122, 2012. doi:10.1145/2380356.2380379.

18 M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co, 1990. doi:10.1137/1024022.

19 M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE International Workshop on Workload Characterization*, pages 3–14, 2001. doi:10.1109/WWC.2001.15.

20 P. M. Hettiarachchi, N. Fisher, M. Ahmed, L. Y. Wang, S. Wang, and W. Shi. The design and analysis of thermal-resilient hard-real-time systems. In *IEEE Real Time and Embedded Technology and Applications Symposium*, pages 67–76, 2012. doi:10.1109/RTAS.2012.17.

21 R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations: Symposium on the Complexity of Computer Computations*, pages 85–103, 1972. doi:10.1007/978-1-4684-2001-2_9.

22 N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, 2003. doi:10.1109/MC.2003.1250885.

23 T. Kuroda. Cmos design challenges to power wall. In *International Microprocesses and Nanotechnology Conference*, pages 6–7, 2001. doi:10.1109/IMNC.2001.984030.

24 J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium*, pages 166–171, 1989. doi:10.1109/REAL.1989.63567.

25 J. Y. T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982. doi:10.1016/0166-5316(82)90024-4.

26 S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *IEEE/ACM International Symposium on Microarchitecture*, pages 469–480, 2009. doi:10.1145/1669112.1669172.

27 C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973. doi:10.1145/321738.321743.

28 A. Rowe, K. Lakshmanan, H. Zhu, and R. Rajkumar. Rate-harmonized scheduling for saving energy. In *IEEE Real-Time Systems Symposium*, pages 113–122, 2008. doi:10.1109/RTSS.2008.50.

29 S. Saewong and R. Rajkumar. Practical Voltage-Scaling for Fixed-Priority RT-Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 106–115, 2003. doi:10.1.1.123.1440.

30 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, 1990. doi:10.1109/12.57058.

31 J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The impact of technology scaling on lifetime reliability. In *International Conference on Dependable Systems and Networks*, pages 177–186, 2004. doi:10.1109/DSN.2004.1311888.

32 R. Viswanath, V. Wakharkar, A. Watwe, and V. Lebonheur. Thermal performance challenges from silicon to systems. *Intel Corp. Manufacturing Group*, 2000. doi:10.1.1.14.8322.

33 S. Wang, J. J. Chen, Z. Shi, and L. Thiele. Energy-efficient speed scheduling for real-time tasks under thermal constraints. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 201–209, 2009. doi:10.1109/RTCSA.2009.29.

34 H. Wei, K. Lin, W. Lu, and W. Shih. Generalized rate monotonic schedulability bounds using relative period ratios. *Information Processing Letters*, 107(5):142–148, 2008. doi:10.1016/j.ipl.2008.02.006.

35 F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *IEEE Annual Foundations of Computer Science*, pages 374–382, 1995. doi:10.1109/SFCS.1995.492493.

36 B. Yun, K. G. Shin, and S. Wang. Predicting thermal behavior for temperature management in time-critical multicore systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 185–194, 2013. doi:10.1109/RTAS.2013.6531091.

37 R. Zhang, M. R. Stan, and K. Skadron. Hotspot 6.0: Validation, acceleration and extension. *University of Virginia*, CS-2015-04.

Energy-Efficient Multi-Core Scheduling for Real-Time DAG Tasks

Zhishan Guo[1], Ashikahmed Bhuiyan[2], Abusayeed Saifullah[3], Nan Guan[4], and Haoyi Xiong[5]

1 **Department of Computer Science, Missouri University of Science and Technology, Rolla, MO, USA**
guozh@mst.edu
2 **Department of Computer Science, Missouri University of Science and Technology, Rolla, MO, USA**
abvn2@mst.edu
3 **Department of Computer Science, Wayne State University, Detroit, MI, USA**
saifullah@wayne.edu
4 **Department of Computing, Hong Kong Polytechnic University, Hong Kong**
csguannan@comp.polyu.edu.hk
5 **Department of Computer Science, Missouri University of Science and Technology, Rolla, MO, USA**
xiongha@mst.edu

Abstract

In this work, we study energy-aware real-time scheduling of a set of sporadic Directed Acyclic Graph (DAG) tasks with implicit deadlines. While meeting all real-time constraints, we try to identify the best task allocation and execution pattern such that the average power consumption of the whole platform is minimized. To the best of our knowledge, this is the first work that addresses the power consumption issue in scheduling multiple DAG tasks on multi-cores and allows intra-task processor sharing. We first adapt the decomposition-based framework for federated scheduling and propose an energy-sub-optimal scheduler. Then we derive an approximation algorithm to identify processors to be merged together for further improvements in energy-efficiency and to prove the bound of the approximation ratio. We perform a simulation study to demonstrate the effectiveness and efficiency of the proposed scheduling. The simulation results show that our algorithms achieve an energy saving of 27% to 41% compared to existing DAG task schedulers.

1998 ACM Subject Classification D.4.7 Real-Time Systems and Embedded Systems

Keywords and phrases parallel task, real-time scheduling, energy minimization, convex optimization

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.22

1 Introduction

Energy consumption remains the cornerstone in designing embedded systems which are mostly battery-operated. Energy-efficient and power-aware computing therefore are gaining increasing attention in the embedded systems research. It is important due to the market demand of increased battery life for portable devices. Moreover, reducing energy consumption could lead to smaller power bills. Being motivated by this goal, there has been a trend in embedded system design and development towards multi-core platforms. In order to better utilize the capacity of multi-core platforms, parallel computation (where an individual task

executes in multiple processors simultaneously) needs to be considered. For example, a recent study [25] has shown that

the energy consumption of executing certain workload perfectly distributed in two cores is significantly less than that of executing the same workload in one core at double frequency.

In this paper, we deal with tasks that are represented as DAGs – that are considered to be the most generalized model of deterministic parallel tasks. For such task models, several results are obtained on schedulability tests under various scheduling policies in [4] [7] [2]. Bonifaci et al. [7] prove a speedup bound of $2 - 1/m$ for Earliest Deadline First (EDF) and $3 - 1/m$ for Deadline Monotonic (DM) respectively, where m is the number of processors. For global EDF scheduling, these techniques are further generalized [2] with an improved pseudo-polynomial time sufficient schedulability test. Analysis of federated and global EDF scheduling is performed in [21] [22]. Processor-speed augmentation bounds for both preemptive and non-preemptive real-time scheduling on multi-core processors is derived in [30]. The work in [3] studies global EDF scheduling for conditional sporadic DAG tasks, which is an extension to the normal sporadic DAG task model. Certain conditional control-flow constructs (such as *if-then-else* constructs) can be modeled using the conditional sporadic DAG task model. Despite of those nice preliminary work on the schedulability analysis of parallel tasks, none of them addresses the energy/power consumption issue.

Energy-Aware Real-Time Scheduling. In the design of embedded systems, energy minimization is a prime requirement. Much work has been done on minimizing the energy cost with respect to sequential tasks for multi-core systems [14] [26] [25] [24]. Specifically, [25] and [26] present an energy efficient task partitioning scheme, where the cores are grouped in frequency islands. The authors in [1] considers both feasibility and energy-awareness while partitioning periodic real-time tasks on a multi-core platform. For EDF scheduling, they show that if the workload is balanced evenly among the processors, deriving optimal energy consumption and finding a feasible partition is NP-Hard. Till date, only little work has been done for energy-aware real-time scheduling of parallel task models. In general, minimizing energy/power consumption of a real-time system is challenging due to the complex (non-linear) relationship between frequency, energy consumption, and execution time of each task.

In this paper, we study the scheduling of a set of sporadic DAG tasks with implicit deadlines on a multi-core platform. To the best of our knowledge, this is the first work that addresses the power consumption issue in scheduling multiple DAG tasks on multi-core. We assume that all the cores that are assigned to a DAG task will always remain active which may lead to a non-negligible power consumption. In order to reduce this effect, we also allow intra-task processor sharing if any core is lightly loaded. First, it will balance the load among the cores and reduce idle time. Second, the required number of cores to schedule a task can be reduced. After merging the cores that are not required can be shut off completely. When the average case execution times are typically small compared to the worst-case execution time (WCET), the cores will remain idle (in that case the active power consumption will be minimized, please see the Power/Energy model described at Section 2). Specifically, we make the following *key contributions* in this paper.

- We propose a multi-processor scheduling algorithm along with the power consumption issues for sporadic DAG tasks with implicit deadlines.
- Under the federated scheduling and task decomposition framework, our table-driven scheduler is shown to be optimal in the sense of average power consumption (i.e., named sub-optimal due to extra constraints included).

- We further allow merging of processors that have been assigned to the same DAG task. We also propose an efficient processor merging technique that is widely applicable for energy-efficiency improvements to most of the existing work on federated DAG task scheduling. We formally prove the NP-completeness of the problem, propose an approximation algorithm, and prove the upper bound of its approximation ratio.
- Simulations are conducted to verify the theoretical results and demonstrate the effectiveness of our algorithm.

The rest of this paper is organized as follows. Section 2 presents the system model and formally defines our problem. Section 3 adapts the task decomposition scheme that transfers parallel DAG tasks into sequential ones and describes our (sub-)optimal federated scheduler based on segment extension and problem transformation (into a convex optimization with linear inequality constraints). Section 4 presents and analyzes the techniques for intra-DAG processor sharing. Section 5 implements gradient based solvers and compares the proposed method with other state-of-the-art schedulers. Section 6 discusses related work and Section 7 concludes the paper.

2 Background and System Model

We consider a multi-core platform of m identical cores to schedule a set of sporadic parallel tasks. The task set is denoted by $\tau = \{\tau_1, \tau_2, \cdots, \tau_n\}$, where each task $\tau_i (1 \leq i \leq n)$ is represented as a DAG with a minimum inter-arrival separation of T_i time units (often referred as the *period*). The *nodes* in a DAG stand for different execution requirements while the *edges* represent the dependencies among the corresponding execution requirements. A parallel task τ_i contains a total of n_i nodes, each denoted by $\mathcal{N}_i^j (1 \leq j \leq n_i)$. The *execution requirement* of node $\mathcal{N}_i^j$ is denoted by c_i^j. A directed edge from node $\mathcal{N}_i^j$ to node $\mathcal{N}_i^k (\mathcal{N}_i^j \rightarrow \mathcal{N}_i^k)$ implies that the execution of $\mathcal{N}_i^k$ cannot start until $\mathcal{N}_i^j$ finishes for every instance (*precedence constraints*). $\mathcal{N}_i^j$, in this case, is called a *parent* of $\mathcal{N}_i^k$, while $\mathcal{N}_i^k$ is a *child* of $\mathcal{N}_i^j$. The *degree of parallelism* M_i of a DAG task τ_i is the number of nodes that can be simultaneously executed.

Each DAG τ_i has an *execution requirement* (i.e., *work*) of C_i which is the sum of the execution requirements of all of its nodes; i.e., $C_i = \sum_{j=1}^{n_i} c_i^j$. A *critical path* of a DAG task is a directed path with the maximum total execution requirements among all other paths in the DAG. For τ_i, the *critical path length*, denoted by L_i, is the sum of execution requirements of the nodes on a critical path. Thus, L_i is the *minimum makespan* of τ_i, meaning that it needs at least L_i time units even when the number of cores m is unlimited. Any two consecutive instances of task τ_i is separated by at least T_i time units – T_i is also the relative deadline of the task as we only consider *implicit deadlines*. 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. A DAG task is *heavy* if it will miss its deadline when all nodes are run sequentially on a processor. A schedule is said to be *feasible* when all sub-tasks (nodes) receive enough execution (up to their execution requirements) within T_i time units from their arrivals, while all precedence constraints are satisfied. The aforementioned terms are illustrated in Figure 1.

Power/Energy Model. Let $s(t)$ (we are assuming continuous frequency scheme) denote the main frequency (speed) of a processor at a certain time t. Then its power consumption $P(s)$ can be modeled as:

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

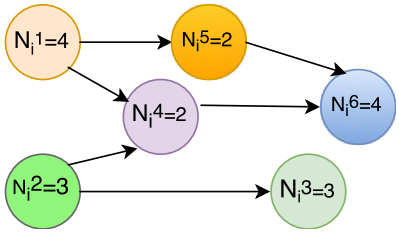


Figure 1 A DAG τ_i with total execution time $C_i = 18$ and minimum inter-arrival separation $T_i = 12$. It is a heavy task since $C_i > T_i$. The path $\mathcal{N}_i^1 \to \mathcal{N}_i^4 \to \mathcal{N}_i^6$ is the critical path with minimum makespan of $P_i = 10 \leq T_i$. As a result, this task may meet its deadline provided enough processors.

where P_s denotes the static power consumption which is introduced in the system due to the leakage current and $P_d(s)$ denotes the active power consumption. $P_d(s)$ is introduced due to the switching activities 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 [25], $\gamma \in [2, 3]$ 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). Clearly, the power consumption function is a convex-increasing function of the processor frequency. By means of dynamic voltage and frequency scaling (DVFS), it is possible to reduce $P_d(s)$ by reducing the processor frequency. In this paper, we focus on minimizing the active energy consumption (due to $P_d(s)$) by means of DVFS. We also target to minimize the static power consumption (due to P_s) by reducing the number of processors by allowing intra-task processor sharing.

The energy consumption of any given period $[b, f]$ can be calculated as $E = \int_b^f P(t)\, dt$, which is known as a nice approximation to the actual 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., $E(C, s) = (\beta + \alpha s^\gamma)(C/s) = \beta C/s + \alpha C s^{\gamma-1}$. Figure 2 shows how different values of γ (varying from 2 to 3) and processor speed s may affect the total energy consumption to complete a certain amount of computation. It is obvious that execution under a speed much lower than the *critical frequencies* [25] (the highlighted most energy efficient execution speed) is extremely energy inefficient (as leakage power becomes the major "contribution"). The power model we adapted complies with much existing (and recent) work in the community, e.g., [1, 33, 32, 13, 25, 26].

Problem Statement. Given a set of implicit-deadline sporadic parallel tasks to be scheduled on a multi-core platform consisting of enough[1] number of identical cores, we want to determine a *feasible* scheduling strategy, while minimizing the overall power consumption for the assigned processors.

Energy-optimal scheduling of parallel tasks on multi-cores is NP-hard in the strong sense [23]. Thus we do not expect to solve this energy optimization problem optimally in this paper. Instead, we will tackle this problem in the following two steps:

- First, we put additional constraints of federated scheduling and follow the existing task decomposition framework [30] (Sec 3), such that the NP-hardness no longer holds. We identify an energy-sub-optimal table-driven scheduler under those additional conditions.

[1] By enough, we mean the number of available processors is no smaller than the sum of max degree of parallelism of the DAG tasks.

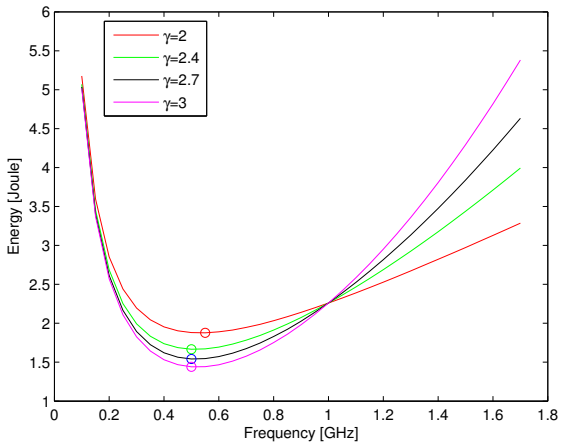


Figure 2 Energy consumption for executing a job with 10^9 computation cycles under various γ values, where $\alpha = 1.76$ W$atts/$GHz^{γ} and $\beta = 0.5$ W$atts$.

- Then, based on the "sub-optimal" solution, we propose heuristics for merging the assigned processors (Sec 4) to further improve the overall energy efficiency when the unnecessary restrictions are removed.

3 Energy-Sub-Optimal Federated Scheduling for DAG tasks

In this section, we restrict our focus on the federated scheduling of DAG tasks. Under the federated approach to multi-core scheduling, each individual task is either restricted to execute on a single processor (as in partitioned scheduling), or has exclusive access to all the processors on which it may execute. Since each processor is dedicated to one DAG task, we can consider each task individually, and try to minimize the energy consumption for a single DAG task (which is the goal of this section).

Given a DAG task, we first apply the existing task decomposition [30] technique to transform a parallel task into a set of sequential tasks with scheduling window [2] constraints for each node (Subsection 3.1) – they are further relaxed into necessary conditions by segment extension (Subsection 3.2). By variable substitution, we then transform the energy minimization problem into a convex optimization problem with linear inequality constraints, which can be solved *optimally* with gradient-based methods (Subsection 3.3).

3.1 Task Decomposition

Task decomposition is a well-known technique that simplifies the scheduling analysis of parallel real-time tasks [30]. In our approach, we adopt task decomposition as the first step that converts each node $\mathcal{N}_i^l$ of the DAG task τ_i to an individual sub-task τ_i^l with a release offset (b_i^l), deadline (f_i^l), and execution requirement (c_i^l). The release time and deadlines are assigned in a way that all dependencies (represented by edges in the DAG) are respected. Thus the decomposition ensures that if all the sub-tasks are schedulable then the DAG is

[2] Scheduling window for a specific node denotes the time frame from its release offset to its deadline.

also schedulable. For the sake of completeness, we briefly describe how task decomposition works in this subsection with an example (Please refer to Section 4 of [30] for more details).

We adapt a slightly modified version of the approach used in [30]. First, we perform the task decomposition using the techniques in [30] as described below. Assuming the execution of the task is on an unlimited number of cores, we draw a vertical line at every time instant where a node starts or ends for each node starting from the beginning. These vertical lines split the DAG into segments, and each segment consists of an equal amount of execution by the nodes that lie in the segment. Parts of different nodes in the same segment can now be considered as threads of execution that run in parallel, and the threads in a segment can start only after those in the preceding segment have finished their executions. Now we will say that the resulting segmented structure of the task is converted into synchronous form and will denote it as τ_i^{syn}. We first allot time to the segments and then add all times assigned to different segments of a node to calculate its allocated time.

Since $P_i \leq T_i$, at the end of each period, there may be a slack where all processors are idle (which is typically energy inefficient). We allocate such idle period *uniformly* by multiplying each segment by a common factor of T_i/P_i for task τ_i.

Task decomposition provides its processor assignment $\mathcal{M}_i^l$ (i.e., a node-to-processor mapping) and a scheduling window $[b_i^l, f_i^l)$ on top of it, in which each node $\mathcal{N}_i^l$ of a task τ_i will be scheduled. The following example demonstrates how task decomposition works.

▶ **Example 1.** Consider task τ_i shown in Figure 1. First of all, we assign all the nodes with no parent ($\mathcal{N}_i^1$ and $\mathcal{N}_i^2$) to separate processors. Then we continue to consider nodes only when all its parent node(s) are assigned. As a result, the beginning of the node will be the latest finishing time of its parent(s) – these are boundaries of the segments, denoted by vertical lines in Figure 3. Specifically, if a node has a single parent, we can start to consider the node right after the finishing time of its parent. For example, when $\mathcal{N}_i^2$ is completed, $\mathcal{N}_i^3$ is immediately assigned to the same processor (as $\mathcal{N}_i^2$ is the only parent).

When a node has multiple parents, we consider the parent that has the latest finishing time. The child node may be assigned to the same processor assigned to its parent with the latest finishing time. For example, $\mathcal{N}_i^4$ has two parents $\mathcal{N}_i^1$ and $\mathcal{N}_i^2$ where $\mathcal{N}_i^1$ completes execution later. So $\mathcal{N}_i^4$ is assigned to the same processor of $\mathcal{N}_i^1$. Please note that a node may have multiple siblings such that it may not always share the same processor with its latest finished parent node – under such scenario, a new processor is allocated to the node. For example, the only parent of $\mathcal{N}_i^5$ is $\mathcal{N}_i^1$ which completes execution at t_i^2. So $\mathcal{N}_i^5$ would be able to execute in the same processor starting from the third segment. But $\mathcal{N}_i^5$ is assigned to a different processor as that specific processor at t_i^3 is already "taken" by its sibling $\mathcal{N}_i^4$.

3.2 Segment Extension

For a DAG task τ_i, the aforementioned task decomposition results in a mapping between a node ($\mathcal{N}_i^l$) and a processor ($\mathcal{M}_i^l$). One of the key issues with the task decomposition process is that the identified scheduling window constraints for the nodes may not be necessary. Take the task described in Figure 3 as an example, where Node $\mathcal{N}_i^3$ may execute in the 4th segment. However, task decomposition requires that Node $\mathcal{N}_i^3$ must finish by the end of Segment 3, which is unnecessary. In this subsection, we describe a systematic way of eliminating such unnecessities so that the boundary constraints for all nodes (b_i^l's and f_i^l's) are both *necessary* and *sufficient*.

Each DAG τ_i is first converted to a synchronous form denoted by τ_i^{syn} with techniques described in Sec. 3.1. We use m_i^k to denote the number of parallel threads in the k-th segment

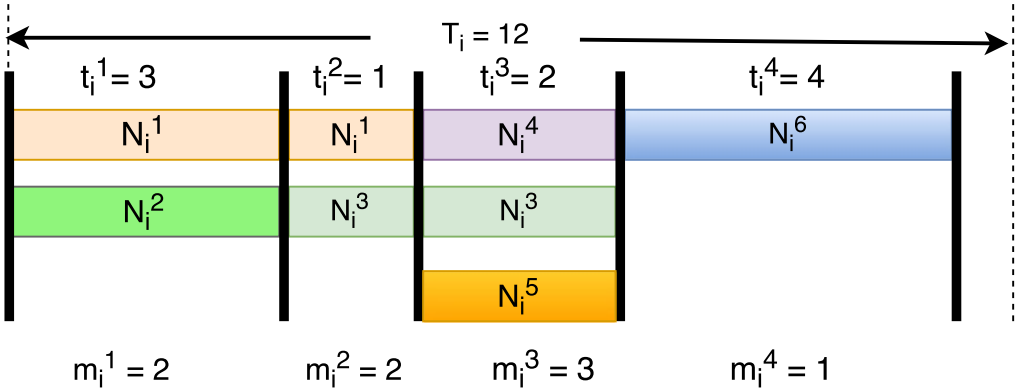


Figure 3 Scheduling window assignments to the nodes of τ_i (in Figure 1) after task decomposition, where m_i^k denotes the degrees of parallelism at k-th segment and the node-processor mapping is: $\bar{\mathcal{M}}_i = \{1, 2, 2, 1, 3, 1\}$, and scheduling windows for the nodes are $[1, 2], [1, 1], [2, 3], [3, 3], [3, 3], [4, 4]$ respectively. The average power consumption under such settings is 3.33 Watts after extending each segment by a common factor of $T_i/P_i = 1.2$.

of τ_i^{syn}. We then apply Algorithm 1 to greedily extend the deadlines f_i^l of each node $\mathcal{N}_i^l$, following any topological order. Note that while performing task decomposition, a node starts execution immediately when all of its predecessors finish execution. Thus the starting time b_i^l cannot be moved earlier – only f_i's have room to be relaxed.

Please note that we have considered table-driven schedulers which usually pre-compute which task would run when. This schedule is stored in a table at the time the system is designed or configured. So what would be the size of the scheduling table for a given set of real-time tasks to be run on a system? The answer is when a set of n tasks is to be scheduled, then the entries in the table will replicate themselves after LCM $(T_1, T_2 \cdots T_n)$, where LCM $(T_1, T_2 \cdots T_n)$ is the hyper-period for the tasks. However, while considering energy consumption we did not consider the space complexity of the scheduling solutions.

Algorithm 1 Segment Extension

1: **Input:** A DAG task τ_i, scheduling windows after decomposition $[b_i^l, f_i^l]$ for any node $\mathcal{N}_i^l \in \tau_i$.
2: **Output:** Extended segment window $[b_i^l, f_i^l)$ for each node $\mathcal{N}_i^l \in \tau_i$.
3: Assume that all nodes $\mathcal{N}_i^l$ are ordered topologically, such that predecessor constraint may only occur between $\mathcal{N}_i^l \longrightarrow \mathcal{N}_i^{l'}$ when $l < l'$.
4: **for** each node $\mathcal{N}_i^l \in \tau_i$ **do**
5: **if** node $\mathcal{N}_i^l$ has successor node(s); i.e., $\exists l', \mathcal{N}_i^l \longrightarrow \mathcal{N}_i^{l'}$
6: **then** $f_i^l \leftarrow \min_{l' | \mathcal{N}_i^l \longrightarrow \mathcal{N}_i^{l'}} \{b_i^{l'}\} - 1$;
7: **else** $f_i^l \leftarrow$ last segment of τ_i^{syn};
8: **end for**
9: **return** $[b_i^l, f_i^l]$ for each node $\mathcal{N}_i^l$.

▶ **Example 2.** Consider again the DAG task τ_i shown in Figure 1. Our algorithm greedily extends the ending segment f_i^l of the nodes as much as possible in the topological order (i.e., increasing l). Using this approach, Node $\mathcal{N}_i^3$ can now execute in Segment 4 (dashed rectangle at Figure 4) and the execution window for all the other nodes remain unchanged. Please note that the processor assignment $\mathcal{M}_i^l$ for any node $\mathcal{N}_i^l$ of a task τ_i remains unchanged in the segment extension process.

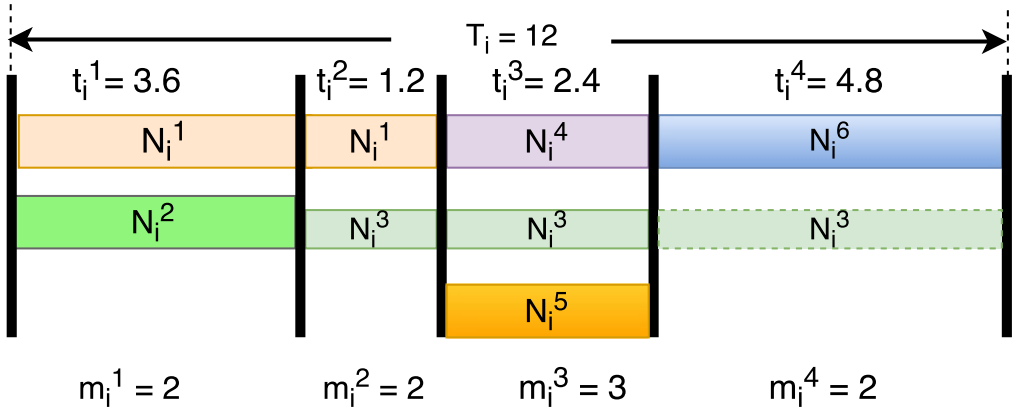


Figure 4 The segment-node mapping for τ_i (in Figure 1) after segment extension. Scheduling windows for the nodes become $[1,2],[1,1],[2,\mathbf{4}],[3,3],[3,3],[4,4]$ respectively, which results in an average power consumption of 3.08 Watts. The height of each block represents the speed of the processor during each segment.

▶ **Lemma 3.** *Under the task decomposition and scheduling framework, after running Algorithm 1 (Segment Extension), the timing constraints we set for each node in a DAG become* necessary *and* sufficient*.*

Proof. The sufficient part is trivial. The scheduling window satisfies all predecessor constraints, while the deadline of the DAG task does not change.

Assume the window after modification $[b_i^l, f_i^l]$ for some node $\mathcal{N}_i^l$ is not necessary; i.e., it can be further extended. Then it must be one of the following two cases:

- An earlier b_i^l still satisfies all predecessor constraints, which is impossible since it is the time all parents are finished.
- A later f_i^l is possible, which contradicts with Lines 5 - 7 of Algorithm 1 as it is already the starting point of its child, or the deadline of the whole DAG. ◀

3.3 Problem Transformation

After task decomposition and segment extension, we have identified the scheduling window $[b_i^l, f_i^l]$ for each node $\mathcal{N}_i^l$, and there is no overlap for any two windows (for different nodes) on the same processor. A natural question arises: *Given a specific node (job) with a predetermined scheduling window on a dedicated processor, what is the most energy-efficient execution (speed) pattern?*

▶ **Theorem 4.** *The total energy consumption (assuming processor remains on)* $\int_a^{a+\Delta} s(t)^\gamma \, dt$ *is minimized in any scheduling window* $[a, a+\Delta]$ *of length* Δ *when execution speed remains the same; i.e.,* $s(t) \equiv C/\Delta$*, where* $C = \int_a^{a+\Delta} s(t) \, dt$ *is the (given) task demand in the window.*

Proof. We define $p(x) = s(t)/C$, then $p(x)$ is a *probability density function (PDF)* over $[a, a+\Delta]$; i.e.,

$$\int_a^{a+\Delta} p(t) \, dt = 1. \tag{2}$$

As a result,

$$\begin{aligned}
&\int_a^{a+\Delta} s(t)^\gamma \, dt = \int_a^{a+\Delta} (C \cdot p(t))^\gamma \, dt \\
&\quad \{\text{re-arranging}\} \\
&\quad = \frac{C^\gamma}{\Delta^{\gamma-1}} \cdot \left(\frac{1}{\Delta} \int_a^{a+\Delta} (\Delta \cdot p(t))^\gamma \, dt \right) \\
&\quad \{\text{Jensen's Inequality [9], the convexity of function } x^\gamma \\
&\quad \text{when } 2 \le \gamma \le 3 \text{ and } x \ge 0, \text{ and } p(x) \text{ being a PDF}\} \\
&\quad \ge \frac{C^\gamma}{\Delta^{\gamma-1}} \left(\int_a^{a+\Delta} p(t) \, dt \right)^\gamma \\
&\quad \{\text{From (2)}\} \\
&\quad = \frac{C^\gamma}{\Delta^{\gamma-1}} \\
&\quad \{\text{Definition of integrating a constant function}\} \\
&\quad = \int_a^{a+\Delta} \left(\frac{C}{\Delta} \right)^\gamma \, dt.
\end{aligned} \tag{3}$$

Thus, the minimal total energy consumption in the specified interval $\int_a^{a+\Delta} s(t)^\gamma \, dt$ can be achieved when speed $s(t)$ remains constant (C/Δ) throughout the interval $[a, a+\Delta]$. ◀

According to Theorem 4, executing all segments with a uniform speed yields minimum possible power consumption under such framework. Hence we can assume that *the speed of any processor does not change within a segment*. Let S_j^k denote the speed of processor j in the k-th segment (executing node $\mathcal{N}_i^l$), and t_i^k denote the length of the segment. The objective is to determine the length of each segment $t_i^k (\ge 0)$ and its execution speed $S_j^k (\ge 0)$ such that total power consumption is minimized.

The first set of constraints guarantees the real-time correctness that each node $\mathcal{N}_i^l$ receives enough execution within its designated window $[b_i^l, f_i^l)$ on its assigned processor $\mathcal{M}_i^l$; i.e.,

$$\forall l, \mathcal{N}_i^l \in \tau_i : \sum_{k=b_i^l}^{f_i^l} t_i^k S_{\mathcal{M}_i^l}^k \ge c_{i,l}. \tag{4}$$

We need one more set of inequalities to bound the total length for all segments of each DAG by its period:

$$\forall i, \sum_k t_i^k \le T_i. \tag{5}$$

Any non-negative speed assignment and segment length setting that satisfy the constraints described in (4) and (5) yield a *correct* schedule that all nodes receive enough execution in their specified scheduling windows (that satisfy all predecessor constraints). Based on these constraints, we would like to add our objective for minimizing average energy consumption per period:

$$\textbf{Minimize}_{\{t_i^k, S_j^k\}} \quad M_i \beta T_i + \sum_{l=1}^{n_i} \sum_{k=b_i^l}^{f_i^l} t_i^k \alpha (S_{\mathcal{M}_i^l}^k)^\gamma,$$

where M_i is the degree of parallelism (and also the number of processors assigned to the task) and n_i is the total number of segments assigned to DAG task τ_i (determined in the previous step).

Since the constraints represented in (4) are non-convex quadratic inequalities, it is in general computationally intractable to solve in polynomial time. We transform this problem into a convex optimization by substituting some variables.

▶ Remark. *According to Theorem 4, executing all segments with a uniform speed yields minimum possible power consumption. If any segment of any core remains idle (scheduling window for any node does not fall at that segment), we simply consider that the execution speed for that segment is 0.*

▶ Remark. *In this paper, we are assuming that the time required to finish a task is exactly equaled to their worst-case execution time (WCET). However, it may happen that some of the tasks may finish early than their WCET. In that case, some of the cores (that are assigned to that tasks) may remain idle for some time. It would lead to the further minimization of the active power consumption (please see the Power/Energy model described at Section 2). So our model actually provides the upper bound of the energy consumption.*

Replacing speed with period lengths and executions. Fortunately, Theorem 4 provides us the basis to get rid of part of the variables. Since all nodes are executed at *constant* speeds within their scheduling windows, given the total length of each assigned segments (i.e., scheduling window), the execution speed of any given node can be determined. As a result, the energy consumption to finish this node can also be calculated. I.e., given a node $\mathcal{N}_i^l$ with total execution requirement of c_i^l, to be executed on segments between b_i^l and f_i^l, we have:

$$\forall k \in [b_i^l, f_i^l], S^k_{\mathcal{M}_i^l} = c_i^l / (\sum_{j=b_i^l}^{f_i^l} t_i^j), \tag{6}$$

which means although a node may be executed in consecutive segments $\forall k \in [b_i^l, f_i^l]$, the speed remains constant throughout the scheduling window and can be represented by a function of executions c_i^l and segment lengths t_i^j. Substituting Equation (6) into the second term of the objective function, we have:

$$\begin{aligned}
&\sum_{l=1}^{n_i} \sum_{k=b_i^l}^{f_i^l} t_i^k \alpha (S^k_{\mathcal{M}_i^l})^\gamma = \sum_{l=1}^{n_i} \left(\sum_{k=b_i^l}^{f_i^l} t_i^k \alpha (c_i^l)^\gamma (\sum_{j=b_i^l}^{f_i^l} t_i^j)^{-\gamma} \right) \\
&\quad \{\text{moving unrelated terms out of the summations}\} \\
&\quad = \alpha \sum_{l=1}^{n_i} \left((\sum_{j=b_i^l}^{f_i^l} t_i^j)^{-\gamma} (\sum_{k=b_i^l}^{f_i^l} t_i^k)(c_i^l)^\gamma \right) \\
&\quad \{\text{combining similar terms}\} \\
&\quad = \alpha \sum_{l | \mathcal{M}_i^l = j} c_l^\gamma (\sum_{k=b_i^l}^{f_i^l} t_i^k)^{1-\gamma}.
\end{aligned} \tag{7}$$

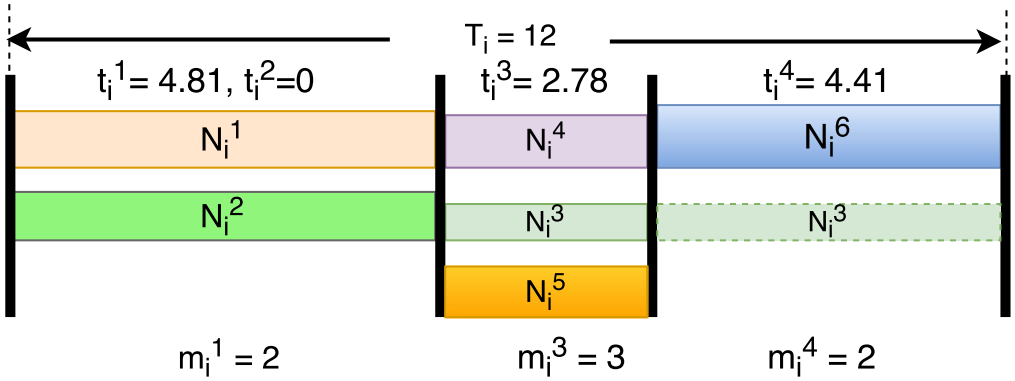


Figure 5 The sub-optimal segment length assignment for power efficiency of the sample task τ_i (in Figure 1), with an average power consumption of 2.94 Watts. The height of each block represents the speed of the processor during each segment.

Thus, the original optimization problem can be equivalently transformed into the following one with only t_i^k as variables.

$$\begin{aligned} \textbf{Minimize}_{\{t_i^k\}} \quad & M_i\beta T_i + \alpha \sum_{l|\mathcal{M}_i^l=j} c_l^\gamma \left(\sum_{k=b_i^l}^{f_i^l} t_i^k\right)^{1-\gamma} \\ \textbf{Subject to} \quad & \forall i, \sum_k t_i^k \le T_i, \\ & \forall i, t_i^k \ge 0. \end{aligned}$$

▶ **Theorem 5.** *Any gradient based method (e.g., fmincon [15] in Matlab) would lead to sub-optimal power consumption under federated scheduling scheme with task decomposition.*

Proof. The sub-optimality comes from three facts:

- The objective function is *convex* as it is a sum of several convex (including linear) functions of the variables t_i^k (detailed proof in Appendix A).
- The linear equality constraints are necessary and sufficient (Lemma 3) for real-time schedulability and predecessor conditions from the input DAG task.
- The variables t_i^k are sufficient to represent a possible optimal scheduler regarding power consumption; i.e., it is safe to assume uniform speed during each segment (Theorem 4). ◀

Figure 5 shows the sub-optimal segment length assignment for the given task τ_i.

4 Processor Sharing: Efficiency Improvement

Task decomposition transforms the parallel task into a set of sequential tasks. The process tries to maximize the degree of parallelism (i.e., assigning as many processors to each DAG task as possible). However, some of these processors may be lightly loaded with poor energy efficiency as the leakage power consumption becomes the majority cost (as demonstrated in Figure 2). Thus the solution derived in Sec 3 is only sub-optimal and can be further improved if we allow merging the lightly loaded processors into a single one, such that leakage power is reduced – see Figure 6 as an example.

In this section, we try to deal with this issue and further improve the overall energy efficiency of our scheduler by merging the workloads assigned to different processors onto a single one.

Specifically, in Subsection 4.1, we merge processors that have been assigned to the same DAG task. In this step, each DAG task is handled individually and the resulting processor-node/DAG assignment remains in the federated scheduling framework.

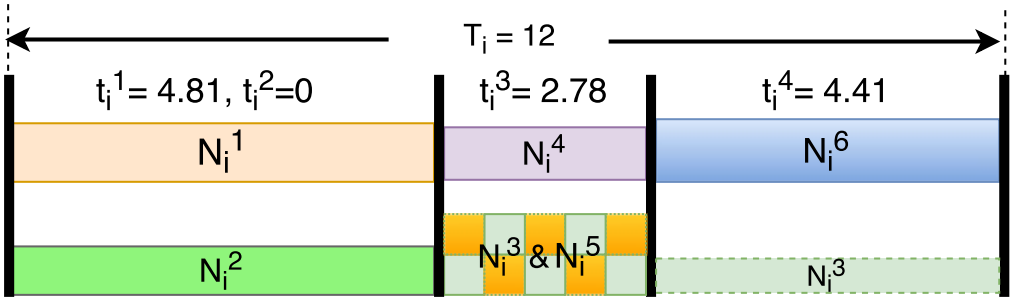


Figure 6 The execution pattern for τ_i (in Fig. 1) after merging Processors 2 and 3, where Nodes $\mathcal{N}_i^3$ and $\mathcal{N}_i^5$ will share Processor 2 (i.e., execute under EDF) within time window $[4.81, 7.59)$ at a higher execution speed. The average power consumption is further reduced to 2.80 Watts. The height of each block represents the speed of the processor during each segment.

▶ Remark. *In practice we have found that once a merge is performed, it is very likely that the new processor becomes quite heavily loaded. As a result, merging a third processor into any merged pair rarely leads to further energy saving. Thus* we only allow the combination of two processors that have never been part of any merging previously. *We plan to consider merging 3 or more processors into one in future work.*

▶ Remark. *In this paper we allow each processor to be merged only once. So the number of context switches is at most 1 per segments. If there are total n_i nodes in task τ_i then the number of context switches is at most $n_i/2$. Normally, the effect of task migrations and context switches is not considered while deriving schedulability test for real time tasks. We are also not considering the effects of these phenomena.*

4.1 Merging Processors Assigned to the Same DAG

Federated scheduling of DAG tasks provides isolation among tasks during execution, such that inter-task interference as well as context switch delays remain small during run-time. In this subsection, we stay in the federated scheduling framework and only consider potential merges among processors of the *same* DAG.

Given a DAG task τ_i with a federated task-to-processor assignment $j = \mathcal{M}_i^k$, the processor execution speeds S_j^k for each segment, segment lengths t_i^k, and the period T_i. For any processor j assigned to this DAG, its original power consumption can be calculated as

$$P_j = \beta + \sum_k \frac{t_i^k}{T_i} (S_j^k)^\gamma. \tag{8}$$

Any pair of processors $\{j, j'\}$ share the same period and segment information as they are assigned to the same DAG task. As a result, the new execution speed for each segment (when merged together) will simply be the sum of the two old ones; i.e., $S_j^l + S_{j'}^l$, and the average power consumption for this new processor can be calculated as:

$$P_{j,j'} = \beta + \sum_k \frac{t_i^k}{T_i} (S_j^k + S_{j'}^k)^\gamma. \tag{9}$$

The pairwise potential power saving can be calculated directly by:

$$\mathcal{P}_{j,j'} = P_j + P_{j'} - P_{j,j'}. \tag{10}$$

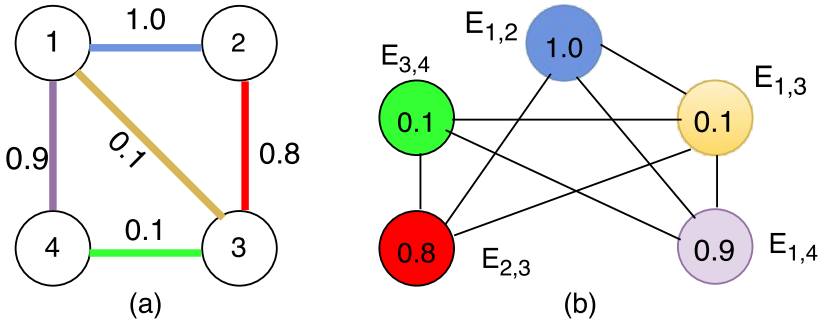


Figure 7 The equivalence of the MPS problem and the MIS problem, where (a) shows a DAG of four processor assignments with potential power savings for merging each pair of the processors, and (b) shows its alternative (equivalent) expression with vertices representing all edges in (a), and edges representing the mutual exclusive constraints.

With the pairwise potential power saving, the Maximization of Power Saving (MPS) problem we are dealing with in the section can be described as follows:

- Given the potential power savings ($\mathcal{P}_{M_i \times M_i}$) for merging each pair of the M_i processors, we wish to find a list of mutual exclusive processor-pairs $\{(p_1, p'_1), ..., (p_N, p'_N)\}(N \leq M_i/2)$, such that the total power saving $\mathcal{P}_i = \sum_{j=1}^{N} \mathcal{P}_{p_j, p'_j}$ is maximized.

▶ **Theorem 6.** *The MPS problem is NP-Complete.*

Proof. MPS is in NP as it takes linear time to verify whether a given solution satisfies the mutual exclusion constraints.

The NP-Hardness comes from the reduction from a well known NP-Complete problem: *Maximum Independent Set* (MIS). An independent set is a set of (weighted) vertices in a graph that no two of which are adjacent. For each vertex in the graph of MIS, we can construct an edge with the same weight in the graph of MPS, and the adjacency of those edges (whether or not they share a common vertex) in MPS can be determined by the adjacency of the edges in the graph of MIS; i.e., each edge in MIS corresponds to a vertex in MPS (see Figure 7 for an illustration). Since this polynomial (linear)-time mapping maintains the adjacency relationship of weighted vertices (in MIS) or edges (in MPS), a solution of MIS (a subset of n_m non-adjacent vertices with maximum total weight) will correspond to a solution of MPS (n_m non-adjacent edges with maximum total weight), and vice versa. ◀

▶ **Example 7.** Take the processor assignment in Figure 7 as an example, where four processors are assigned to a DAG task. The weight $\mathcal{P}_{i,j}$ for each edge represents the potential power saving when merging processors i and j, calculated from (10). The edge $\{2, 4\}$ is missing since merging these two processors will lead to higher power consumption (i.e., $\mathcal{P}_{2,4} < 0$).

For each vertex in Figure 7 (b), there is a corresponding edge with the same weight in Figure 7 (a), and vice versa. A feasible subset of edges in Figure 7 (a) (e.g., $\{1, 4\}$ and $\{2, 3\}$) corresponds to a subset of vertices in Figure 7 (b) (e.g., $E_{1,4}$ and $E_{2,3}$) that none of the two are directly connected by an edge.

For this example, we could choose to merge Processors 1&2 and 3&4 (with a gain of 1.1 Watts), 1&4 and 2&3 (with a gain of 1.7 Watts), or 1&3 (with a gain of 0.1 Watts). Although obviously the second option is leading to the optimal solution, we need to explore all combinations to find that out (Theorem 6 already shows the intractability). As a result, instead of seeking for the global optimal solution for merging, here we choose to greedily select (see Step 2 below) the pair with the maximum gain in each step.

Now we describe the **key steps of our proposed processor merging method**:

1. **For** *each* pair of processors $\{j, j'\}$ of the (same) DAG, calculate the potential power savings $\mathcal{P}_{j,j'}$ for merging them together according to (10).
2. *Greedily* choose the pair $\{j, j'\}$ of processors with the maximum power saving $\mathcal{P}_{j,j'}$, and merge them together by updating $\mathcal{P}'$ value(s) of the nodes on j' to j. The merged nodes will be executed on processor j under EDF, with given per-segment (fixed) speed settings. Note that EDF is an optimal uni-processor scheduler for sporadic task systems, and thus will guarantee temporal correctness as far as cumulative capacity remains the same.
3. *Remove* the two processors (and also the new one, see Remark 4) from the candidate pool, by updating elements in the jth row, the j'-th row, the jth column, and the j'-th column of the power saving matrix $\mathcal{P}$ into 0.
4. **If** there is no positive elements in $\mathcal{P}$, return the updated mapping $\mathcal{P}'$, **else** go to Step 2 (i.e., merging two un-touched processors may lead to further energy savings).

Although the MIS problem in general cannot be approximated to any constant factor in polynomial time (unless P = NP) [5], fortunately, each edge in the original figure can be joint with at most $2(M_i - 2)$ other edges, which indicates that the degree of each vertex in the graph after problem transformation is upper bounded by $2(M_i - 2)$. Thus we have the following *approximation ratio bound*.

▶ **Theorem 8.** *The greedy approach has an approximation ratio no greater than* $(2M_i - 2)/3$*, where* $M_i \geq 3$ *is the total number of processors*[3] *before merging of DAG task* τ_i*; i.e., the degree of parallelism of the task.*

Proof. Since we only allow a processor to be considered in *one* pair in each round, the graph resulted from the reduction in Theorem 6 is a $(2M_i - 4)$-regular graph; i.e., the degree of each vertex cannot exceed $2M_i - 4$. According to Theorem 5 in [18], the greedy algorithm achieves an approximation ratio of $(2M_i - 2)/3$. ◀

5 Simulation Study

In this section, we use experiments to evaluate the power efficiency of the proposed mechanisms, and compare them with existing algorithms for DAG task systems.

Generation of workloads. Our DAG generator follows the Erdos-Renyi method [12] with a given number of nodes. For the *harmonic period* case, the periods are multiples of each other [30] by enforcing them to be powers of 2. Specifically, we find the smallest value α such that $L_i \leq 2^\alpha$ and set T_i to be 2^α. Regarding the *arbitrary period* case, we use *Gamma distribution* [16] to generate a random parameter, and set the period as $T_i = L_i + 2(c_i/m)(1 + \Gamma(2, 1)/4)$ (according to [30]).

We compare the power consumption by varying two parameters: (i) task periods (densities) (Sec. 5.1) and (ii) number of nodes in each DAG task (Sec. 5.2). Under each parameter setting, we randomly generate 100 different DAG task sets, each consisting of 5 DAG tasks, and compare the average power consumption of the following scheduling algorithms:

- Federated scheduling with task decomposition, where each node is executed as soon as possible under full speed [30];

[3] Note that when $M_i = 2$, there are only two processors in the candidate pool, and the decision is straightforward – based on whether merging them can lead to lower power consumption.

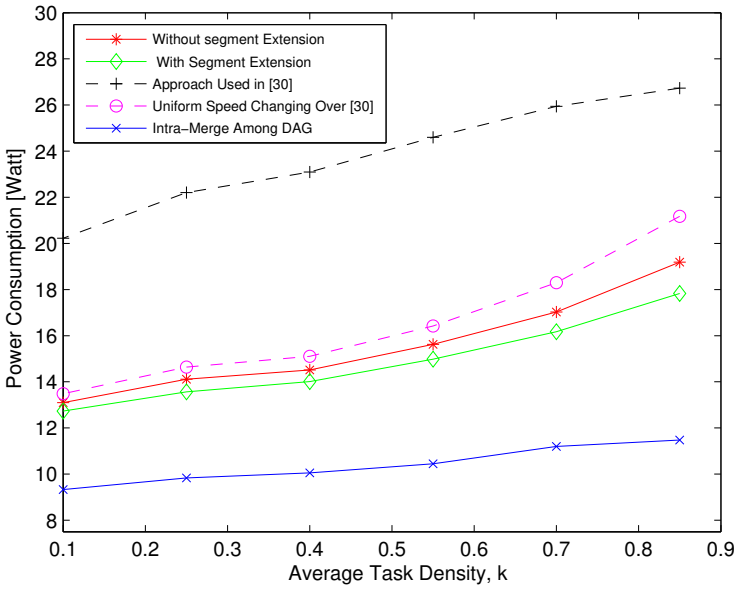


Figure 8 Comparison of power consumption with different approaches for DAG tasks with a fixed number of nodes as 30.

- Federated scheduling with task decomposition, where length of each segment is further extended uniformly (according to their loads) [30];
- Federated scheduling with task decomposition, where lengths of segments are determined by the proposed convex optimization (Sec. 3.3);
- Energy-sub-optimal federated scheduling with task decomposition, where lengths of segments are determined by convex optimization (Sec. 3.3) after performing segment extension (Sec. 3.2);
- Federated scheduling with intra-DAG processor merging (Sec. 4.1);

5.1 Varying Task Periods (Densities)

Here we vary the minimum inter-arrival separation for each task, such that the average density of a set is controlled. We vary the period in an allowable range ($P_i \leq T_i \leq C_i$) by assigning T_i as $P_i + (1-k)(C_i - P_i)$, where $k \in [0,1]$ is named as the *density* of the task – note that this is different from the normal density definition for sequential tasks. We fix the number of nodes within each DAG task as 30, and show the average power consumption in Figure 8.

The first thing we notice from Figure 8 is that the average energy consumption increases as the average density of the set increases (due to decreasing of the period). This phenomenon makes sense as higher density would lead to tighter real-time restrictions, which lead to less room for our segment length optimization.

As shown in Figure 8, stretching each segment would lead to significant power savings compared to finishing them at full speed and leaving the processor idle for some portion of time (matching Theorem 2). Comparing to the existing uniform stretching for all segments of each DAG task, our convex optimization based methods would find a better execution pattern in terms of power efficiency. We also found that segment extension is helpful in removing unnecessary constraints for finding better execution patterns.

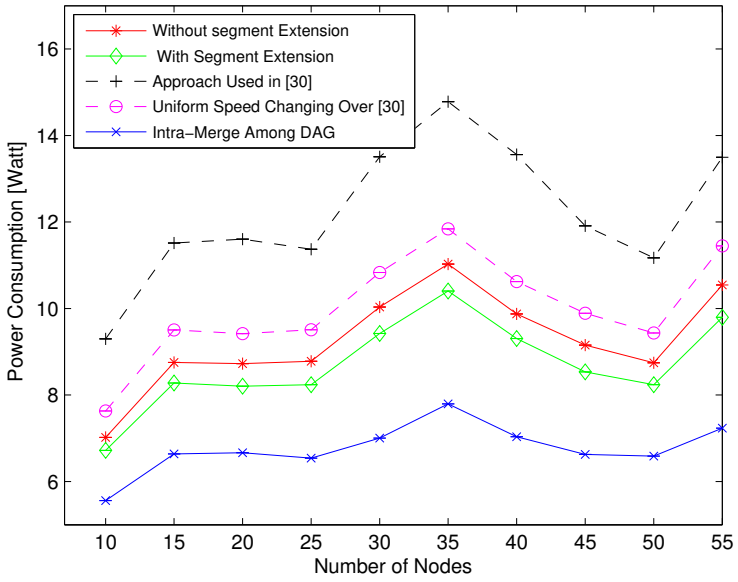


Figure 9 Comparison of average power consumption per task set with different approaches for tasks with harmonic periods.

It is easy to tell that the improvements to the average power consumption are huge when applying the processor merging techniques described in Sec. 4. The improvement is larger when density of the task is high. On average, our proposed methods (including segment extension and intra-DAG merging) are leading to a reduction of the power consumption ranging from 29.2% to 40.5%.

5.2 Different Numbers of Nodes in a DAG Task

Now we vary the number of nodes within each DAG task without changing the period T_i. In this set of comparisons, we consider both harmonic (reported in Figure 9) and arbitrary periods (reported in Figure 10) for a set. For each setting of parameters, we randomly generate 100 task sets with various number of nodes (from 10 to 55, with an increment of 5) and report the average performances of the power consumption over the 100 sets for each case.

First of all, we observe similar improvements in energy efficiency with the proposed techniques when the number of nodes vary, comparing to the previous set of experiments (with fixed number of nodes and varying task density). Specifically, the intra-DAG merging techniques discussed in Subsections 4.1 lead to a reduction in the power consumption for at least 27.29% and 34.27% for harmonic and arbitrary periods, respectively (compared to the result of convex optimization with segment extension discussed in Section 3.3), while the average power savings are 28.23% and 37.80%.

Secondly, when comparing curves in Figures 9 and 10, we observe that task sets with harmonic periods typically result in lower energy consumption compared to arbitrary periods (under same task density and number of nodes per task).

Finally, from the reported performances, we did not observe significant dependencies between the power consumption and the number of nodes for the DAG tasks. This indicates

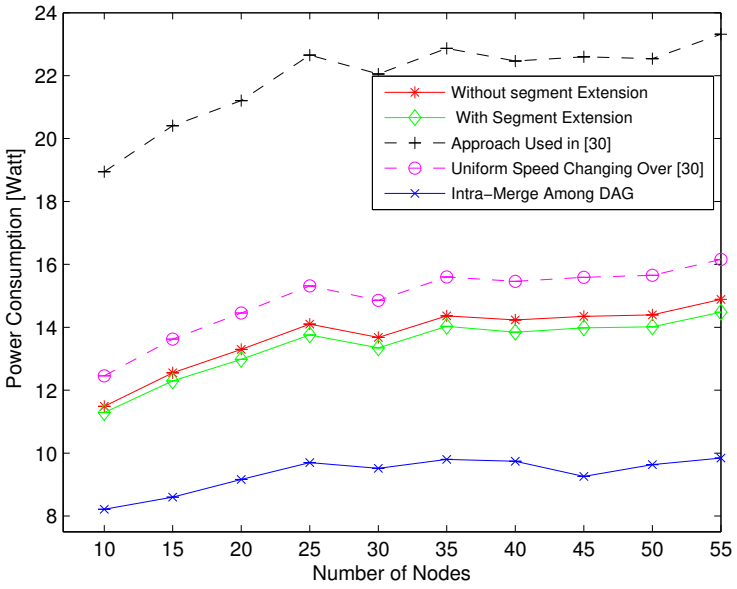


Figure 10 Comparison of average power consumption per task set with different approaches for tasks with arbitrary periods.

that the proposed methods are *robust* to various settings of parameters and combination of DAG tasks.

6 Related Work

The work that deals with schedulability tests for various scheduling policies on parallel task model is already mentioned in Section 1. None of them has considered power/energy consumption issues. In addition, much work has been done in energy/power consumption minimization for sequential tasks. Bini et al. discuss the problem of finding an optimal solution for a system with discrete speed levels for a set of periodic/sporadic tasks [6]. They have considered both EDF and Fixed-Priority (FP) scheduling policies. Jejurikar has considered non-preemptive tasks in order to deal with shared resources [19]. Chen et al. presents an energy-efficient design for heterogeneous multiprocessor platform [11]. No previous work considers parallel task model.

Actually, intra-task parallelization and power consumption issues have not yet received sufficient attention. Zhu et al. have considered power-aware scheduling for graph-tasks [34]. For dependent tasks, [10] provides techniques that combine dynamic voltage and frequency scaling (DVFS) and dynamic power management, where each core in the platform can be switched on and off individually. For block-partitioned multi-core processors (where cores are grouped into blocks and each block has a common power supply scaled by DVFS), energy efficiency is investigated in [29]. The authors in [28] consider power-aware policy for scheduling parallel hard real-time systems, where the multi-thread processing is used. [27] considers dealing with parallel tasks under Gang scheduling policy, where all parallel instances of a task use a processor in the same window. Based on level-packing, an efficient scheduling algorithm is proposed [20] [31]. The authors in [31] have considered energy minimization for frame-based tasks (i.e., same arrival time and a common deadline for all the tasks)

with implicit deadlines. Similar frame based model is considered in [17], where precedence constraints can be specified among the tasks. As mentioned previously, no existing work allows intra-task processor sharing, and considers the (more general) DAG task workload model.

7 Conclusion

This paper studies the scheduling of a set of sporadic DAG tasks with implicit deadlines. Upon guaranteeing real-time correctness, we try to minimize the overall power consumption of the whole platform. A power-sub-optimal scheduler is proposed under the condition of federated scheduling and task decomposition. Achieving the optimal solution for the more general (non-federated) case is shown to be NP-Complete. Based on the solution under federated scheduling, a greedy heuristic is proposed to further improve the power efficiency, with proved upper bound of the approximation ratio.

To our knowledge, this is the first work in the real-time systems community that (i) considers power issues for scheduling recurrent DAG tasks and (ii) allows intra-Task processor sharing. Still, our work has its restrictions: (i) during the processor merging process, we allow each processor to be merged only once – two or more merging may further reduce the power consumption; (ii) we only considered implicit deadlines, and the extension to constrained deadline case is not trivial; (iii) we have shown the evaluation through simulation. In the future, we plan to validate our algorithm in modern-generation processor to show how much the predicted energy savings correlate to the measurements on a real-life system.

References

1 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.

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

3 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.

4 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.

5 Cristina Bazgan, Bruno Escoffier, and Vangelis Th. Paschos. Completeness in standard and differential approximation classes: Poly-apx- and ptas-completeness. *Theoretical Computer Science*, 339(2-3):272–292, 2005.

6 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.

7 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.

8 Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

9 David Chandler. Introduction to modern statistical mechanics. *pp. 288. Foreword by David Chandler. Oxford University Press, Sep 1987. ISBN-10: 0195042778. ISBN-13: 9780195042771*, page 288, 1987.

10 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.

11 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.

12 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.

13 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.

14 Nathan Fisher, Jian-Jia Chen, Shengquan Wang, and Lothar Thiele. Thermal-aware global real-time scheduling on multicore systems. In *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, pages 131–140. IEEE, 2009.

15 `https://www.mathworks.com/help/optim/ug/fmincon.html`.

16 `http://en.wikipedia.org/wiki/Gammadistribution`.

17 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.

18 Magnús Halldórssonz and Jaikumar Radhakrishnan. Greed is good: Approximating independent sets in sparse and bounded-degree graphs. *Algorithmica*, 18(1):145–163, 1997.

19 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.

20 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.

21 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.

22 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.

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

24 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.

25 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.

26 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.

27 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.

28 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. ACM, 2016.

29 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.

30 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.

31 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.

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

33 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.

34 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.

A The Convexity of the Dynamic Energy Consumption

Since leakage power consumption remains constant (which is convex), we will prove that the dynamic part of the energy consumption function is convex:

$$E(\tau) = \sum_{1 \le i \le n} C_i^{\gamma} (< \alpha_i, \tau >)^{1-\gamma}. \tag{11}$$

Here τ refers to a k-dimension positive vector, in which each element is positive and refers to the length of a specific segment of a DAG task. α_i is a binary vector, in which each element $\alpha_{i,j} \in \{0, 1\}$ identifies if the node is selected for the segment. $|\alpha_i| \geq 1$ since at least one segment must be assigned). $< \alpha_i, \tau >$ refers to the inner-product of the two vectors, C_i refers to a non-negative constant, and $\gamma \in [2, 3]$. Thus the energy consumption is modeled as $E(\tau)$ – a function over the time-allocation $\tau \in \mathbb{R}_+^k$.

We prove the convexity of $E(\tau)$ when $\tau \in \mathbb{R}_+^k$ with the following four steps:

1. We name $f(\tau) =< \alpha, \tau >$ as a function of inner-product of τ with any binary vector α and $|\alpha| \geq 1$. Obviously, this function is a linear function over τ and should be both *convex* and *concave*. Further, given $\tau \in \mathbb{R}_+^k$, we have $f(\tau) > 0$. Thus we can conclude $f(\tau)$ is a *positive concave* function.
2. According to page 3-3 of [8], x^p is convex when $x > 0$ and $p \leq 0$. Thus, when $\gamma \in [2, 3]$ (i.e., $-2 \leq 1 - \gamma \leq -1$) and $x > 0$, the function $g(x) = x^{1-\gamma}$ should be a *non-increasing* convex function.
3. According to page 3-17 of [8], if $g(x)$ is a *non-increasing convex* function and $f(\tau)$ is a *concave* function over $\forall \tau \in \mathbb{R}_+^k$, then $g(f(\tau))$ should be a convex function over $\forall \tau \in \mathbb{R}_+^k$.

4. The function $E(\tau)$ and $f_i(\tau)$ could be written as:

$$E(\tau) = \sum_{1 \leq i \leq n} C_i^{\gamma} g(f_i(\tau)) \tag{12}$$

$$f_i(\tau) = (< \alpha_i, \tau >) \tag{13}$$

As C_i^{γ} is non-negative, $E(\tau)$ could be considered as the *non-negative-weighted sum* of convex functions (i.e., $g(f_i(\tau))$), and $E(\tau)$ should be a *convex function*.

Contego: An Adaptive Framework for Integrating Security Tasks in Real-Time Systems

Monowar Hasan[1], Sibin Mohan[2], Rodolfo Pellizzoni[3], and Rakesh B. Bobba[4]

1 **University of Illinois at Urbana-Champaign, Urbana, IL, USA**
mhasan11@illinois.edu
2 **University of Illinois at Urbana-Champaign, Urbana, IL, USA**
sibin@illinois.edu
3 **University of Waterloo, Ontario, Canada**
rodolfo.pellizzoni@uwaterloo.ca
4 **Oregon State University, Corvallis, OR, USA**
rakesh.bobba@oregonstate.edu

Abstract

Embedded real-time systems (RTS) are pervasive. Many modern RTS are exposed to unknown security flaws, and threats to RTS are growing in both number and sophistication. However, until recently, cyber-security considerations were an afterthought in the design of such systems. Any security mechanisms integrated into RTS must (a) *co-exist* with the real-time tasks in the system and (b) operate *without* impacting the timing and safety constraints of the control logic. We introduce Contego, an approach to integrating security tasks into RTS without affecting temporal requirements. Contego is specifically designed for *legacy* systems, *viz.,* the real-time control systems in which major alterations of the system parameters for constituent tasks is not always feasible. Contego combines the concept of *opportunistic execution* with hierarchical scheduling to maintain compatibility with legacy systems while still providing flexibility by allowing security tasks to operate in different *modes*. We also define a metric to measure the effectiveness of such integration. We evaluate Contego using synthetic workloads as well as with an implementation on a realistic embedded platform (an open-source ARM CPU running real-time Linux).

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

Keywords and phrases real-time systems, security, hierarchical scheduling

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.23

1 Introduction

Embedded real-time systems (RTS) are used to monitor and control physical systems and processes in many domains, *e.g.,* manned and unmanned vehicles including aircraft, spacecraft, unmanned aerial vehicles (UAVs), submarines and self-driving cars, critical infrastructures like the electric grid and process control systems in industrial plants, to name just a few. They rely on a variety of inputs for correct operation and have to meet stringent safety and timing requirements. Failures in RTS can have catastrophic consequences for the environment, the system, and/or human safety [1, 10].

Traditionally, RTS were designed using proprietary protocols, platforms and software and were not connected to the rest of the world, *i.e.,* they were air gapped. As a result cyber-security was not a design priority in such systems. However, the drive towards remote monitoring and control facilitated by the growth of the Internet, the rise in the use of

commercial-off-the-shelf (COTS) components, standardized communication protocols and the high value of these systems to adversaries have been challenging the status quo. While safety and fault-tolerance have long been important design considerations in such systems, traditional fault-tolerance techniques that were designed to counter and survive random or accidental faults are not sufficient to deal with cyber-attacks orchestrated by an intelligent and capable adversary. A number of high-profile attacks on real systems, *e.g.,* Stuxnet [15] and attack demonstrations by researchers on automobiles [20, 10] and medical devices [12] have shown that the threat is real.

Given the increasing cyber-security risks, it is essential to have a layered defense and integrate resilience against such attacks into the design of controllers and actuators (*i.e.,* embedded RTS). It is also critical to retrofit existing controllers and actuators with protection, detection, survival and recovery mechanisms. However, *any security mechanisms have to co-exist with real-time tasks in the system and have to operate without impacting the timing and safety constraints of the control logic.* This creates an apparent tension between security requirements (*e.g.,* having enough cycles for effective monitoring and detection) and the timing and safety requirements. For example, how often and how long should a monitoring and detection task run to be effective but not interfere with real-time control or other safety-critical tasks? While this tension could potentially be addressed for newer systems at design time, it is especially challenging in the retrofitting of *legacy* systems for which the control tasks are already in place and perhaps *cannot be modified.* Another challenge is to ensure that an adversary cannot easily evade such mechanisms. Further, the deterministic nature of task schedules in RTS may provide attackers with known windows of opportunity in which they can run undetected [11, 41].

Our focus in this work is on *retrofitting security mechanisms into legacy RTS*, for which modification of existing real-time tasks' parameters (such as run-times, period, task execution order, *etc.*) is not always feasible. In contrast to existing mechanisms [24, 42], the proposed method does *not* require any architectural modifications and hence is particularity suitable for systems designed using COTS components. The framework developed in this paper is based on our earlier work [18] in which we proposed to incorporate monitoring and detection mechanisms by implementing them as separate *sporadic tasks* and executing them *opportunistically*, that is, with the lowest priority so that real-time tasks are not affected. However, if the security tasks always execute with lowest priority, they suffer more interference (*i.e.,* preemption from high-priority real-time tasks) and the consequent longer detection time (due to poor response time) will make the security mechanisms less effective. In order to provide *better responsiveness* and increase the effectiveness of monitoring and detection mechanisms, we now propose a multi-mode framework called Contego[1]. For the most part, Contego executes in a PASSIVE mode with opportunistic execution of intrusion detection tasks as before [18]. However, Contego will *switch to an* ACTIVE *mode of operation* to perform additional checks as needed (*e.g.,* fine-grained analysis, used as an example in Section 6.2). This ACTIVE mode potentially executes with higher priority, while ensuring the schedulability of real-time tasks. Thus Contego subsumes the approach in our earlier work [18] and provides faster detection.

The contributions of this paper can be summarized as follows:

- We introduce Contego, an extensible framework to integrate security tasks into legacy RTS (Section 2).
- Contego allows the security tasks to execute with minimal perturbation of the scheduling

[1] A preliminary version [19] of this work was presented at a workshop without published proceedings.

order of the real-time tasks while guaranteeing their timing constraints (Sections 4–5). The proposed method can adapt to changes due to malicious activities by switching its mode of operation.

- We propose a metric to measure the security posture of the system in terms of frequency of execution (Section 3).
- We evaluate the schedulability and security of the proposed approach using a range of synthetic task sets and a prototype implementation on an ARM-based development board with real-time Linux (Section 6).

2 Security and System Model

2.1 Attack Model

RTS face threats in various forms, depending on the system and the goals of an adversary. For example, adversaries may insert, eavesdrop on or modify messages exchanged by system components, may manipulate the processing of sensor inputs and actuator commands and/or could try to modify the control flow of the system [42]. Further, rather than try to crash the system aggressively, an intruder in reconnaissance mode may want to monitor the system behavior and gather information for later use. For instance, an intruder may utilize side-channels to monitor the system behavior and infer system information (*e.g.,* hardware/software architecture, user tasks and thermal profiles, *etc.*) that may eventually help maximize the impact of an attack [11].

Let us consider an RTS (say an avionics electronic control unit) developed using a multi-vendor model [30], *viz.,* its components are manufactured and integrated by different vendors. For example, tasks in the system component manufactured by vendor v_i are very sensitive and considered classified or mission-critical (*e.g.,* images captured by the camera on the surveillance UAV). It may be undesirable for any vendor $v_j \neq v_i$ to gain unintended information about sensitive contents, even if, say, vendor v_j is trusted with control tasks for controlling the RTS. Similarly, the control laws from vendor v_j may contain a proprietary algorithm and vendor v_j may not want other vendors to gain knowledge about the algorithm. Protected communications and network monitoring/detection mechanisms are necessary but insufficient to deal with such threats. Therefore, *additional security tasks* may need to be added into the system to deal with such threats [25]. The security mechanisms could be protection, detection or response mechanisms, depending on the system requirements. For example, a sensor measurement correlation task may be added to detect sensor manipulation, a change detection task may be added to detect intrusions or additional state-cleansing tasks [26, 30, 27] can be added to deal with stealthy adversaries trying to glean sensitive information through side channels.

It is worth mentioning that the addition of such security mechanisms may necessitate changes to the schedule of real-time tasks. Contego is different from earlier work in which integration of security impacted the schedulability [27, 26, 30], required modification of the existing schedulers [39, 21], or necessitated architectural modifications [24, 42]. In contrast, Contego aims to integrate such security tasks *without* impacting the timeliness constraints (*i.e.,* schedulability) required for safe operation (in both modes) and retaining the original schedule of real-time tasks most of the time (*e.g.,* in PASSIVE mode when security tasks are executing opportunistically with lowest priority). We highlight that rather than designing specific intrusion detection tasks that target specific attack behaviors, the generic framework proposed in this work allows one to integrate a given security mechanism (referred to as

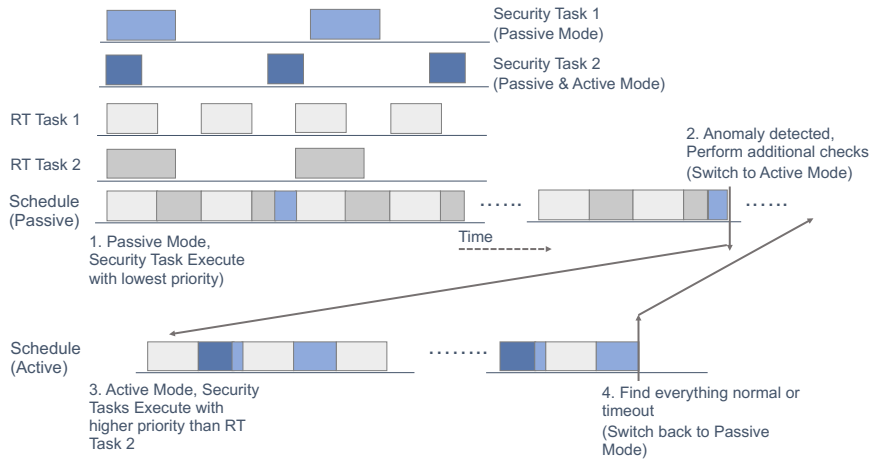


Figure 1 Contego: Flow of operations depicting the PASSIVE and ACTIVE modes for the security tasks.

security tasks) into the system without perturbing the system parameters (*e.g.,* period of the real-time tasks, execution order, *etc.*).

2.2 Overview of Contego

As illustrated in Fig. 1, Contego improves the security posture of the system (that contains a set of real-time tasks) by integrating additional security tasks and allowing them to execute in two different *modes* (*viz.,* PASSIVE and ACTIVE). If the system is deemed to be clean (*i.e.,* not compromised), security routines can execute *opportunistically*[2] (*e.g.,* when other real-time tasks are not running). However if any anomaly or unusual behavior is suspected, the security policy may switch to ACTIVE mode (*e.g.,* more fine-grained checking or response) and execute with *higher priority* for a *limited amount of time* (since our goal is to ensure security with minimum perturbation of the scheduling order of the real-time tasks). The security routines may go back to normal (*e.g.,* PASSIVE) mode if:

- No anomalous activity is found within a predefined time duration, say T^{AC}; or
- The intrusion is detected and malicious entities are removed (or an alarm triggered if human intervention is required).

Although we allow the security tasks to execute with higher priority than some of the real-time tasks in ACTIVE mode, the proposed framework ensures that the timeliness constraints (*e.g.,* deadlines) for *all* of the real-time tasks are always satisfied in *both* modes. By using this strategy, Contego not only enables *compatibility with legacy systems* (*e.g.,* in normal situation real-time scheduling order is not perturbed), but also provides *flexibility to promptly deal with anomalous behaviors* (*i.e.,* the security tasks are promoted to higher priority so that they can experience less preemption and achieve better response times).

[2] Which is also the default mode of operation.

2.3 System Model

2.3.1 Real-Time Tasks

In this paper we consider the widely used fixed-priority sporadic task model [28]. Let us consider a uniprocessor system consisting of m fixed-priority sporadic real-time tasks $\Gamma_R = \{\tau_1, \tau_2, \cdots, \tau_m\}$. Each real-time task $\tau_j \in \Gamma_R$ is characterized by (C_j, T_j, D_j), where C_j is the WCET, T_j is the minimum inter-arrival time (or period) between successive releases and D_j is the relative deadline. We assume that priorities are distinct and assigned according to the rate monotonic (RM) [22] order.

The processor utilization of τ_j is defined as $U_j = \frac{C_j}{T_j}$. Let $hp_R(\tau_j)$ and $lp_R(\tau_j)$ denote the sets of real-time tasks that have higher and lower priority than τ_j, respectively. We assume that the real-time task-set Γ_R is *schedulable* by a fixed-priority preemptive scheduling algorithm. Therefore, the worst-case response time w_i is less than or equal to the deadline D_i and the following inequality is satisfied for all tasks $\tau_j \in \Gamma_R$: $w_j \leq D_j$, where $w_j = w_j^{k+1} = w_j^k$ is obtained by the following recurrence relation [2]:

$$w_j^0 = C_j, \quad w_j^{k+1} = C_j + \sum_{\tau_h \in hp_R(\tau_j)} \left\lceil \frac{w_j^k}{T_h} \right\rceil C_h. \tag{1}$$

In Eq. (1), $\sum_{\tau_h \in hp_R(\tau_j)} \left\lceil \frac{w_j^k}{T_h} \right\rceil C_h$ is the worst-case interference to τ_j due to preemption by the tasks with higher priority than τ_j (*e.g.,* $hp_R(\tau_j)$). The recurrence will have a solution if $w_j^{k+1} = w_j^k$ for some k.

2.3.2 Security Tasks

With a view of integrating security into the system, let us add additional fixed-priority security tasks that will be executed in PASSIVE and ACTIVE modes. We model PASSIVE and ACTIVE mode security tasks as independent *sporadic tasks*. The PASSIVE and ACTIVE mode tasks are denoted by the sets $\Gamma_S^{pa} = \{\tau_1, \tau_2, \cdots, \tau_{n_p}\}$ and $\Gamma_S^{ac} = \{\tau_1, \tau_2, \cdots, \tau_{n_a}\}$, respectively. We assume that security tasks in both modes follow RM priority order. Each security task $\tau_i \in \{\Gamma_S^{pa} \cup \Gamma_S^{ac}\}$ is characterized by the tuple $(C_i, T_i^{des}, T_i^{max}, \omega_i)$, where C_i is the WCET, T_i^{des} is the most desired period between successive releases (hence $F_i^{des} = \frac{1}{T_i^{des}}$ is the desired execution frequency of a security routine) and T_i^{max} is the maximum allowable period beyond which security checking by τ_i may not be effective. The parameter $\omega_i > 0$ is a designer-provided weighting factor that may reflect the criticality of the security task[3] τ_i. Critical security tasks would have larger ω_i. The security tasks have implicit deadlines, *e.g.,* $D_i = T_i, \forall \tau_i$ that implies security tasks should complete before their next monitoring instance. We do not make any specific assumptions about the security tasks in different modes. For instance, both PASSIVE and ACTIVE mode task-sets may contain completely different sets of tasks (*e.g.,* $\{\Gamma_S^{pa} \cap \Gamma_S^{ac}\} = \emptyset$) or may contain (partially) identical tasks with different parameters (*e.g.,* period and/or criticality requirements).

In PASSIVE mode, security tasks are executed with *lower priority* than the real-time tasks. Hence the security tasks do not have any impact on real-time tasks and cannot perturb

[3] As an example, the default configuration of Tripwire [36], an intrusion detection system (IDS) for Linux that we use as case study in Section 6.2, has different criticality levels (*viz.,* weights), *i.e., High* (for scanning files that are significant points of vulnerability), *Medium* (for non-critical files that are of significant security impact) and so forth.

the real-time scheduling order. In ACTIVE mode, we allow the security tasks to execute with a priority higher than that of certain low priority real-time tasks. This provides us with a trade-off mechanism between security (*e.g.,* responsiveness) and system constraints (*e.g.,* scheduling order of real-time tasks). Since the task priorities are distinct, there are m priority-levels for real-time tasks (indexed from 0 to $m-1$ where level 0 is the highest priority). Among the m priority-levels, we assume that ACTIVE mode security tasks can execute with a priority-level up to l_S ($0 < l_S \leq m$), $l_S \in \mathbb{Z}$. Although any period T_i within the range $T_i^{des} \leq T_i \leq T_i^{max}$ is acceptable for PASSIVE (*e.g.,* $\tau_i \in \Gamma_S^{pa}$) and ACTIVE (*e.g.,* $\tau_i \in \Gamma_S^{ac}$) mode security tasks, the actual period T_i is not known a priori. Furthermore, for ACTIVE mode security tasks (*e.g.,* $\tau_i \in \Gamma_S^{ac}$), we need to find out the suitable priority level $l \in [l_S, m]$. Therefore our goal is to find the *suitable period* (for both PASSIVE and ACTIVE mode security tasks) as well as the *priority-level* (for ACTIVE mode security tasks) that achieve the best trade-off between schedulability and defense against security breaches without violating the real-time constraints.

3 Period Adaptation

As already mentioned, one fundamental problem in integrating security tasks is to determine *which* security tasks will be running *when*. This brings up the challenge of determining the *right periods* (*viz.,* the minimum inter-execution times) for the security tasks. For instance, some critical security routines may be required to execute more frequently than others. However, if the period is too short (*e.g.,* the security task repeats too often) then it will use too much of the processor time and eventually lower the overall system utilization. As a result, the security mechanism itself might prove to be a hindrance to the system and reduce the overall functionality or, worse, safety. In contrast, if the period is too long, the security task may not always detect violations, since attacks could be launched between two instances of the security task.

One may wonder why we cannot assign the desired period (*e.g.,* $T_i = T_i^{des}$) in both PASSIVE and ACTIVE modes and set the ACTIVE mode priority level as $l = l_S$ so that the security tasks can always execute with the desired frequency (*i.e.,* $F_i^{des} = \frac{1}{T_i^{des}}$) and experience less interference (*e.g.,* preemption) from real-time tasks. However, since our goal is to integrate security mechanisms in legacy systems with minimal[4] or no perturbation, setting $T_i = T_i^{des}$, $\forall \tau_i$ in either or both mode(s) may significantly perturb the real-time scheduling order. If the schedulability of the system is not analyzed after the perturbation, some (or all) of the real-time tasks may miss their deadlines and thus the main safety requirements of the system will be threatened. The same argument is also true for ACTIVE mode if we set $l = l_S$ (or arbitrarily from the range $[l_S, m]$) and do not perform schedulability analysis carefully.

3.1 Tightness of the Monitoring

As mentioned earlier, the actual period as well as the priority-levels of the security tasks are unknown and we need to *adapt* the periods within acceptable ranges. We measure the security of the system by means of *achievable periodic monitoring*. Let T_i be the period of the security task $\tau_i \in \{\Gamma_S^{pa} \cup \Gamma_S^{ac}\}$ that needs to be determined. Our goal is to minimize the

[4] In ACTIVE mode Contego does not introduce any timing violations for the real-time tasks, but their execution might be delayed due to interference from high-priority security tasks (*e.g.,* the tasks with priority-level $l \in [l_S, m]$).

gap between the achievable period T_i and the desired period T_i^{des} and therefore we define the following metric:

$$\eta_i = \frac{T_i^{des}}{T_i}, \tag{2}$$

that denotes the *tightness* of the frequency of periodic monitoring for the security task τ_i. Thus $\eta^{pa} = \sum_{\tau_i \in \Gamma_S^{pa}} \omega_i \eta_i$ and $\eta^{ac} = \sum_{\tau_i \in \Gamma_S^{ac}} \omega_i \eta_i$ denote the *cumulative tightness* of the achievable periodic monitoring for PASSIVE and ACTIVE mode, respectively. This monitoring frequency metric, provides for instance, one way to trade-off security with schedulability. Recall that if the interval between consecutive monitoring events is too large, the adversary may remain undetected and harm the system between two invocations of the security task. Again, a very frequent execution of security tasks may impact the schedulability of the real-time tasks. This metric $\eta^{(\cdot)}$ will allow us to execute the security routines with a frequency closer to the desired one while respecting the temporal constraints of the other real-time tasks.

3.2 Problem Overview

One may wonder why we cannot schedule the security tasks in the same way that the existing real-time tasks are scheduled. For instance, a simple approach to integrating security tasks in PASSIVE mode without perturbing real-time scheduling order is to execute security tasks at a *lower priority* than all real-time tasks. Hence, the security routines will be executing only during slack times when no other higher-priority real-time tasks are running. Likewise, in ACTIVE mode, security tasks can be executed at a lower priority than more critical, high-priority real-time tasks. Hence, the security tasks will only be executing when other real-time tasks with priority-levels higher than l_S are not running.

When both real-time and security tasks follow RM priority order, we can formulate a nonlinear optimization problem for PASSIVE mode with the following constraints that maximizes the cumulative tightness of the frequency of periodic monitoring:

(**P1**)

$$\max_{\mathbf{T^{pa}}} \quad \eta^{pa}$$

$$\text{Subject to:} \quad \sum_{\tau_i \in \Gamma_S^{pa}} \frac{C_i}{T_i} \leq (m+n_p)(2^{\frac{1}{m+n_p}} - 1) - \sum_{\tau_j \in \Gamma_R} \frac{C_j}{T_j} \tag{3a}$$

$$T_i \geq \max_{\tau_j \in \Gamma_R} T_j \qquad \forall \tau_i \in \Gamma_S^{pa} \tag{3b}$$

$$T_i^{des} \leq T_i \leq T_i^{max} \quad \forall \tau_i \in \Gamma_S^{pa} \tag{3c}$$

where $\mathbf{T^{pa}} = [T_1, T_2, \cdots, T_{n_p}]^\mathsf{T}$ is the optimization variable for PASSIVE mode that needs to be determined. The constraint in Eq. (3a) ensures that the utilization of the security tasks are within the remaining RM utilization bound [22]. The RM priority order for real-time and security tasks is ensured by the constraints in Eq. (3b), while Eq. (3c) ensures the restrictions on periodic monitoring.

Recall that in ACTIVE mode, we allow the security tasks to execute when the real-time tasks with priority-levels higher than l_S are not running. Hence, to ensure the RM priority order in ACTIVE mode, we need to modify the constraints in Eq. (3b) as follows:

$$T_i \geq \max_{\tau_j \in \Gamma_{R_{hp(l_S)}}} T_j, \quad \forall \tau_i \in \Gamma_S^{ac} \tag{4}$$

where $\Gamma_{R_{hp(l_S)}}$ represents the set of real-time tasks that are higher priority than level l_S. In addition, the constraints in Eq. (3a) and Eq. (3c) also need to be updated to consider ACTIVE mode task-sets (*e.g.,* Γ_S^{ac}) and the number of active mode security tasks (n_a). Thus for ACTIVE mode we can formulate an optimization problem similar to that of **P1** with the objective function: $\max_{\mathbf{T^{ac}}} \ \eta^{ac}$, where $\mathbf{T^{ac}} = [T_1, T_2, \cdots, T_{n_a}]^\mathsf{T}$ is the ACTIVE mode optimization variable.

One of the limitations of the above approach is that the overall system utilization is limited by the RM bound which has the theoretical upper bound of processor utilization only about $\lim_{n\to\infty} n(2^{\frac{1}{n}} - 1) = \ln 2 \approx 69.31\%$ [22], where n is the total number of tasks under consideration. Further, the security tasks' periods need to satisfy the constraints in Eq. (3b) and Eq. (4) (for PASSIVE and ACTIVE modes, respectively) to follow RM priority order. In addition, instead of focusing only on optimizing the periods of the security tasks, Contego aims to provide a *unified* framework that can achieve other security aspects (*viz.,* responsiveness). Thus we follow an alternative approach similar to one we proposed in earlier work[5] [18]. Specifically, we had proposed to use a *server* [13] to execute security tasks. Our security server is motivated by the needs of hierarchical scheduling [35]. Under hierarchical scheduling, the system is composed of a set of components (*e.g.,* real-time tasks and a security server, in our context) and each of which comprises multiple tasks or subcomponents (*e.g.,* security tasks). The server abstraction not only allows us to provide better isolation between real-time and security tasks, but also enables us to integrate additional security properties (such as responsiveness) as we discuss in the following.

4 The Security Server

The server [13] is an abstraction that provides execution time to the security tasks according to a predefined scheduling algorithm. Our proposed security server is characterized by the *capacity* Q and *replenishment period* P. The server is executed with lowest-priority in PASSIVE mode. However, in ACTIVE mode, the server can switch to any allowable priority-level[6] within the range $[l_S, m]$.

4.1 Reformulation of the Period Adaptation Problem using Servers

When security tasks execute within the server, we need to modify the constraints in the period adaption problem considering the server parameters Q and P. In the following we briefly discuss how to customize the period adaptation problem with the inclusion of the server.

Let us use $UB_{\mathcal{S}(Q,P),\Gamma}$ to denote the utilization bound for the set of tasks Γ executing within the server. When the smallest period of the task is greater than or equal to $3P - 2Q$, it has been shown [31] that the upper bound of the utilization factor for the security tasks is given by $UB_{\mathcal{S}(Q,P),\Gamma} = n\left[\left(\frac{3-\frac{Q}{P}}{3-2\frac{Q}{P}}\right)^{\frac{1}{n}} - 1\right]$, where n is number of tasks in the set Γ.

Thus with the inclusion of the server in PASSIVE mode, we can modify the constraints in

[5] The approach we proposed in our earlier work [18] is analogous to the PASSIVE mode of Contego.
[6] Calculation of the server priority-level is described in Section 5.

Eqs. (3a) and (3b) as follows:

$$\sum_{\tau_i \in \Gamma_S^{pa}} \frac{C_i}{T_i} \leq n_p \left[\left(\frac{3 - \frac{Q^{pa}}{P^{pa}}}{3 - 2\frac{Q^{pa}}{P^{pa}}} \right)^{\frac{1}{n_p}} - 1 \right] \tag{5a}$$

$$T_i \geq 3P^{pa} - 2Q^{pa}, \quad \forall \tau_i \in \Gamma_S^{pa}. \tag{5b}$$

Therefore, selection of the periods for security tasks in PASSIVE mode is a nonlinear constrained optimization problem that can be formulated as follows:

(**P2**)

$$\max_{\mathbf{T^{pa}}} \sum_{\tau_i \in \Gamma_S^{pa}} \omega_i \frac{T_i^{des}}{T_i}, \quad \text{Subject to: (5a), (5b), (3c).}$$

where Q^{pa} and P^{pa} are the server capacity and replenishment period in PASSIVE mode, respectively. The formulation of the PASSIVE mode period adaptation problem presented above is similar to that we proposed in earlier work [18]. Similarly, in ACTIVE mode, the period adaptation problem can be reformulated as follows:

(**P3**)

$$\max_{\mathbf{T^{ac}}} \sum_{\tau_i \in \Gamma_S^{ac}} \omega_i \frac{T_i^{des}}{T_i}$$

$$\text{Subject to: } \sum_{\tau_i \in \Gamma_S^{ac}} \frac{C_i}{T_i} \leq n_a \left[\left(\frac{3 - \frac{Q^{ac}}{P^{ac}}}{3 - 2\frac{Q^{ac}}{P^{ac}}} \right)^{\frac{1}{n_a}} - 1 \right] \tag{7a}$$

$$T_i \geq 3P^{ac} - 2Q^{ac} \quad \forall \tau_i \in \Gamma_S^{ac} \tag{7b}$$

$$T_i^{des} \leq T_i \leq T_i^{max} \quad \forall \tau_i \in \Gamma_S^{ac} \tag{7c}$$

where Q^{ac} and P^{ac} are the server capacity and replenishment period in ACTIVE mode, respectively.

4.2 Selection of the Server Parameters

The period adaptation problem illustrated in Section 4.1 is derived based on a given set of server parameters, *e.g.,* $(Q^{(\cdot)}, P^{(\cdot)})$. However, a fundamental problem is to find a suitable pair of server capacity $Q^{(\cdot)}$ and replenishment period $P^{(\cdot)}$ that respects the real-time constraints of the tasks in the system. Our approach to selecting the server parameters in PASSIVE and ACTIVE mode is described below.

4.2.1 Parameter Selection in Passive Mode

Recall that in PASSIVE mode, the server will execute with the lowest priority to have compatibility with existing real-time tasks. Since the security tasks execute within the server, we need to ensure the following two constraints:

- *The server is schedulable*: that is the server's capacity and interference from higher priority real-time tasks are less than the replenishment period; and
- *The security tasks are schedulable*: the minimum *supply* by the server to the security tasks is greater than the worst-case workload generated by the security tasks.

Note that since the server is running with lowest priority, the real-time constraints (*e.g.,* $w_j \leq D_j, \forall \tau_j \in \Gamma_R$) and the task execution order are not affected in the PASSIVE mode. Based on the above two constraints, we illustrate an approach for determining the server parameters by formulating it as a *constraint optimization problem.*

The security server is referred to as *schedulable* if the worst-case response time of the server does not exceed its replenishment period [13]. Thus, following an approach similar to ones in earlier work [18, 40], the *server schedulability constraint* can be represented as follows:

$$Q^{pa} + \Delta_{\mathcal{S}^{pa}} \leq P^{pa} \tag{8}$$

where $\Delta_{\mathcal{S}^{pa}} = \sum_{\tau_h \in hp_R(\tau_{\mathcal{S}^{pa}})} \left(\frac{P^{pa}}{T_h} + 1\right) C_h$ is the worst-case interference experienced by the server when preempted by the higher priority real-time tasks. In the above equation, the set of real-time tasks with higher priority than the server (*i.e.,* $hp_R(\tau_{\mathcal{S}}^{pa}) = \Gamma_R$) is fixed.

Let us use $hp_S^{pa}(\tau_i)$ to denote the set of PASSIVE mode security tasks that are higher priority than $\tau_i \in \Gamma_S^{pa}$. To ensure schedulability of the security tasks, we can derive the *minimum supply* of the server delivered to the security tasks by using the periodic resource model from the literature [35, 40, 18]. In particular, the constraints on the server supply to ensure *schedulability of the security tasks* [18] can be expressed as:

$$\frac{Q^{pa}}{P^{pa}} \left[T_i - (P^{pa} - Q^{pa}) - \Delta_{\mathcal{S}^{pa}}\right] \geq I_i^{pa}, \quad \forall \tau_i \in \Gamma_S^{pa} \tag{9}$$

where $I_i^{pa} = C_i + \sum_{\tau_h \in hp_S^{pa}(\tau_i)} \left\lceil \frac{T_i}{T_h} \right\rceil C_h$ is the worst-case workload generated by the security task τ_i and $hp_S^{pa}(\tau_i)$ during the time interval of T_i. This workload is a constant for a given input.

Since we need to ensure maximal processor utilization for the security tasks without violating the real-time constraints of the system, we define the following objective function: $\max_{Q^{pa}, P^{pa}} \frac{Q^{pa}}{P^{pa}}$. With this objective function and the constraints in Eqs. (8)–(9), the PASSIVE mode server parameter selection problem can be formulated as follows:

(**P4**)

$$\max_{Q^{pa}, P^{pa}} \frac{Q^{pa}}{P^{pa}}, \quad \text{Subject to: (8), (9)}$$

where server parameters Q^{pa} and P^{pa} are the optimization variables.

4.2.2 Parameter Selection in Active Mode

In ACTIVE mode, the security server is *no longer the lowest priority task.* Since the server can execute with priority l_S, there could be up to $m - l_S$ low priority real-time tasks than that of the server. Thus we need to ensure the schedulability of the real-time tasks that are executing with a priority lower than the server. Hence, in addition to the constraints described in Section 4.2.1 (*i.e.,* Eqs. (8)-(9)), we need to consider the following:

- *The real-time tasks with lower priority than the server are schedulable*: that is, the interferences from the server and other higher priority real-time tasks do not violate the deadlines for these low-priority tasks.

We therefore define the following constraints to ensure the *schedulability of the low-priority real-time tasks*:

$$C_j + \sum_{\tau_h \in hp_R(\tau_j)} \left\lceil \frac{D_j}{T_h} \right\rceil C_h + \left(\frac{D_j}{P^{ac}} + 1 \right) Q^{ac} \leq D_j, \quad \forall \tau_j \in lp_R(\tau_{\mathcal{S}}^{ac}) \tag{11}$$

where $\sum_{\tau_h \in hp_R(\tau_j)} \left\lceil \frac{D_j}{T_h} \right\rceil C_h$ is the interference experienced by τ_j from other real-time tasks and $\left(\frac{D_j}{P^{ac}} + 1 \right) Q^{ac}$ is the worst-case interference caused to τ_j by the server in ACTIVE mode. As illustrated in Section 5, we iterate through the allowable priority ranges (*e.g.,* $[l_S, m]$) to find the server priority in ACTIVE mode. Note that for a given priority-level, the set of tasks $lp(\tau_{\mathcal{S}}^{ac})$ is predefined. Thus the only variables for the constraints in Eq. (11) are the server capacity Q^{ac} and replenishment period P^{ac}.

Let us use $hp_S^{ac}(\tau_i)$ to denote the set of ACTIVE mode security tasks that are higher priority than $\tau_i \in \Gamma_S^{ac}$. Just as in **P4** we can now formulate the ACTIVE mode parameter selection problem as follows:

(**P5**)

$$\max_{Q^{ac}, P^{ac}} \frac{Q^{ac}}{P^{ac}}, \text{ Subject to: (11) and}$$

$$Q^{ac} + \sum_{\tau_h \in hp_R(\tau_{\mathcal{S}^{ac}})} \left(\frac{P^{ac}}{T_h} + 1 \right) C_h \leq P^{ac} \tag{12a}$$

$$\frac{Q^{ac}}{P^{ac}} \left[T_i - (P^{ac} - Q^{ac}) - \Delta_{\mathcal{S}^{ac}} \right] \geq I_i^{ac} \quad \forall \tau_i \in \Gamma_S^{ac} \tag{12b}$$

where the set of real-time tasks with higher priority than the server (*i.e.,* $hp_R(\tau_{\mathcal{S}}^{ac}) \subset \Gamma_R$) is a constant for a given priority-level and $I_i^{ac} = C_i + \sum_{\tau_h \in hp_S^{ac}(\tau_i)} \left\lceil \frac{T_i}{T_h} \right\rceil C_h$ is the worst-case workload generated by the security task τ_i and $hp_S^{ac}(\tau_i)$. Note that the schedulability of the higher priority real-time tasks (*e.g.,* $\forall \tau_j \in hp_R(\tau_{\mathcal{S}}^{ac})$) is already ensured by definition.

▶ Remark. The formulation of the period adaptation and server parameter selection problems are nonlinear constraint optimization problems and are nontrivial to solve in their current form. However, these problems can be transformed into a geometric programming (GP) [6] problem. In addition, it is also possible to reformulate the non-convex GP representation into equivalent convex form that can be solved using known algorithms such as *interior point* [7, Ch. 11] method. For details of this reformulation, we refer the readers to earlier work [18].

4.3 Discussion on Mode Switching

As mentioned earlier, by default, Contego operates in PASSIVE mode. However, when a malicious activity is suspected, a PASSIVE-to-ACTIVE mode change request will be issued. Similarly, an ACTIVE-to-PASSIVE mode change request will be placed if the system seems clean after fine-grained checking, or a malicious entity is found and removed. In steady-state (*e.g.,* when security tasks are executing in PASSIVE or ACTIVE mode), the schedulability of the real-time tasks is already guaranteed by the analysis presented in Section 4.2.

When Contego switches from PASSIVE mode to ACTIVE mode, the schedulability of real-time tasks will not be affected. The reason this that *all* the real-time tasks are higher priority than the security tasks in PASSIVE mode and hence do not suffer any additional

interference from security tasks during mode change. Therefore, the schedulability of real-time tasks during PASSIVE-to-ACTIVE mode switching is already covered by steady-state analysis (Section 4.2.1).

During ACTIVE-to-PASSIVE mode switching, observe that schedulability of the real-time tasks that have a priority higher than the server (*i.e.,* $hp_R(\tau_S^{ac})$) is not affected. When the mode switch request is issued, the ACTIVE mode server (and the security tasks) stop execution and the control is then switched to the lowest priority PASSIVE mode server. Note that the constraints in Eq. (11) that ensures the schedulability of the low-priority real-time tasks already captures the worst-case interference introduced by the server. Hence the server will not impose any more interference (even if the mode switch is performed in the middle of the execution of a busy interval) on the low-priority real-time tasks than what we have calculated in the steady-state analysis (Section 4.2.2). Therefore if both the PASSIVE and ACTIVE modes task-sets are schedulable, the system will also be schedulable with mode changes.

5 Algorithm Development

We develop a simple scheme to obtain the security task's period (for both PASSIVE and ACTIVE mode) and priority-level (for ACTIVE mode). The overall algorithm, Algorithm 1, works as follows.

To find the PASSIVE mode parameters, we initialize the security task's period with the desired period and solve the server parameter selection problem **P4** (Lines 10–11). If there exists a solution (*e.g.,* the constraints are satisfied), we then obtain the periods of the security tasks by solving **P2** (Line 13). In the event that neither of these optimization problems returns a solution, we report the task-set as unschedulable (Line 20), since it is not possible to execute security tasks opportunistically without violating real-time constraints.

To select ACTIVE mode parameters, the algorithm iterates through each of the acceptable priority-levels $[l_S, m]$ and tries to obtain the periods that maximize tightness for periodic monitoring without violating the real-time constraints (Lines 26–36). If there exists a solution (*e.g.,* constraints in **P5** and **P3** are mutually consistent), we store the solution in a candidate list. The algorithm then finds the best priority-level from the candidate solution sets that provides the maximum tightness (Line 39). In the event that no candidate solutions are found for any of the allowable priority ranges, the algorithm reports the task-set as unschedulable.

If *both* the PASSIVE and ACTIVE mode tasks are schedulable, then Algorithm 1 returns the corresponding periods and the ACTIVE mode priority-level (Line 4). Otherwise, the system is considered as unschedulable (Line 7) since it is not possible to integrate security tasks with desired requirements. This unschedulability result hints that the designers of the system should update system parameters (*e.g.,* the number of security tasks, desired and maximum allowable periods of the security tasks, periods of the real-time tasks, if permissible, *etc.*) in order to integrate security mechanisms.

6 Evaluation

We evaluate Contego with randomly generated synthetic workloads (Section 6.1) as well as a proof-of-concept implementation on an ARM-based embedded development board and real-time Linux (Section 6.2).

Algorithm 1 Feasibility Checking and Parameter Selection

Input: Set of real-time tasks, Γ_R, PASSIVE and ACTIVE mode security tasks Γ_S^{pa} and Γ_S^{ac}, allowable priority ranges $[l_S, m]$

Output: The tuple $\{l^*, \mathbf{T^{pa}}, Q^{pa}, P^{pa}, \mathbf{T^{ac}}, Q^{ac}, P^{ac}\}$, *e.g.,* ACTIVE mode server priority-level, ACTIVE and PASSIVE mode periods of the security tasks and ACTIVE and PASSIVE mode server parameters if the task-set is schedulable; Unschedulable otherwise

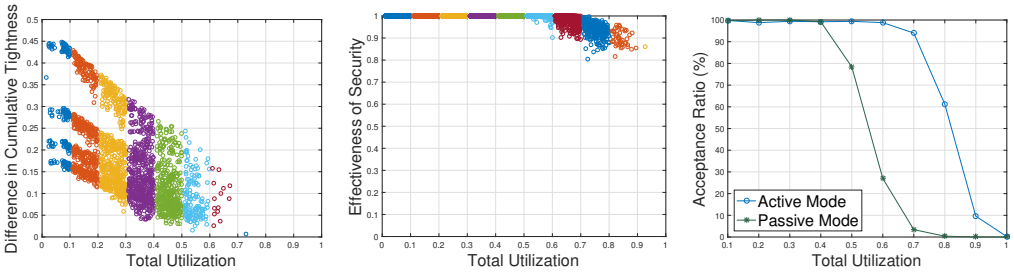
Obtain PASSIVE and ACTIVE mode parameters using the functions PassiveModeParamSelection(Γ_R, Γ_S^{pa}) and ActiveModeParamSelection(Γ_R, Γ_S^{ac}, l_S)
if Solution Found in BOTH Modes **then**
  / return the parameters */*
  return $\{l^*, \mathbf{T^{pa}}, Q^{pa}, P^{pa}, \mathbf{T^{ac}}, Q^{ac}, P^{ac}\}$
else
  / not possible to integrate security tasks in the system */*
  return Unschedulable
end if

---

function PassiveModeParamSelection(Γ_R, Γ_S^{pa})
  Initialize PASSIVE mode period $T_i := T_i^{des}, \forall \tau_i \in \Gamma_S^{pa}$
  Solve **P4** to obtain server parameters
  if SolutionFound **then**
    Solve **P2** to obtain security periods
    if SolutionFound **then**
      / return the parameters */*
      return $\mathbf{T^{pa}}$, Q^{pa}, P^{pa} where Q^{pa}, P^{pa} and $\mathbf{T^{pa}}$ are the solutions obtained by **P4** and **P2**
    end if
  else
    / unable to integrate* PASSIVE *mode security tasks */*
    return Unschedulable
  end if
end function

---

function ActiveModeParamSelection(Γ_R, Γ_S^{ac}, l_S)
  Schedulable := **false**
  Initialize ACTIVE mode security task's period $\mathbf{T}(l')_{\forall l' \in [l_S, m]} := [T_i^{des}]^{\mathsf{T}}_{\forall \tau_i \in \Gamma_S^{ac}}$
  for each priority level $l' \in [l_S, m]$ **do**
    Solve **P5** to obtain server parameters
    if SolutionFound **then**
      Solve **P3** to obtain security periods
      if SolutionFound **then**
        / store the parameters for priority level l' where Q^*, P^* and $\mathbf{T}^*$ are the solutions obtained by* **P5** *and* **P3** **/*
        $Q(l') := Q^*, P(l') := P^*, \mathbf{T}(l') := \mathbf{T}^*$
        Schedulable := **true**
      end if
    end if
  end for
  / obtain the parameters that provide best metric */*
  if Schedulable **then**
    Find the priority-level l^* from the solution vector $\mathbf{T}(l')_{\forall l' \in [l_S, m] |}$ tasks at l' is schedulable that gives the maximum cumulative tightness $\eta^{ac} = \sum_{\tau_i \in \Gamma_S^{ac}} \eta_i$
    Set $\mathbf{T^{ac}} := \mathbf{T}(l^*)$, $Q^{ac} := Q(l^*)$, $P^{ac} := P(l^*)$
    / return the parameters */*
    return l^*, $\mathbf{T^{ac}}$, Q^{ac}, P^{ac}
  else
    / unable to integrate* ACTIVE *mode security tasks */*
    return Unschedulable
  end if
end function



(a) Difference in cumulative tightness.

(b) Effectiveness of security in different system utilization.

(c) Schedulability of task-sets in both modes.

Figure 2 Experiments with synthetic task-sets: **(a)** PASSIVE mode vs. ACTIVE mode: difference in cumulative tightness of achievable periodic monitoring, $\eta^{av} - \eta^{pa}$. Non-zero difference indicates that the ACTIVE mode tasks achieve better tightness than PASSIVE mode tasks. Each of the data points represents schedulable task-sets. **(b)** The effectiveness of security vs. total utilization of the system. The closer the y-axis values to 1, the nearer each security task's period is to the desired period. **(c)** Schedulability of real-time and security tasks in both modes. The acceptance ratio is defined by the ratio of the number of accepted task sets over the total number of generated tasks. For each of the data points, 500 individual task-sets were tested. In figure (a) and (b), task-sets from different base-utilization groups are distinguished by different colors.

6.1 Experiment with Synthetic Task-sets

6.1.1 Simulation Setup

In order to generate task-sets with an even distribution of tasks, we grouped the real-time and security task-sets by base-utilization from $[0.01 + 0.1 \cdot i, 0.1 + 0.1 \cdot i]$, where $i \in \mathbb{Z} \wedge 0 \leq i \leq 9$. Each utilization group contained 500 task-sets. In other words, a total of 5000 task-sets were tested for each of the experiments. The utilization of the real-time and security tasks were generated by the UUniFast [4] algorithm and we used GGPLAB [29] to solve the optimization problems.

We used the parameters similar to those used in earlier research [26, 18]. In particular, each task-set instance contained $[3, 10]$ real-time and $[2, 5]$ security tasks in each of the modes. Each real-time task $\tau_j \in \Gamma_R$ had a period $T_j \in [10 \text{ ms}, 100 \text{ ms}]$ and we assumed $l_S = \lceil 0.4m \rceil$. The desired periods for the security tasks $\forall \tau_i \in \{\Gamma_S^{pa} \cup \Gamma_S^{ac}\}$ were selected from $[1000 \text{ ms}, 3000 \text{ ms}]$ and the maximum allowable period was assumed to be $T_i^{max} = 10T_i^{des}$. We considered $\omega_i = 1$, $\forall \tau_i \in \{\Gamma_S^{pa} \cup \Gamma_S^{ac}\}$ and the total utilization of the security tasks was assumed to be no more than 30% of the real-time tasks.

6.1.2 Results

6.1.2.1 Impact on Cumulative Tightness

In Fig. 2a one can see the difference in the tightnesses of the periodic monitoring obtained by PASSIVE and ACTIVE mode (*i.e.*, $\eta^{ac} - \eta^{pa}$). For fair comparison we used the same task-sets for both modes. The x-axis of Fig. 2a represents the total system utilization (*e.g.*, utilization of both real-time and security tasks). The positive values in the y-axis of Fig. 2a imply that the ACTIVE mode tasks obtain better tightness that the PASSIVE mode tasks.

The figure shows that ACTIVE mode tasks can achieve better cumulative tightness, and that the cumulative tightness η^{pa} is comparatively better in low to medium utilization. The main reason is that in ACTIVE mode security tasks are allowed to execute with higher priority,

that causes less interference and eventually increases the feasible region in the optimization problems (and hence provides better tightness). For higher utilizations the difference is close to zero. This is because, as utilization increases there is less slack in the system, making it difficult to schedule security tasks frequently and resulting in similar levels of tightness for both modes.

6.1.2.2 Effectiveness of Security

The parameter $\eta^{(\cdot)}$ is given by the total number of security tasks and provides insights on cumulative measures of security. However, in this experiment (refer to Fig. 2b) we wanted to measure the effectiveness of the security of the system by observing whether *each* of the security tasks in any mode can achieve an execution frequency closer to the desired one. Hence we used the following metric: $\xi = 1 - \frac{\|\mathbf{T}^* - \mathbf{T^{des}}\|_2}{\|\mathbf{T^{max}} - \mathbf{T^{des}}\|_2}$ where $\mathbf{T}^*$ is the solution obtained from Algorithm 1, $\mathbf{T^{des}} = [T_i^{des}]_{\forall \tau_i}^{\mathsf{T}}$ and $\mathbf{T^{max}} = [T_i^{max}]_{\forall \tau_i}^{\mathsf{T}}$ are the desired and maximum period vector (refer to Section 6.1.1), respectively, and $\|\cdot\|_2$ denotes the Euclidean norm. The closer the value of ξ to 1, the nearer each of the security task's period is to the desired period. As the total utilization increases, the feasible set of the period adaptation problem that respects all constraints in the optimization problems becomes more restrictive. As a result, we see the degradation in effectiveness (in terms of ξ) for the task-sets with higher utilization. However, from our experiments we find that Contego can achieve periods that are *within* 18% *of the desired periods.*

6.1.2.3 Impact on the Schedulability

We used the *acceptance ratio* metric to evaluate schedulability. The acceptance ratio (y-axis in Fig. 2c) is defined as the number of accepted task-sets (*e.g.,* the task-sets that satisfied all the constraints) over the total number of generated ones. As depicted in Fig. 2c the ACTIVE mode task-set achieves better schedulability compared to the PASSIVE ones. Recall that ACTIVE mode task-sets can be promoted up to priority level l_S. As a result ACTIVE mode security tasks potentially experience less interference than the PASSIVE ones. This flexibility gives the optimization routines a larger feasibility region to satisfy all the constraints.

6.2 Experiment with Security Applications in an Embedded Platform

To observe the performance of the proposed scheme in a practical setup, we implemented Contego on an embedded platform. Our experimental platform [3] was configured with 1 GHz ARM Cortex-A8 single-core processor and 512 MB RAM. We used Linux as the operating system – that allowed us to utilize the existing Linux-based IDSes (refer to Section 6.2.2) for the evaluation. Since the vanilla Linux kernel is unsuitable for hard real-time scheduling, we enabled the real-time capabilities with the Xenomai [38] 2.6.3 real-time patch (kernel version 3.8.13-r72) on top of an embedded Debian Linux console image.

We measured the WCET of the real-time and security tasks using ARM cycle counter registers (*e.g.,* CCNT), giving us nanosecond-level precision. Since these registers are not enabled by default, we developed a Linux kernel module to access the registers from our application codes. Our prototype implementation was developed in C and uses a fixed-priority scheduler powered by the Xenomai real-time patch. Sporadic real-time and security tasks in the system were defined by Xenomai `rt_task_create`() function and were suspended after the completion of corresponding instances using the `rt_task_wait_period`() function.

Table 1 Real-time task parameters for the UAV control system.

Task	Function	Period (ms)
Guidance	Select the reference trajectory (*i.e.,* altitude and heading)	1000
Controller	Execute closed-loop control functions (*e.g.,* actuator commands)	5000
Reconnaissance	Read radar/camera data, collect sensitive information and send data to the base control station	10000

Table 2 Security tasks used in the experiments.

Task	Function	Mode
Check own binary of the security routine (Tripwire)	Scan files (*viz.,* compare their hash value) in the following locations: `/usr/sbin/siggen`, `/usr/sbin/tripwire`, `/usr/sbin/twadmin`, `/usr/sbin/twprint`, `/usr/local/bro/bin`	ACTIVE
Check critical executables (Tripwire)	Scan file-system binary (`/bin`, `/sbin`)	ACTIVE and PASSIVE
Check critical libraries (Tripwire)	Scan file-system library (`/lib`)	ACTIVE
Monitor network traffic (Bro)	Scan predefined network interface (`en0`)	ACTIVE and PASSIVE

6.2.1 Real-time Tasks

For a real-time application, we considered a UAV control system (refer to Table 1). We implemented it using an open-source UAV model [37]. The original application codes were based on the STM32F4 micro-controller (ARM Cortex M4) and developed for FreeRTOS [16]. Because of differences in library support and execution semantics, we updated the source codes accordingly and ported them to Linux.

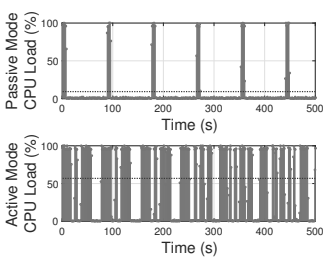
6.2.2 Security Tasks

To integrate security in the aforesaid control system, we included additional security tasks. For the security tasks, we considered two lightweight open-source intrusion detection mechanisms, (i) Tripwire [36], that detects integrity violations by storing clean system state during initialization and using it later to detect intrusions by comparing the current system state against the stored clean values, and (ii) Bro [8] that monitors anomalies in network traffic. As Table 2 shows, we consider several security tasks in both modes, *e.g., protecting security task's own binary files*, *protecting system binary and library files*, *monitoring network traffic*. In each mode, we set the desired and maximum allowable periods of the security tasks such that utilization of the security tasks did not exceed 50% of the total system utilization.

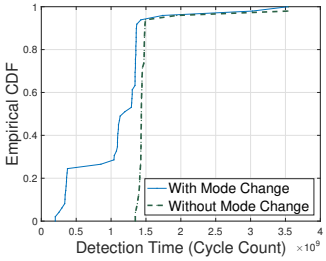
6.2.3 Experience and Evaluation

6.2.3.1 Performance Impact in Different Modes

In the first set of experiments, we measured the average CPU load when the security tasks were executing in PASSIVE and ACTIVE modes. For that, we executed the security tasks



(a) CPU load in different modes.



(b) Empirical CDF of detection time.

Figure 3 Experiments with synthetic task-sets: **(a)** The CPU load when the security tasks executed in PASSIVE (top) and ACTIVE (bottom) mode, respectively. The dotted line represents average load over the observation duration (500 s). **(b)** The empirical distribution of time to detect the intrusions when mode change was allowed vs when security tasks were run only in PASSIVE mode. We used ARM cycle counter registers to measure the detection time. A total of 50 individual experiment instances were examined to obtain the timing traces.

independently for 500 s in PASSIVE and ACTIVE modes and observed the CPU load using `/proc/stat` interface (represents the y-axis of Fig. 3a).

As Fig. 3a shows, running security tasks in ACTIVE mode increased the average CPU load compared to running them in PASSIVE mode. This is because ACTIVE mode contains more security tasks (*e.g.,* 4 compared to 2, refer to Table 2) and they execute more frequently than in PASSIVE mode. Because of the nature of applications, most RTS prefer predictability over performance. The overhead of running security tasks in ACTIVE mode comes with increased security guarantees that will suffice for many RTS.

6.2.3.2 Impact on Detection Time

To study the detection performance we injected malicious code into the system that mimics anomalous behaviors. We assumed that an attacker can take over[7] one of the low-priority real-time tasks (referred to as the victim task) and is able to insert malicious code that can execute with a privilege similar to that of legitimate tasks. We launched the attack at both the *network* and *host*-level. We defined network-level DoS attacks as too many rejected usernames and passwords submitted from a single address and used a real FTP DoS trace [17] to demonstrate the attack. Malware (such as LRK, tOrn, Adore, *etc.*) in general-purpose Linux environments causes damage to the system by modifying or overwriting the system binary [14, Ch. 5]. Thus we follow a similar approach to demonstrate a host-level attack, *viz.,* we injected ARM shellcode [33] to override the victim task's code and launched the attack by modifying the contents in the file-system binary. We obtained the periods of the security tasks in both modes by solving the period adaptation problem (Algorithm 1) and set it as the period of security tasks (by using the Xenomai `rt_task_set_periodic`() function). For each of the experiments, the work-flow was as follows. We started with a clean (*e.g.,* uncompromised) system state, launched the DoS attack at any random time of the program execution and then injected the shellcode after a random interval, and finally logged the time required by security tasks to detect the attacks. Initially the security tasks ran in PASSIVE

[7] One way to override a task could be to use an approach similar to one presented in the literature [11] that exploits the deterministic behavior of the real-time scheduling.

mode. When the network-level attack was suspected by the security task (*e.g.,* Bro), a mode change request was placed and the control was switched to ACTIVE mode with the corresponding ACTIVE mode tasks (see Table 2). As mentioned in Section 2.2, our focus is *not* on the effectiveness of a particular IDS here but on the effectiveness of integration of the IDSes into RTS. Therefore we controlled the experimental environment so that the results were not affected by the false positive/negative rates of the IDS used in the evaluation. In particular, both of the launched attacks were detectable by the respective IDSes used in the evaluation. Detection times were measured using ARM cycle counter registers (CCNT). To ensure the accuracy of the detection time measurements, we disabled all the frequency scaling features in the kernel (by using the `cpufrequtils` utility) and allowed the platform to execute with a constant frequency (*e.g.,* 1 GHz, the maximum frequency of our experimental platform).

We compared the performance of Contego with that of an earlier approach [18] that has no provision for mode changes and in which the security tasks are run with the lowest priority (similar to the PASSIVE mode of operation in Contego). Specifically, we measured the time to detect both the host and network-level intrusions, and plot the empirical cumulative distribution function (CDF) of those detection times in Fig. 3b. The x-axis in Fig. 3b represents the detection time (in cycle count) and the y-axis represents the probability that the attack would be detected by that time. The empirical CDF is defined as $\widehat{F}_{\alpha}(\jmath) = \frac{1}{\alpha}\sum_{i=1}^{\alpha}\mathbb{I}_{[\zeta_i \leq \jmath]}$, where α is the total number of experimental observations, ζ_i is the time taken to detect the attack in the i-th experimental observation, and $\jmath$ represents the x-axis values (*viz.,* the detection times in cycle count) in Fig. 3b. The indicator function $\mathbb{I}_{[\cdot]}$ outputs 1 if the condition $[\cdot]$ is satisfied and 0 otherwise.

From Fig. 3b we can see that Contego provides better detection time (*i.e.,* fewer cycle counts required to detect the intrusions). From our experiments we find that *on average* Contego detects attacks 27.29% faster than the reference scheme does. The approach from the literature [18] allows the security tasks to run only when other real-time tasks are not running, leading to more interference (*e.g.,* higher response times), and does not provide any mechanisms to adapt against abnormal behaviors (*e.g.,* the DoS attack in the experiments). In contrast, Contego allows quick response to anomalies (by switching to ACTIVE mode when a DoS attack is suspected). Since ACTIVE security tasks can run with higher priority and less interference without impacting the timeliness constraints of real-time tasks, Contego had a superior detection rate in general for most of the experiments without impacting safety.

7 Discussion

Although Contego provides an integrated approach to guarantee safety and security in RTS, this framework can be extended in several directions. In the following, we briefly analyze Contego against different threat models and discuss the limitations of the current framework with possible directions of improvement.

7.1 Threat Analysis

The security mechanism will collapse if the adversary can compromise *all* the security tasks. To do so, the adversary would need to intrude into the system, remain undetected and monitor the schedule [11] (to override the security tasks) *over a long period of time*. Guaranteeing the integrity of the security tasks is an interesting research problem by itself and will be investigated in our future work. While compromising all the security tasks

could be *difficult* in practice, it nevertheless would be worthwhile to harden the security posture of Contego further by *randomizing task schedules* while guaranteeing the safety of the real-time tasks by using approaches similar to one recently proposed in the literature [41]. Randomizing the schedule of real-time and security tasks reduces the determinism (and thus the predictability of security tasks' execution) and further reduce the chance of information leakage. Randomizing task schedules in RTS, unlike traditional systems, is not straightforward since it leads to priority inversions [32] that, in turn, cause missed deadlines, and hence, put the safety of the system at risk. We intend to incorporate randomization protocols on top of Contego in future work.

The underlying detection algorithms in security tasks could raise false positive errors that may cause the system to switch modes unnecessarily. Again, a clever adversary may remain undetected and provide a fake indication of malicious activity. This may cause Contego to frequently switch modes thus reducing performance and availability. Although Contego *guarantees that the system will remain schedulable* (and hence safe) even with mode changes (refer to Section 4.3), running of security tasks in ACTIVE mode could impose additional overheads (*i.e.,* increased load as we have seen in Fig. 3a) that designers of the system may want to avoid. The false positive/negative errors can be mitigated by carefully designing the detection algorithms based on application requirements. Further, we argue that forced mode changes would require an adversary to intrude in the system and *remain undetected for a long time*. In practice that could be *difficult* and *unlikely* in the presence of several intrusion detection tasks.

7.2 Limitations and Improvement

In Contego each security task has a desired frequency of execution for better security coverage. Security tasks so far have been treated as *independent* and *preemptive*, but in practice, some security monitoring may need *atomicity* or non-preemptive execution. Further, security tasks may have *dependencies* wherein one task depends on the output from one or more other tasks. For example, an anomaly detection task might depend on the outputs of multiple scanning tasks, or, the scheduling framework might need to follow certain *precedence constraints* for security tasks. In order to ensure the integrity of monitoring security, the security application's own binary might need to be examined first before it checks the system binary files. In that case, the *c*umulative tightness of the achievable periodic monitoring proposed in Section 3 might no longer be a reasonable metric. Constraints to ensure that the dependent security tasks are executed often enough should be included and the optimization problem may need to be reformulated and evaluated with different metrics.

While *time-to-detect* is a useful metric, it is hard to quantify in a comprehensive way as it depends on a number of factors such as the efficacy of monitoring tasks, the kind of intrusion *etc.* and is a lagging metric. Identifying and designing better security metrics is an important and challenging problem. In future work we will undertake it in the narrow context of integrating monitoring and detection tasks into RTS.

8 Related Work

In our earlier work [18] we proposed to use a server to integrate security tasks and execute them opportunistically at a lower priority than real-time tasks. That approach was useful for legacy RTS where perturbing the schedule of real-time tasks was not an option – however, the downside was longer time for detection. In contrast, Contego can respond to anomalous

activities in an adaptive manner and provide improved monitoring frequency and detection time when needed.

A state cleanup mechanism has been introduced [26], and further generalized [30, 27] such that the fixed-priority scheduling algorithm was modified to mitigate information leakage through shared resources. A new scheduler [39] and enhancements to an existing dynamic priority scheduler [21] were proposed to meet real-time requirements while maximizing the level of security achieved. Researchers have also proposed a schedule obfuscation method [41] aimed at randomizing the task schedule while providing the necessary real-time guarantees. Such randomization techniques can improve the security posture by minimizing the predictability of the deterministic RTS scheduler. Recent work [24, 42] on architectural frameworks has aimed to protect RTS against security threats. However, those approaches came at the cost of reduced schedulability or may require architectural/scheduler-level modifications. In comparison, Contego aims to integrate security *without* any significant modification of the system properties and does *not* violate the temporal constraints or schedulability of the real-time tasks.

Although not in the context of security in RTS, there exists other work [5] in which the authors statically assign the periods for multiple independent control tasks by considering control delay as a cost metric and estimating the delay through an approximate response time analysis. In contrast, our goal is to ensure security without violating the timing constraints of the real-time tasks. Hence, instead of minimizing response time, we attempt to assign the best possible periods and priority-levels so that we can minimize the perturbation between the achievable period and desired period for all the security tasks.

An on-demand fault detection and recovery mechanism has been proposed [23] in which the system can operate in different modes. Specifically, when a fault is detected, a high-assurance controller is activated to replace the faulty high-performance controller. While fault-tolerance may also be a design consideration, Contego focuses primarily on integrating mechanisms that can foil cyber-attacks. There also exist work in the context of mixed-criticality systems (MCS) where application tasks of different criticality requirements (*e.g.,* deadline and execution time) share same computation and/or communication resources (refer to literature [9] for a survey of MCS). MCS is different than the problem considered in this work due to the fact that security properties (*i.e.,* adaptive switching depending on runtime behavior or frequent execution of monitoring events for faster detection) are often different than temporal requirements (*e.g.,* satisfying deadline constraints for mixed-criticality tasks). However, the theory and concepts emerged from MCS can also be applied to the real-time security problems to further harden the security posture of future RTS.

9 Conclusion

The sophistication of recent attacks on UAVs [34], automobiles [20, 10], medical devices [12] as well as an industrial control systems [15], indicates that RTS are becoming more vulnerable. In this paper we are making steps towards the development of a comprehensive framework to integrate security mechanisms and provide a glimpse of *security design metrics* for RTS. Designers of RTS are now able to improve their security posture, which will also improve overall *safety* – and that is essentially the main goal of such systems.

References

1 Marshall Abrams and Joe Weiss. Malicious control system cyber security attack case study–Maroochy Water Services, Australia. *McLean, VA: The MITRE Corporation*, 2008.

2 Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *SE Journal*, 8(5):284–292, 1993.

3 BeagleBone Black. `https://beagleboard.org/black`.

4 Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *RTS Journal*, 30(1-2):129–154, 2005.

5 Enrico Bini and Anton Cervin. Delay-aware period assignment in control systems. In *IEEE RTSS*, pages 291–300, 2008.

6 Stephen Boyd, Seung-Jean Kim, Lieven Vandenberghe, and Arash Hassibi. A tutorial on geometric programming. *Opt. & Eng.*, 8(1):67–127, 2007.

7 Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge University Press, 2004.

8 The Bro Network Security Monitor. `https://www.bro.org`.

9 Alan Burns and Robert Davis. Mixed criticality systems – a review. Technical report, University of York, 2013. [Online]. URL: `https://www-users.cs.york.ac.uk/~burns/review.pdf`.

10 Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Sec. Symp.*, 2011.

11 Chien-Ying Chen, Rakesh B. Bobba, and Sibin Mohan. Schedule-based side-channel attack in fixed-priority real-time systems. Technical report, University of Illinois, 2015. [Online]. URL: `http://hdl.handle.net/2142/88344`.

12 Shane S. Clark and Kevin Fu. Recent results in computer security for medical devices. In *MobiHealth*, pages 111–118, 2011.

13 Rob Davis and Alan Burns. An investigation into server parameter selection for hierarchical fixed priority pre-emptive systems. In *IEEE RTNS*, 2008.

14 Ethical hacking and countermeasures: Secure network operating systems and infrastructures, 2017.

15 Nicolas Falliere, Liam O. Murchu, and Eric Chien. W32. Stuxnet dossier. *White paper, Symantec Corp., Security Response*, 5:6, 2011.

16 FreeRTOS. `http://www.freertos.org`.

17 FTP Brute-force attack trace. `https://github.com/bro/bro/blob/master/testing/btest/Traces/ftp/bruteforce.pcap`.

18 Monowar Hasan, Sibin Mohan, Rakesh B. Bobba, and Rodolfo Pellizzoni. Exploring opportunistic execution for integrating security into legacy hard real-time systems. In *IEEE RTSS*, pages 123–134, 2016.

19 Monowar Hasan, Sibin Mohan, Rakesh B. Bobba, and Rodolfo Pellizzoni. A server model to integrate security tasks into fixed-priority real-time systems. In *IEEE CERTS*, pages 61–68, 2016.

20 Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. Experimental security analysis of a modern automobile. In *IEEE S&P*, pages 447–462, 2010.

21 Man Lin, Li Xu, Laurence T. Yang, Xiao Qin, Nenggan Zheng, Zhaohui Wu, and Meikang Qiu. Static security optimization for real-time systems. *IEEE Trans. on Indust. Info.*, 5(1):22–37, 2009.

22 Chung Laung Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *JACM*, 20(1):46–61, 1973.

23 Xue Liu, Hui Ding, Kihwal Lee, Qixin Wang, and Lui Sha. ORTEGA: An efficient and flexible software fault tolerance architecture for real-time control systems. In *IEEE ECRTS*, pages 125–134, 2008.

24 Daniel Lo, Mohamed Ismail, Tao Chen, and G. Edward Suh. Slack-aware opportunistic monitoring for real-time systems. In *IEEE RTAS*, pages 203–214, 2014.

25 Sibin Mohan. Worst-case execution time analysis of security policies for deeply embedded real-time systems. *ACM SIGBED Review*, 5(1):8, 2008.

26 Sibin Mohan, Man-Ki Yoon, Rodolfo Pellizzoni, and Rakesh B. Bobba. Real-time systems security through scheduler constraints. In *IEEE ECRTS*, pages 129–140, 2014.

27 Sibin Mohan, Man-Ki Yoon, Rodolfo Pellizzoni, and Rakesh B. Bobba. Integrating security constraints into fixed priority real-time schedulers. *RTS Journal*, 52(5):644–674, 2016. doi:10.1007/s11241-016-9252-5.

28 A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical report, Massachusetts Institute of Technology, 1983.

29 Almir Mutapcic, Kwangmoo Koh, Seungjean Kim, Lieven Vandenberghe, and Stephen Boyd. GGPLAB: a simple Matlab toolbox for geometric programming, 2006. URL: https://stanford.edu/~boyd/ggplab/.

30 Rodolfo Pellizzoni, Neda Paryab, Man-Ki Yoon, Stanley Bak, Sibin Mohan, and Rakesh B. Bobba. A generalized model for preventing information leakage in hard real-time systems. In *IEEE RTAS*, pages 271–282, 2015.

31 Saowanee Saewong, Ragunathan (Raj) Rajkumar, John P. Lehoczky, and Mark H. Klein. Analysis of hierarchical fixed-priority scheduling. In *IEEE ECRTS*, pages 173–181, 2002.

32 Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. on Comp.*, 39(9):1175–1185, 1990.

33 Shellcode on ARM architecture. http://shell-storm.org/shellcode.

34 Daniel P. Shepard, Jahshan A. Bhatti, Todd E. Humphreys, and Aaron A. Fansler. Evaluation of smart grid and civilian UAV vulnerability to GPS spoofing attacks. In *Proc. of the ION GNSS Meeting*, volume 3, 2012.

35 Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *IEEE RTSS*, pages 2–13, 2003.

36 Open Source Tripwire. https://github.com/Tripwire/tripwire-open-source.

37 UAV Control Codes. https://github.com/Khan-drone/flight-control.

38 Xenomai – Real-time framework for Linux. https://xenomai.org.

39 Tao Xie and Xiao Qin. Improving security for periodic tasks in embedded systems through scheduling. *ACM TECS*, 6(3):20, 2007.

40 Man-Ki Yoon, Jung-Eun Kim, Richard Bradford, and Lui Sha. Holistic design parameter optimization of multiple periodic resources in hierarchical scheduling. In *DATE*, pages 1313–1318, 2013.

41 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 *IEEE RTAS*, pages 1–12, 2016.

42 Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Jung-Eun Kim, and Lui Sha. SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems. In *IEEE RTAS*, pages 21–32, 2013.

WCET-Driven Dynamic Data Scratchpad Management With Compiler-Directed Prefetching*

Muhammad R. Soliman[1] and Rodolfo Pellizzoni[2]

1 **University of Waterloo, Waterloo, ON, Canada**
mrefaat@uwaterloo.ca
2 **University of Waterloo, Waterloo, ON, Canada**
rpellizz@uwaterloo.ca

Abstract

In recent years, the real-time community has produced a variety of approaches targeted at managing on-chip memory (scratchpads and caches) in a predictable way. However, to obtain safe WCET bounds, such techniques generally assume that the processor is stalled while waiting to reload the content of the on-chip memory; hence, they are less effective at hiding main memory latency compared to speculation-based techniques, such as hardware prefetching, that are largely used in general-purpose systems. In this work, we introduce a novel compiler-directed prefetching scheme for scratchpad memory that effectively hides the latency of main memory accesses by overlapping data transfers with the program execution. We implement and test an automated program compilation and optimization flow within the LLVM framework, and we show how to obtain improved WCET bounds through static analysis.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

Keywords and phrases scratchpad, LLVM, prefetching, real-time, genetic algorithm

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.24

1 Introduction

The performance of computer programs can be significantly affected by main memory latency, which has largely remained similar in recent years [18]. As a consequence, cache prefetching has been extensively researched in the architecture community [16]. Prefetching techniques incorporate hardware and/or software to hide cache miss latency by attempting to load cache lines from main memory before they are accessed. The essence of these techniques is speculation of the data locality and the cache behavior, which makes them unsuitable to provide Worst-Case Execution Time (WCET) guarantees for real-time programs.

In the context of real-time systems, there has been significant attention to the management of on-chip memory in recent times. In particular, a large number of allocation schemes for scratchpad memories have been proposed in the literature; compared to caches, ScratchPad Memory (SPM) requires an explicit management of transfers from/to main memory. We note that cache memories can also be managed in a predictable manner similar to SPM, for example employing cache locking [9]. These techniques allow the derivation of tighter WCET

* This work was supported in part by NSERC DG 402369-2011 and CMC Microsystems. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.

bounds by statically determining if a memory instruction will access the on-chip memory or the main memory. However, they do not solve the fundamental memory latency problem, because they generally assume that the core is stalled while the content of on-chip memory is reloaded.

To address such issue, in this paper we present a novel compiler-directed prefetching scheme that optimizes the allocation of program data in on-chip memory with the objective to minimize the WCET. Our method relies on a Direct Memory Access (DMA) controller to move data between on-chip memory and main memory. Compared to related work, we do not stall the program while transferring data; instead, we rely on static program analysis to determine when the data is used in the program, and we prefetch it into the on-chip memory ahead of its use so that the time required for the DMA transfer can be *overlapped* with the program execution. As we show in our evaluation, for certain benchmarks our solution allows to efficiently reduce the stall time due to memory latency. More in details, we provide the following contributions:

- We describe an allocation mechanism for SPM that manages DMA transfers with minimum added overhead to the program. For simplicity and as a proof of concept, we implement our mechanism using a dedicated SPM controller, but we argue that a similar scheme could be supported by other platforms with the required DMA functionality. To statically determine which accesses target the SPM, we introduce a program representation and allocation constraints based on refined code regions.
- We develop an allocation algorithm for data SPM that takes into account the overlap between DMA transfers and program execution.
- We show how to model the proposed mechanism in the context of static WCET analysis using a standard data-flow approach for processor analysis.
- We fully implement all required code analysis, optimization and transformation steps within the LLVM compiler framework [12], and test it on a collection of benchmarks. Outside of loop bound annotations, our prototype is able to automatically compile and optimize the program without any programmer intervention.

The rest of the paper is organized as follows. We recap related work in Section 2. We then introduce a motivating example in Section 3. We detail the region-based program representation in Section 4, and our proposed allocation mechanism in Section 5. Section 6 discusses the allocation algorithm, and Section 7 introduces the WCET abstraction for our prefetch mechanism. Finally, we present the compiler implementation in Section 8 and experimental results in Section 9, and provide concluding remarks in Section 10.

2 Related Work

SPM management has been widely explored in the literature, both for code and data allocation. We focus on data SPM as it drew more attention in the literature due to the challenges connected to data usage analysis and optimization. Many approaches target improving the average case performance [17, 24, 2, 29, 5, 7]. Other mechanisms optimize the allocation for WCET in real-time systems [21, 26, 11, 6]. In general, management techniques are divided between static or dynamic. Static methods partition the data memory between SPM and main memory with fixed allocation of the SPM at compile-time [2, 21]. On the other hand, dynamic methods adapt to the changing working data set of the program by moving objects between SPM and main memory during run-time [17, 24, 6, 7, 26, 29]. Since our proposed scheme allows us to more efficiently hide the cost of data transfers, we focus on dynamic allocation.

The closest related work in the scope of dynamic methods for data SPM are [29, 5, 8], which apply prefetching through DMA. In [29], the authors proposed a data pipelining technique for SPM that utilizes DMA to achieve data parallelization for multiple iterations of a loop based on the iteration access patterns of arrays. The work in [5] proposes a general prefetching scheme for on-chip memory. It exploits the usage of DMA priorities and pipelining to prefetch arrays with high reuse to minimize the energy and maximize average performance. In [8], the authors add a dedicated DMA engine to the processor to control the DMA transfers using a job queue, similarly to the mechanism proposed in our work. They also provide high level functions to manage the DMA. However, no optimized allocation scheme is discussed. Furthermore, all three discussed works target the average case rather than the worst case.

In the context of real-time systems, the closest line of work is the PRedictable Execution Model (PREM) [19, 22, 3]. Under PREM, the data and code of a task are fetched into on-chip memory before execution, preferably using DMA. A variety of co-scheduling schemes (see for example [15, 1]) have been proposed to avoid stalling the processor by scheduling the DMA operations for one task with the execution of another task on the same core. However, we argue that such approaches suffer from three main limitations, that we seek to lift in this work.

1. Statically loading all data and code before the beginning of the program severely limits the flexibility and precision of the allocation.
2. DMA transfers cannot be overlapped with the execution of the same task, only other tasks. This makes the proposed approaches less suitable for many-core systems, where it might be preferable to execute a single task/thread on each core.
3. With the exception of [14], the proposed approaches assume manual code modification, which we find unrealistic in practice. An automated compiler tool-chain is described in [14], but since it relies on profiling, it cannot guarantee WCET bounds.

3 Motivating Example

In this section, we present an example that shows the benefit of data prefetching in SPM-based systems. Given a set of data *objects* used by a program, the general SPM allocation problem is to determine which subset of objects should be allocated in SPM to minimize the WCET of the program. Since the latency of accessing an object in the SPM is less than in main memory, we can compute the *benefit* in terms of WCET reduction for each object allocated in the SPM. We model the program's execution with a Control Flow Graph (CFG) where nodes represent basic blocks, *i.e.*, straight-line pieces of code.

In particular, Figure 1 shows the CFG of a program where object x is read/modified in basic blocks BB_2 and BB_4 and object y is read in BB_4. Note that BB_2 and BB_4 are loops, since they include back-edges (*i.e.*, the program execution can jump back to the beginning of the block); hence, x and y can be accessed many times. Assume that the SPM can only fit x or y. A static SPM allocation approach will choose to allocate either x or y for the whole program execution. A dynamic SPM allocation approach will try to maximize the benefit by possibly evicting one of the two objects to fit the other during the program execution.

Let the benefit of accessing x from the SPM instead of the main memory be 100 cycles for BB_2 and 10 cycles for BB_4. Similarly, the benefit of accessing y from the SPM in BB_4 is 70 cycles. Let the cost to transfer x from main memory to the SPM or vice-versa be 20 cycles, and the cost for y be 40 cycles. Then, for static allocation, the total benefit of allocating x is $100 + 10 = 110$ cycles and the cost is 2*20 cycles (fetch x from memory to

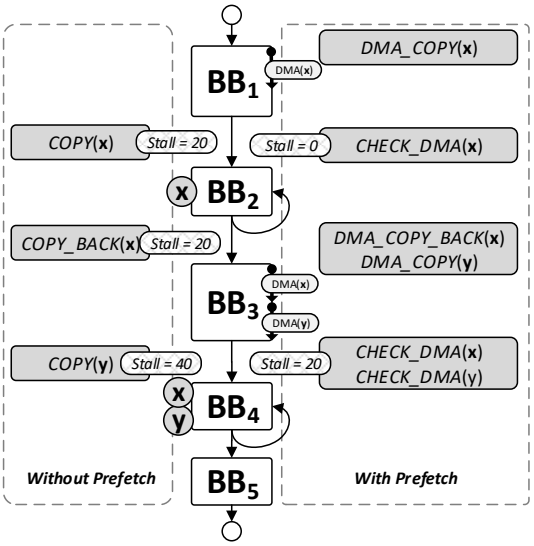


Figure 1 Motivating Example.

SPM at the beginning of the program and write it back from SPM to main memory at the end). Similarly, the benefit for allocating y is 70 cycles and the cost is 40 cycles (fetch only as y is not modified, so there is no need to write it back to main memory). The optimal allocation would choose x as it has a net benefit of 70 cycles versus 30 cycles for y.

In previous approaches that adopt dynamic allocation, the program execution has to be interrupted to transfer objects either using a software loop or a DMA unit. We represent this case in the *without prefetch* box in Figure 1. In the example, x is fetched before BB_2 and written back after BB_2 to empty the SPM for y. Then, y is fetched before BB_4. Since x is allocated in the SPM for BB_2 and y is allocated for BB_4, this results in a total benefit of $100 + 70 = 170$. The program will stall before BB_2 to fetch x, after BB_2 to write-back x, and before BB_4 to fetch y resulting in total cost of $20 + 20 + 40 = 80$ cycles. The net benefit is $170 - 80 = 90$ cycles, which is 20 cycles better than the static allocation.

However, if memory transfers can be parallelized with the execution of the program, we next show that we can exploit the SPM more efficiently. We illustrate the prefetching sequence in the *with prefetch* box in Figure 1. Let us assume that the amount of execution time that can be overlapped with DMA transfers is 30 and 40 cycles for BB_1 and BB_3, respectively. We start prefetching x before BB_1 by configuring the DMA to copy x from main memory to SPM. Then, we poll the DMA before BB_2 where x is first used to ensure that the transfer has finished. Since transferring x requires less cycles than the maximum overlap for BB_1 (20 versus 30), the prefetch operation for x finishes in parallel with the execution of BB_1; hence, there is no need to stall the program before x can be accessed from the SPM in BB_2. Before BB_3, we first write-back x so that we have enough space in the SPM to then prefetch y. We propose to schedule both transfers back-to-back, *e.g.* using a scatter-gather DMA, in parallel with the execution of BB_3. Since the amount of overlap for BB_3 is 40, the write-back for x completes after 20 cycles, leaving 20 additional cycles of overlap for the prefetch of y. Hence, by the time BB_4 is reached, the CPU stalls for $40 - 20 = 20$ cycles to complete prefetching y before using it in BB_4. For the described prefetching approach, the benefit is the same as the dynamic allocation. However, the cost is lower as the CPU only stalls for 20 cycles. The net benefit is $170 - 20 = 150$ cycles, compared to 90 cycles without prefetching.

4 Region-Based Program Representation

The motivating example shows that the cost of copying objects between main memory and SPM can be reduced by overlapping DMA transfers with program execution. However, to achieve a positive benefit, we also need to predict whether any given memory access targets the SPM rather than main memory. In general, programs contain branches and function calls, making such determination possibly dependent on the execution path. To produce tight WCET bounds, a fundamental goal of our approach is to *statically determine which memory accesses are in the SPM regardless of the flow through the program.* To achieve this objective, in this section we consider a program representation based on code *regions* [10] and we add constraints on how objects can be allocated in the SPM based on regions.

We consider a program composed of multiple functions. Let $G_f = (N_f, E_f)$ be the CFG for function f, where N_f is the set of nodes representing basic blocks and E_f is the set of edges. A *Single Entry Single Exit (SESE) region* is a sub-graph of the CFG that is connected to the remaining nodes of the CFG with only two edges, an entry edge and an exit edge. A region is called *canonical* if there is no set of regions that can be combined to construct it. Any two canonical regions are either disjoint or completely nested. The canonical regions of a program can be organized in a *region tree* such that the *parent* of a region is the closest containing region, and children of a region are all the regions immediately contained within it. Two regions are *sequentially composed* if the exit of one region is the entry of the following region. Note that a basic block with multiple entry/exit edges does not construct a region by itself.

Figure 2a shows an example CFG and its canonical regions. The corresponding region tree is shown in Figure 2b. In this example, region r_1 is the parent of regions r_2, r_3 and r_4. Regions r_2 and r_3 are sequentially composed; this is represented by a solid-line box in the figure.

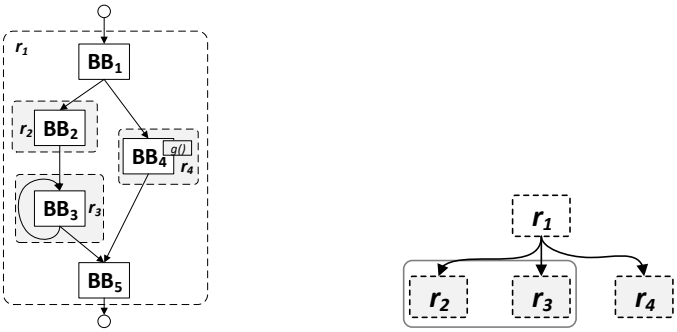
In the rest of the paper, we use the term *allocation* to refer to the act of reserving a space for an object in the SPM at a given address during the execution of the program code. In our solution, we restrict the allocation of objects on a per-region basis: space for an object is reserved upon entering a region, and the object is then evicted from the SPM upon exiting the same region. This guarantees that the object is available in the SPM independently of which path the program takes through the region; as an example, if we allocate object x in r_1, then we statically know that any reference to x in BB_5 will access the SPM independently of whether the program flows through $BB_2 - BB_3$ or BB_4, or of how many iterations of the loop in BB_3 are taken.

Unfortunately, the proposed region-based allocation has two limitations: (1) we cannot allocate an object in BB_1 only, because BB_1 is not a region; (2) in the example, BB_4 performs a call to another function $g()$. Since the entirety of BB_4 is a region, we cannot decide to allocate an object only for the call to $g()$, or only for the rest of the code of BB_4. To address these limitations, we propose to construct a refined region tree that allows a finer granularity of allocation.

To obtain the refined regions, we first construct a modified graph $\bar{G}_f = (\bar{N}_f, \bar{E}_f)$ from G_f, where $\bar{N}_f$ is the set of basic block nodes, call nodes and merge/split nodes and $\bar{E}_f$ is the set of edges such that:

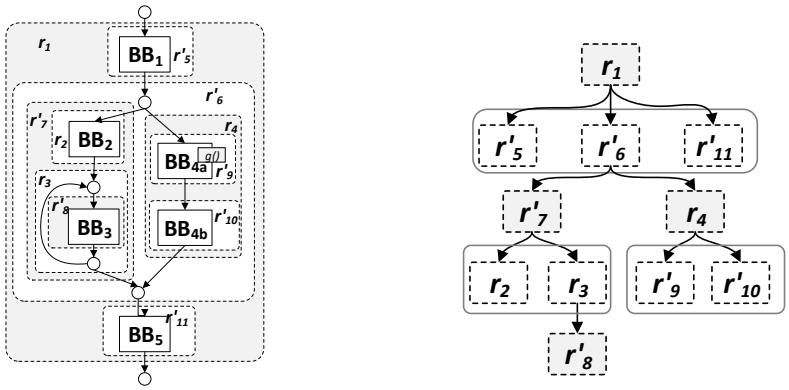
- Each call to a function in G_f is split into a separate *call node*.
- A *merge/split node* is inserted before/after a basic block/call node with multiple entry/exit edges.

Note that after the transformation, every node in $\bar{G}_f$ that is not a merge/split node has a single entry and a single exit; hence, it is a region. We denote a region that consists of a



(a) Program CFG **(b)** Region tree

Figure 2 Program CFG G_f and region tree.



(a) Refined program CFG **(b)** Refined region tree

Figure 3 Refined program CFG $\bar{G}_f$ and region tree.

sequence of sequentially composed regions as a *sequential region*. A sequential region is not canonical as it is constructed by combining other regions. Finally, we construct the refined region tree by considering both canonical regions and maximal sequential regions, *i.e.*, any sequential region that encompasses a maximal sequence of sequentially composed regions. It is proved in [25] that adding maximal sequential regions to the tree still results in a unique region tree.

Figure 3 shows the refined CFG and region tree for the example in Figure 2. We added merge points before BB_3 and BB_5, and split points after BB_1 and BB_3. Assuming that function $g()$ is called at the beginning of BB_4, we split BB_4 to a call node BB_{4a} that contains the function call and a basic block BB_{4b} for the rest of the instructions in BB_4. In the refined region tree in Figure 3b, regions r_1 to r_4 are the same as in the original region tree, while regions r'_5 to r'_{11} are added as a result of the refinement process. Regions r_1, r'_7 and r_4 are sequential regions. We refer to r'_9 as a *call region* as it contains the call node BB_{4a}. Finally, we use the term *trivial region* to denote any leaf of the refined region tree (r'_5, r'_{11}, r_2, r'_8, r'_9 and r'_{10} in the example); note that by definition, each trivial region must comprise either a single basic block or a single call node, *i.e.*, trivial regions represent code segments in the program. Since allocations are based on regions, for simplicity we will omit individual nodes when representing CFGs and instead draw regions.

5 Allocation Mechanism

We now present our proposed allocation mechanism in detail. In the rest of the paper, we assume the following:

- We focus solely on the allocation of data SPM, as it is generally more challenging. We assume a separate instruction SPM that is large enough to fit the code.
- The allocation is object-based, meaning that we do not allow allocation of parts of an object. Data pipelining techniques for loops and field-based allocation for data structures could further improve the allocation, especially for small sizes of SPM. We keep this possible expansion to future work.
- We assume that the target program does not use recursion or function pointers and that local objects have fixed or bounded sizes. We argue that these assumptions conform with standard conventions for real-time applications.
- We assume that all loops in the program are bounded. The bounds can be derived using compiler analysis, annotations or profiling.
- We consider global and stack objects only for allocation. We rely on pointer analysis to determine the references of the load/store instructions. For stack objects, we convert large local objects to global objects before allocation as we discuss in Section 8. While our method can be extended to handle dynamic pointers as shown in technical report [20], we do not discuss it here due to space constraints.
- For simplicity, we consider a system comprising a single core running one program with no preemption. However, the proposed method could be extended to a multicore system supporting a predictable arbitration for main memory as long as each core is provided with private or partitioned SPM.

As discussed in the motivating example, to efficiently manage the dynamic allocation of multiple objects we require a DMA unit capable of queuing multiple operations. In general, many commercial DMA controllers with scatter-gather functionality support such requirement, albeit the complexity of managing the DMA controller in software could increase with the number of transfers. As a proof of concept, we based our implementation on a dedicated unit, which we call the *SPM controller*; we reserve implementation on a COTS platform as future work [1].

Our proposed mechanism works by inserting *allocation commands* in the code of the program, which are then executed by the SPM controller. The process of allocating an object starts with reserving space in the SPM and prefetching the object from main memory if necessary (`ALLOC` command). Then, once the prefetch operation is complete, the SPM address is read and passed to the memory references that access the object (`GETADDR` command). Finally, the object is evicted from the SPM and written back to main memory if necessary (`DEALLOC` command). As discussed in Section 4, we restrict object allocation based on regions; hence, the `ALLOC` command is always inserted at the beginning of a region, and the corresponding `DEALLOC` command at the end of the same region. In the rest of the section, we first detail the operation of the SPM controller, followed by the semantic of the allocation commands. Finally, we provide a comprehensive allocation example.

[1] For example, the Freescale MPC5777M SoC used in previous work [22] includes both SPM memory and a dedicated I/O processor that could be used to implement the described management functionalities.

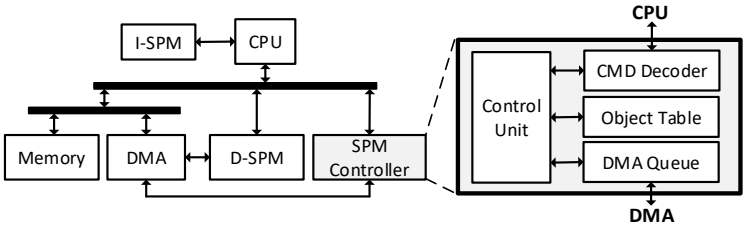


Figure 4 SPM-based System.

5.1 SPM controller

Figure 4 shows the proposed SPM controller and its connections to an SPM-based system. There is a separate instruction SPM (I-SPM) that is assumed to fit the code of the program. The data SPM (D-SPM) is managed by the SPM controller. Since the processor must be able to access the SPM directly, the SPM is assigned an address range distinct from main memory. The SPM controller is also a memory mapped unit, since the CPU sends allocation commands to the SPM controller by reading/writing to its address range. Note that we assume physical memory addresses in this implementation, *i.e.*, no virtual address mapping is used. The system incorporates a DMA unit for memory transfers. The D-SPM is assumed to be dual-ports, which means that access to the SPM by the CPU and transferring data between SPM and main memory using DMA can occur simultaneously. The proposed allocation method and WCET analysis can be applied for single-port SPM, but this will offer less opportunity to overlap the memory transfers. The DMA is connected to a shared bus with the main memory. This bus can be used by either the CPU or the DMA. To efficiently support the parallelization of memory transfers with the execution time, the DMA is designed to work in transparent mode: it transfers an object only when the CPU is not using the main memory. Whenever the CPU requests the memory bus, the DMA yields to the request and stalls any ongoing transfer until the memory bus is released.

The SPM controller consists of *command decoder*, *object table*, *DMA queue* and *control unit* as shown in Figure 4. As discussed, allocation commands are encoded as load/store instructions to the SPM controller. So, the command decoder reads the address and the data of the memory operation and decodes them into one of the allocation commands; the control unit then executes the command using the object table and the DMA queue.

The object table tracks the state of the program objects. Note that only the subset of program objects that can be allocated in the SPM are tracked by the object table. An object table with 32 entries is sufficient in our tests. However, if the number of objects in the program exceeds the number of entries in the object table, the object table can hold only the allocated objects at each program point. An entry in the object table contains the main memory address, the size of the object, the SPM address and allocation flags that reflect the status of the object:

`A`	(A)llocated in the SPM
`PF_OP`	(P)re(F)etching (OP)eration has been scheduled
`WB_OP`	(W)rite-(B)ack (OP)eration has been scheduled
`WB`	(W)rite-(B)ack the object when de-allocated if it is used
`U`	(U)sed in the SPM
`USERS`	number of current users of the object

The `USERS` field records the number of allocations that have issued an `ALLOC` command for the object and are still using the object in the SPM, *i.e.*, the corresponding `DEALLOC`

has not been reached. It is incremented by `ALLOC` and decremented by `DEALLOC`. We show an example for the usage of this field in Section 5.3. The DMA is configured with source address, size and destination address extracted from an entry in the object table. DMA operations are added to the DMA queue that allows scheduling multiple memory transfers and executing them in FIFO order.

5.2 Allocation Commands

`ALLOC` command reserves the space in the SPM and schedules a DMA transfer if necessary. The command has the following syntax: `ALLOC.XX (TBL_IDX, MEM_ADDR)`, where `TBL_IDX` is the table index for the object and `MEM_ADDR` is the allocation address in the SPM. There are four versions of `ALLOC.XX` command according to the directives `XX`: `ALLOC`, `ALLOC.P`, `ALLOC.W`, `ALLOC.PW`. The `P` directive directs the controller to prefetch the object from main memory. The SPM controller will schedule a prefetch transfer for the object and set `PF_OP` flag in the object entry. If the `P` directive is not used, the object is allocated directly and flag `A` is set. Otherwise, flag `A` is set once the prefetch transfer completes. The `W` directive sets the `WB` flag in the object entry which directs the controller to copy back the object to the main memory when de-allocated. The `P` directive is used in two cases: (1) if, during the execution of the region where the object is allocated, the current value of the object is read or (2) the object is partially modified, *e.g.* writing some elements of an array. The `W` directive is used if the object is modified, so that the main memory is updated with the new values after de-allocating the object. Note that for local objects defined in a function, there is no need to prefetch the object before its first use in the function or write-back the object after its last use in the function.

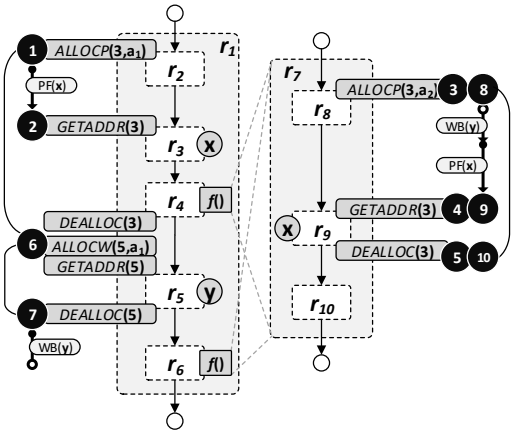
`DEALLOC` command de-allocates the object in table index `TBL_IDX` from the SPM: `DEALLOC (TBL_IDX)`. If the `WB` and `U` flags are set in the object entry at `TBL_IDX`, the controller will schedule a write-back transfer, set `WB_OP` flag and reset `A` flag. Otherwise, the object will be de-allocated by simply resetting `A` flag.

`GETADDR` command returns the current address of the object associated with entry `TBL_IDX` in the object table: `GETADDR (TBL_IDX)`. If `PF_OP` or `WB_OP` flag is set in the table index `TBL_IDX`, the controller stalls until the DMA completes transferring the object. If no transfer is scheduled or after the transfer finishes, the controller returns the SPM address if `A` flag is set and the main memory address otherwise. `GETADDR` command is only added before the first use of the object after an allocation/de-allocation. The address returned by the command is then applied for all the next uses until another allocation/de-allocation occurs. This process is compiler-automated and does not require per-access address translation from main memory to SPM addresses, as in related work [26]; hence, we do not add extra overhead to the critical path of the processor.

If a prefetch transfer has been scheduled and a `DEALLOC` command is issued for the object to be prefetched, the transfer is canceled as the object is not needed anymore. Also, if a write-back transfer has been scheduled for an object and it was followed by `ALLOC.XX` for the same object, the transfer is canceled if the object is allocated to the same SPM address, otherwise the transfer is not canceled. This is particularly important for allocations within loops, when the object can be allocated to the same address over multiple iterations.

5.3 Example

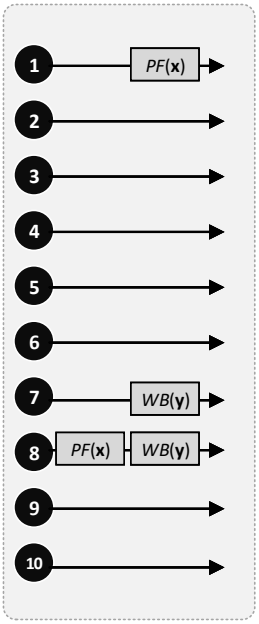
Figure 5 shows an allocation example for two objects x and y corresponding to entries `3` and `5` in the object table. Figure 5a shows the CFG of two functions where r_1-r_{10} represent



(a) Allocation Process

	IDX	MM_ADDR	SIZE	SPM_ADDR	A	PF_OP	WB_OP	WB	U	USERS
1	3	0x2000	80	a_1	0	1	0	0	0	1
	5	0x1000	40		0	0	0	0	0	0
2	3	0x2000	80	a_1	1	0	0	0	1	1
	5	0x1000	40		0	0	0	0	0	0
3	3	0x2000	80	a_1	1	0	0	0	1	2
	5	0x1000	40		0	0	0	0	0	0
4	3	0x2000	80	a_1	1	0	0	0	1	2
	5	0x1000	40		0	0	0	0	0	0
5	3	0x2000	80	a_1	1	0	0	0	1	1
	5	0x1000	40		0	0	0	0	0	0
6	3	0x2000	80		0	0	0	0	0	0
	5	0x1000	40	a_1	1	0	0	1	1	1
7	3	0x2000	80	a_2	0	0	0	0	0	0
	5	0x1000	40	a_1	0	0	1	0	0	0
8	3	0x2000	80	a_2	0	1	0	0	0	1
	5	0x1000	40	a_1	0	0	1	0	0	0
9	3	0x2000	80	a_2	1	0	0	0	1	1
	5	0x1000	40		0	0	0	0	0	0
10	3	0x2000	80	a_2	0	0	0	0	0	0
	5	0x1000	40		0	0	0	0	0	0

(b) Object Table: Entries (`3` , `5`)



(c) DMA Queue

Figure 5 Allocation Example.

regions. x is read in r_3 and r_9, and y is written in r_5. Note that function f, comprising regions r_7 to r_{10}, is called from two different call regions, r_4 and r_6. In the example, we assume that x is allocated at address a_1 in the SPM in sequentially composed regions r_2, r_3 and r_4. Then, it is evicted to empty enough space for y to be allocated in r_5. However, x is also allocated inside function f at a different address a_2. Note that f could also be called from other, non represented call regions in the program, and assigning address a_2 to x might be required to fit x in the SPM together with other objects allocated in the unrepresented call regions.

We use program points ❶ to ❿ to follow the allocation process. Entries `3` and `5` of the object table are traced in Figure 5b; and the DMA queue is shown in Figure 5c. At ❶, x is allocated to address a_1 with `P` directive. In the object table, `PF_OP` is set to indicate x is being prefetched and `USERS` is incremented. A prefetch transfer $PF(x)$ is scheduled in

the DMA queue. At ❷, `GETADDR (3)` checks entry `3` for the address; if $PF(x)$ has not finished at this point, the CPU stalls until the prefetch finishes; then x is allocated, `PF_OP` is reset, and `A` is set; and the CPU continues execution. Also, `U` is set to mark x as used in the SPM. In r_4, function f is called. At ❸, an allocation of x to a_2 is issued; however, x is already in the SPM at address a_1. So, no new allocation at a_2 is performed, and `USERS` is incremented in entry `3` to indicate that two `ALLOC` commands (users) have been executed for x. `GETADDR (3)` at ❹ returns a_1. When x is deallocated at ❺, `USERS` is decremented in entry `3`. However, x is not evicted as there is another user for it. When x is deallocated at ❻, x is evicted as this is the last user of x in the SPM. There is no need to write-back x as `WB` was not set. y is also allocated to a_1 at ❻ with `W` directive. So, no prefetch is scheduled and flags `A` and `WB` are set in entry `5`. Before y is used in r_5, `GETADDR (5)` is executed which sets `U` flag in entry `5` and returns address a_1. At ❼, y is deallocated and a write-back transfer is scheduled as both `WB` and `U` flags are set. f is called again in r_6. At ❽, a prefetch is issued for x to a_2. The DMA queue will have both $WB(y)$ and $PF(x)$ scheduled. At ❾, $WB(y)$ has finished while $PF(x)$ is still ongoing. The execution is stalled until the prefetch for x is complete, then `GETADDR (3)` returns a_2. Finally at ❿, x is deallocated.

An essential observation is that the state of the SPM and the sequence of DMA operations in function f depend on which region calls f: if f is called from r_4, then x is already available in SPM at address a_1, and the allocation to a_2 is not used. If instead f is called from r_6, x is allocated to a_2 and the object must be prefetched from main memory. Therefore, let σ be the *context* under which a region executes, *i.e.*, the sequence of call regions starting from the main function; note that since the main function of the program is not called by any other function, the only valid context for regions in the main is $\sigma = \emptyset$. We denote the execution of a region r_n in a context σ as r_n^σ, which we call a *region-context* pair. Then, allocation decisions, which involve adding allocation commands in the code, must be based on regions, but the state of the SPM and DMA operations, which are needed for WCET estimation, depend on region-context pairs. Intuitively, this is equivalent to considering multiple copies of each region r_n, one for each context in which r_n can execute.

6 Allocation Problem

We now discuss how to determine a set of allocations for the entire program with the objective to minimize the WCET of the program. For the remaining of the section, we use S_{SPM} to denote the size of the SPM. $V = \{v_1, \ldots, v_j, \ldots\}$ is the set of allocatable objects, where $S(v_j)$ denotes the size of object v_j. We let $R = \{r_1, \ldots, r_n, \ldots\}$ be the set of program regions across all functions. Without loss of generality, we assume that region indexes are topologically ordered, so that each parent region has smaller index than its children, each call region has smaller index than the regions in the called function, and sequentially composed regions have sequential indexes; this is also the order used in Figure 5. Note that such topological order must exist since the refined region tree for each function is unique, and furthermore the call graph has no loops due to the absence of recursion. Finally, to define the relation between region-context pairs we introduce a parent function $\wp(r_n^\sigma)$ for a region-context r_n^σ in function f as follows: if r_n is the root region of the refined region tree for f, then $\wp(r_n^\sigma) = r_m^{\sigma'}$, where $r_m^{\sigma'}$ is the region-context that calls f in context σ. Otherwise, $\wp(r_n^\sigma) = r_m^\sigma$, where r_m is the parent region of r_n. As an example based on Figure 5, assume that r_4 executes in context σ. Then when r_7 is called from r_4, r_7 executes in context $\sigma \cup r_4$. We further have $\wp(r_7^{\sigma \cup r_4}) = r_4^\sigma$, while for example $\wp(r_8^{\sigma \cup r_4}) = r_7^{\sigma \cup r_4}$.

We begin by formalizing the conditions under which a set of allocations is feasible as a satisfiability problem. This is similar to a multiple knapsack problem where regions are

knapsacks (available space in SPM), except that we add additional constraints to model the relation between regions. Remember that to allocate an object v_j in a region r_n, we have to assign an address in the SPM to the object. Hence, an allocation solution is represented by an assignment to the following decision variables over all regions $r_n \in R$ and all objects $v_j \in V$:

$$alloc_{r_n}^{v_j} = \begin{cases} 1, & \text{if } v_j \text{ is allocated in } r_n \\ 0, & \text{otherwise} \end{cases}$$

$assign_{r_n}^{v_j}$ = address assigned to v_j in r_n

An allocation solution is feasible if the allocated objects fit in the SPM at any possible program point. As discussed in Section 5.3, the state of the SPM depends on the context under which a region is executed. Hence, we introduce new helper variables to define the availability of an object v_j in a region-context r_n^σ:

$$avail_{r_n^\sigma}^{v_j} = \begin{cases} 1, & \text{if } v_j \text{ is available in SPM for execution of } r_n^\sigma \\ 0, & \text{otherwise} \end{cases}$$

$address_{r_n^\sigma}^{v_j}$ = address of v_j in the SPM during execution of r_n^σ

We can determine the value of the helper variables based on the allocation:

$$\forall v_j, r_n^\sigma : alloc_{r_n}^{v_j} \vee avail_{\wp(r_n^\sigma)}^{v_j} \Leftrightarrow avail_{r_n^\sigma}^{v_j}. \tag{1}$$

Equation 1 simply states that v_j is available in the SPM during the execution of r_n^σ if either v_j is allocated in r_n, or if v_j was already available in the SPM during the execution of the parent region-context pair.

$$\forall v_j, r_n^\sigma : avail_{\wp(r_n^\sigma)}^{v_j} \Rightarrow address_{r_n^\sigma}^{v_j} = address_{\wp(r_n^\sigma)}^{v_j}. \tag{2}$$

$$\forall v_j, r_n^\sigma : \neg avail_{\wp(r_n^\sigma)}^{v_j} \wedge alloc_{r_n}^{v_j} \Rightarrow address_{r_n^\sigma}^{v_j} = assign_{r_n}^{v_j}. \tag{3}$$

Equations 2, 3 specify the address in the SPM. If the object was already available in the parent region-context, then the address is the same. Otherwise, if the object is allocated in r_n, then the address is the one assigned by the allocation.

Example: refer to the example in Figure 5, where x is allocated with assigned address a_1 in r_2, r_3 and r_4 and with address a_2 in r_8 and r_9. For context $\sigma \cup r_6$, we have $avail_{r_7^{\sigma\cup r_6}}^{x} = 0$, since x is not available in r_6^σ, the parent of $r_7^{\sigma\cup r_6}$. Hence, we also have $address_{r_8^{\sigma\cup r_6}}^{x} = a_2$. However, for context $\sigma \cup r_4$ we obtain $avail_{r_7^{\sigma\cup r_6}}^{x} = 1$ and $address_{r_7^{\sigma\cup r_6}}^{x} = a_1$, hence $address_{r_8^{\sigma\cup r_6}}^{x} = a_1$.

Finally, given the object availability and address for each region-context pair, we can express the feasibility conditions for the allocation problem such that the allocated objects fit within the SPM and concurrent allocated objects have non-overlapping address ranges.

$$\forall v_j, r_n^\sigma : avail_{r_n^\sigma}^{v_j} \Rightarrow address_{r_n^\sigma}^{v_j} + S(v_j) \leq S_{SPM}. \tag{4}$$

$$\forall v_j, v_k, r_n^\sigma, j \neq k : (avail_{r_n^\sigma}^{v_j} \wedge avail_{r_n^\sigma}^{v_k}) \Rightarrow (address_{r_n^\sigma}^{v_j} + S(v_j) \leq address_{r_n^\sigma}^{v_k}) \vee (address_{r_n^\sigma}^{v_k} + S(v_k) \leq address_{r_n^\sigma}^{v_j}). \tag{5}$$

Equation 4 states that if v_j is in the SPM during the execution of r_n^σ, then it must fit within the SPM size. Equation 5 states that if both v_j and v_k are in the SPM during the execution of r_n^σ, then their addresses must not overlap.

As long as Equations 4, 5 are satisfied for a given solution in all region-context pairs, all objects fit in the SPM; hence, the allocation problem can be feasibly implemented. To do so, we next discuss how to determine the list of commands (`ALLOC` / `DEALLOC` / `GETADDR`) that

must be added to each region. For a region r_n that is not sequentially composed, an `ALLOC` is inserted at the beginning of the region and a `DEALLOC` at the end of the region. In the case of sequential regions, to reduce the number of DMA operations, we note the following: if the same object v_j is allocated in two sequentially composed regions r_p and r_q with the same assigned address, then there is no need to `DEALLOC` v_j at the end of r_p and `ALLOC` it again at the beginning of r_q. Hence, we consider the maximal sequence of sequentially composed regions $r_p, \ldots, r_q$ such that for every region r_n in the sequence: $alloc_{r_n}^{v_j} = 1$ and the address $assign_{r_n}^{v_j}$ assigned to v_j is the same. We then add the `ALLOC` command at the beginning of r_p and the `DEALLOC` command at the end of r_q. The `P` and `W` flags of the `ALLOC` command are set as discussed in Section 5.2 based on the usage throughout the whole sequence.

▶ **Example.** *Refer to the example in Figure 5, where x is allocated in two regions in sequence (r_8 and r_9). `ALLOC` is inserted before r_8 and `DEALLOC` is inserted after r_9. `P` flag is set in `ALLOC` even though x is not used in r_8, but it is read in r_9. Similarly, W is not set as x is not modified in neither r_8 nor r_9.*

Finally, to compute the WCET for the program, we need to determine whether an `ALLOC` / `DEALLOC` command triggers a DMA operation; this again depends on the context σ in which a given region r_n is executed, as demonstrated by the example in Section 5.3. As in Equation 2, we know that the `ALLOC` will be canceled if v_j was already available in the parent region-context; hence, for a region r_n that performs an `ALLOC` on v_j and a context σ, the `ALLOC` generates a DMA prefetch on v_j only if both the `P` flag in the `ALLOC` is set and $avail_{\wp(r_n^\sigma)}^{v_j} = 0$ (similarly for `DEALLOC`, a DMA operation is generated if the `W` flag is set and $avail_{\wp(r_n^\sigma)}^{v_j} = 0$).

6.1 WCET Optimization

For a given allocation solution $\{alloc_{r_n}^{v_j}, assign_{r_n}^{v_j} | \forall v_j, r_n\}$, the described procedure determines the set of objects available in the SPM and the set of DMA operations for each region-context r_n^σ. Assuming that bounds on the time required for SPM and main memory accesses are known, this allows us to determine the benefit (WCET reduction) for every trivial region in context σ, as well as the length of DMA operations. For a dynamic allocation approach without prefetch, the length of DMA operations could simply be summed to the execution time of the corresponding region, since DMA operations stall the core.

However, for our proposed prefetching approach, the cost of DMA operations depends on the overlap: since the DMA works in transparent mode, for a trivial region the maximum amount of overlap is equal to the execution time of its code minus the time that the CPU accesses main memory directly. Furthermore, since the length of DMA operations is generally longer than the execution of a trivial region, the total overlap depends on the program flow. Therefore, we compute the amount of overlap as part of an integrated WCET analysis, which we present in Section 7. We solve the allocation problem by adopting a heuristic approach that first searches for feasible allocation solutions, and then run the WCET analysis on feasible solutions to determine the best allocation; we discuss it next in Section 6.2.

Finally, we note that the proposed region-based allocation scheme is a generalization of the approaches used in related work on dynamic allocation. In [21], the authors applied a structured analysis to choose a set of variables for static allocation. They analyzed innermost loop as Directed Acyclic Graph (DAG) for worst case path and then collapsed the loop into a basic block to analyze the outer loop. The region tree representation captures these structures such as loops, conditional statements and functions as regions. The dynamic

Algorithm 1 Address Assignment

Input: region information, $\{alloc_{r_n}^{v_j} | \forall v_j, r_n\}$

1: **for all** region r_n by increasing index starting with r_1 **do**
2: $\quad end\_addr_{r_n} \leftarrow$ ASSIGN_ADDRESSES(r_n)

3: **function** ASSIGN_ADDRESSES(r_n)
4: $\quad end\_addr_{r_n} = \max_\sigma\{end\_addr_{\wp(r_n^\sigma)}\}$
5: $\quad$ **if** r_{n-1} is not sequentially composed with r_n **then**
6: $\quad\quad$ **for all** v_j such that $alloc_{r_n}^{v_j}$ **do**
7: $\quad\quad\quad assign_{r_n}^{v_j} \leftarrow end\_addr_{r_n}$
8: $\quad\quad\quad end\_addr_{r_n} \leftarrow end\_addr_{r_n} + S(v_j)$
9: $\quad$ **else**
10: $\quad\quad$ **for all** v_j such that $alloc_{r_n}^{v_j} \wedge alloc_{r_{n-1}}^{v_j}$ **do**
11: $\quad\quad\quad assign_{r_n}^{v_j} \leftarrow assign_{r_{n-1}}^{v_j}$
12: $\quad\quad$ **for all** v_j such that $alloc_{r_n}^{v_j} \wedge \neg alloc_{r_{n-1}}^{v_j}$ **do**
13: $\quad\quad\quad$ Compute $assign_{r_n}^{v_j}$ using best fit based on already assigned addresses
14: $\quad\quad end\_addr_{r_n} \leftarrow \max_{v_j \text{ s.t. } alloc_{r_n}^{v_j}}\{assign_{r_n}^{v_j} + S(v_j)\}$

allocation in [24] is based on program points around loops, if statements and functions which can be matched with an entry/exit of a region. In [6], Deverge *et al.* proposed a general graph representation that allows different granularities of allocation. The authors formulated the dynamic allocation problem based on the flow constraints which can also be applied to the region representation. All such approaches use heuristics to determine the overall program allocation. Hence, to allow a fair evaluation focused on the benefits of data prefetching, in Section 9 we compare our proposed scheme against a standard dynamic allocation approach with no overlap using the same region-based program representation and search heuristic.

6.2 Allocation Heuristic

The allocation heuristic adopts a genetic algorithm to search for near-optimal solutions to the allocation problem.

- **Chromosome Model:** The chromosome is a binary string where each bit represents one of the $alloc_{r_n}^{v_j}$ decision variables. Note that we do not represent the $assign_{r_n}^{v_j}$ decision variables in the chromosome; instead, we use a fast address assignment algorithm as part of the fitness function to find a feasible address assignment for a chromosome.
- **Fitness Function:** The fitness fit of a chromosome represents the improvement in the WCET of the program with this allocation if it is feasible. The fitness function first applies the address assignment algorithm to the chromosome. If the allocation is not feasible, the chromosome has $fit = 0$. Otherwise, we execute the WCET analysis after the program is transformed to insert the allocation commands; the fitness of the allocation is then assigned as $fit = WCET_{MM} - WCET_{alloc}$ where $WCET_{MM}$ is the WCET with all the objects in main memory and $WCET_{alloc}$ is the WCET for the analyzed solution.
- **Initialization:** The initial population $P(0)$ is generated randomly with feasible solutions, *i.e.*, $fit > 0$.
- **Evolution Operations:** The evolution process incorporates random selection, one-point crossover and random bit mutation to generate $P'(t+1)$. The elite chromosomes with highest fitness from $P(t)$ and $P'(t+1)$ are chosen to form the next population $P(t+1)$.

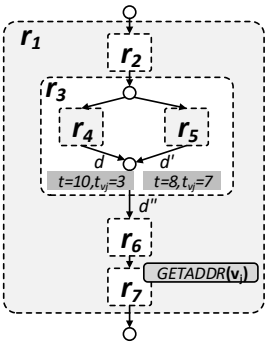


Figure 6 WCET Example: Merging states from different paths.

- **Termination:** The algorithm is terminated after k generations or if the best chromosome does not change for n generations.

The address assignment algorithm is depicted in Algorithm 1. Given a chromosome, the region tree is traversed in topological order assigning addresses to the allocated objects in each region. The topological order visits all the nodes with the same parent before visiting the children. For the root of a function, all the parents (call regions) of the function are visited before the root of the function. Also, for a sequence of sequentially composed regions, the order of the sequence is maintained. After the objects in a region are assigned to SPM addresses, an end address to the last allocated address is maintained. For each region r_n, the previous end address is the maximum of all parent regions (note that if r_n is not the root of its function, it has a single parent region). For a region that is not sequentially composed or the first region in a sequence of regions, addresses are iteratively assigned to the allocated objects starting from the previous end address. For a region in a sequence, an allocated object maintains the same address as the previous region if the object is allocated in both. Otherwise, a best fit algorithm is used to assign the remaining addresses. The end address for each region is then computed as the maximum end address for any allocated object. Note that the algorithm trivially ensures that objects allocated in a region cannot overlap with any object that is available in a parent; hence, Equation 5 is always satisfied. However, the algorithm is not optimal, since it does not consider that an allocation might not be required in any context where the object is already available in the SPM. Finally, the allocation is considered feasible only if the end address never exceeds the SPM size; this guarantees that Equation 4 is also satisfied.

7 WCET Analysis

We discuss how to model the behavior of our prefetch mechanism in the context of static timing analysis so that a safe bound to the WCET of the program running uninterrupted can be computed. We assume a given allocation solution computed based on Section 6. We rely on the standard approach of Data Flow Analysis (DFA) [27], where the detailed state of the hardware is generalized into an *abstract state* based on the theory of abstract interpretation [4, 23]. To avoid maintaining a different state for each path through the program, the analysis relies on computing fixed points by "merging" states when paths joins (*i.e.*, branch join and loops entry/exit). In detail, given two abstract states d and d', we need to compute a *join operator* $\vee$ such that the resulting state $d'' = d \vee d'$ is more general than either d or d'. We model time as natural numbers, processor clock cycles.

Consider as an example the execution of the CFG and associated region tree in Figure 6 in a context σ. Assume that the analysis for the path through r_4^σ has determined an upper bound to the execution time of the program up to this point equal to $t = 10$, and an upper bound to the remaining time to complete a DMA fetch operation for an object v_j equal to $t_{v_j} = 3$. For the path through r_5^σ, we instead have $t = 8, t_{v_j} = 7$, *i.e.*, the execution takes longer along the path through r_4^σ than through r_5^σ, but results in a shorter remaining DMA time. Assume now that a `GETADDR` command on object v_j is executed at the beginning of region/context r_7^σ. The amount of time that the command will block is then equal to t_{v_j} minus the amount of overlap that the DMA operation has with r_6^σ, or zero if the operation completes during r_6^σ. Assume a simple case where the execution through r_6^σ requires Δ units of time and performs no access to main memory, so that the DMA operation can overlap up to Δ. The program can then resume from `GETADDR` at time $t + \Delta + \max(t_{v_j} - \Delta, 0)$. Hence, note that for $\Delta = 7$, the worst case path is through r_4^σ, resulting in a time of 17 units against 15 for the path through r_5^σ. However, for $\Delta = 3$, the worst case path is through r_5^σ, with a time of 15 time units against 13 for the path through r_4^σ. In summary, we cannot determine which path through a branch leads to the worst case unless we analyze the regions following the branch in the CFG (r_6^σ and r_7^σ in the example).

If we do not want to keep both states after the branch, a trivial solution would be to merge them by computing a join state with $t = \max(10, 8) = 10$ and $t_{v_j} = \max(3, 7) = 7$. However, this would lead us to over-approximate the time for the `GETADDR`, resulting in 17 time units for $\Delta = 3$, rather than the computed bound of 15 time units. Therefore, we seek to derive a tighter abstraction. Due to the inherent complexity in the theory of abstract interpretation, in this section we present the main intuition about our abstraction and why it results in a safe WCET bound; a formal proof of correctness is provided in the technical report [20].

Intuitively, every abstract state d is composed of two information: the elapsed program execution time $d.t$, and a set of *timers* $\{t_{v_j}\}$. For an object v_j, $d.t_{v_j}$ represents the worst case time required to complete either a prefetch or write-back operation in the allocation queue; since the allocation queue is served in FIFO order, this represents the time to transfer that specific object, plus the time required for all operations ahead of it in the queue. For the example in Figure 6, let d be the state through r_4^σ and d' be the state through r_5^σ. Since there is only one DMA operation in the queue, we have $d.t = 10, d.t_{v_j} = 3$ and $d'.t = 8, d'.t_{v_j} = 7$. The join state $d'' = d \vee d'$ is then computed as follows:

$$d''.t = t_{\max} = \max(d.t, d'.t), \tag{6}$$

and for every timer t_{v_j}:

$$d''.t_{v_j} = \max\big(d.t_{v_j} - (t_{\max} - d.t), d'.t_{v_j} - (t_{\max} - d'.t)\big). \tag{7}$$

Based on Equations 6, 7, we compute a join state for the example $d''.t = \max(10, 8) = 10, d''.t_{v_j} = (3 - (10 - 10), 7 - (10 - 8)) = 5$. Note that this abstraction is tighter compared to the values $t = 10, t_{v_j} = 7$ obtained by the trivial over-approximation. In particular, it is easy to see that for the provided example, the time for the `GETADDR` command computed based on d'' is *exactly* equal to the worst case between d and d' for any value of Δ, albeit for more complex cases involving multiple DMA operations it is still a (tighter) over-approximation. The key intuition is that adding Δ units of time to the execution time of the program is always worse than adding Δ units of time to the length of timers, since a `GETADDR` might block the program for a time at most equal to the length of the corresponding timer. Hence, if the execution time along two paths differs by a value Δ, we are guaranteed to obtain an

upper bound if we consider the longest execution time but subtract Δ units of time from the timers along the shortest path, as performed in Equation 7.

Note that in general, a single DMA operation could overlap with many regions, and the amount of overlap can be further modified by the path through each region and allocation commands for both the same and other objects. Due to the presence of the max term in Equation 7, modeling the WCET problem as an ILP (a technique also known as implicit path enumeration) would require adding a large number of auxiliary variables. Therefore, we propose to instead compute the WCET using a structure-based approach [27] using the region tree, as summarized in Algorithm 2.

Algorithm 2 WCET Analysis

Input: initial program state d with $d.t = 0$, region information, allocation solution

```
1:  d ← ANALYZE_REGION(r_1, ∅, d)
2:  return d.t + max_{v_j}{d.t_{v_j}}

3:  function ANALYZE_REGION(r, σ, d)
4:      if r is trivial region then
5:          d ← STATE_TRANSFER(r, σ, d)
6:          if r calls a region r_n then
7:              d ← ANALYZE_REGION(r_n, σ ∪ r, d)
8:      else
9:          for all paths p_i in r do
10:             d_i ← d
11:             for all subregions r_n along p_i do
12:                 d_i ← ANALYZE_REGION(r_n, σ, d_i)
13:         d ← JOIN(r, σ, {d_i})
14:     return d
```

Starting from an initial abstract program state d and region r_1, the root of the main function, the algorithm recursively calls function $ANALYZE\_REGION$ to update state d based on the execution of region r in context σ. If r is a trivial region, then function $STATE\_TRANSFER$ is used to update d based on the region's code, including any allocation command. Note that we need to pass the context σ to the function, since as explained in Section 6, the availability and address of objects in the SPM depends on the context for the region. If the region is a call region, we also need to recursively invoke $ANALYZE\_REGION$ on the called region after updating the context. If region r is not trivial, then we need to recursively analyze all sub-regions along every path in r; this results in an updated state d_i for each path p_i. The states are then joined by function $JOIN$. If region r has no backedge (*i.e.*, it is not a loop), then the function simply applies the join operator over all states d_i. If the region is a loop, then function $JOIN$ performs a fixed-point iteration over the abstract state based on loop iteration bounds. At the end of the analysis, we return the total elapsed time plus the maximum timer length, to indicate the need to complete any remaining write back operation.

Finally, note that while we focused on modeling the behavior of DMA operations, the abstract state can also model both architectural states, such as the state of the processor pipeline [23], as well as the value of program variables, which can be used to exclude invalid paths (flow analysis) and compute loop bounds [13].

8 Implementation

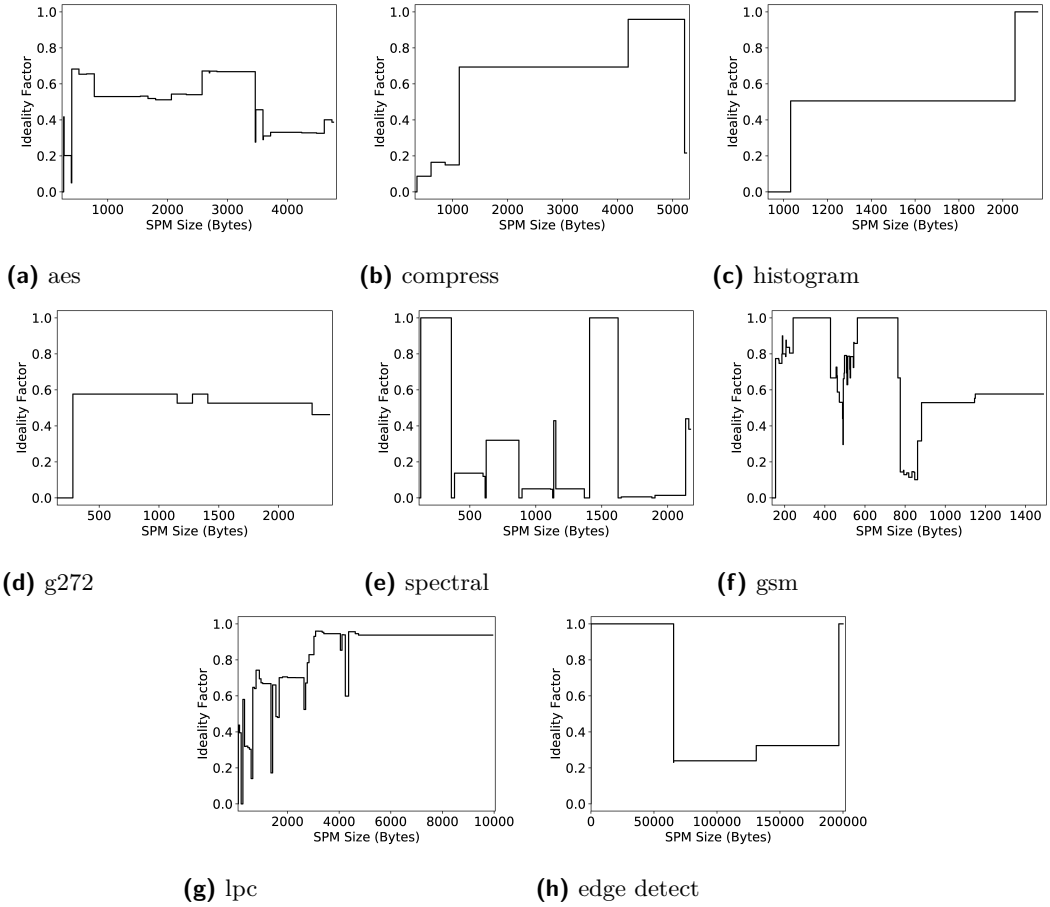
A compiler-integrated flow is used to implement the allocation algorithm. The flow analyzes the program, runs the allocation algorithm, applies the required transformations, and generates an executable. We integrated our flow with the open-source LLVM compiler [12]. The following passes are applied on the Intermediate Representation (IR).

- **Convert Stack Objects to Globals.** Each function frame in the stack has two components: (1) temporary spilled registers and calling context; (2) local objects. Allocating the full stack might be unfeasible if the maximum stack depth does not fit in the SPM. In order to allow a flexible allocation scheme for the stack, a pass is implemented to promote large local objects to global objects [11]. This reduces the maximum stack size, so that the stack can be considered an object in the allocation algorithm, and allows allocating local objects either in main memory or in the SPM without the need to manage multiple stacks.
- **Region Tree Generation.** We use the provided region analysis in LLVM to construct the refined region tree.
- **SPM Allocation.** The allocation algorithm generates an optimized allocation solution. As discussed in Section 6.2, we compute the fitness of a feasible solution by analyzing the WCET. So, the code is transformed to insert the allocation commands and to modify the memory references and then the program is analyzed for WCET.
- **Code Transformation.** Transforming the code includes inserting allocation commands and modifying memory references. As each region is defined by two edges, we simply insert a new basic block with the allocation commands (`ALLOC` / `DEALLOC`) on this edge (entry/exit). Most of these basic blocks are optimized by the compiler and integrated with other basic blocks when possible. For `GETADDR` commands, we find the first instruction that references an object after an allocation/de-allocation and insert `GETADDR` before it. After that, all the references to the object until the next allocation/de-allocation are modified to the address returned by `GETADDR` command.
- **WCET Analysis.** The LLVM IR code is compiled to assembly code for the target processor. The execution time for each basic block is extracted from the program assembly based on the processor model. We use the information from the compiler back-end to conduct the WCET analysis.
- **Code Generation.** The assembly code for the final allocation is generated and a linker script that specifies the memory sections is used to produce the executable.

9 Evaluation

The evaluation of the prefetching approach for data SPM is performed using a simple MIPS processor model with a 5-stages pipeline and no branch predictor. For memory instructions, we consider a latency for a word access of 10 cycles to main memory, 1 cycle to SPM and 1 cycle to the SPM controller. For the DMA, we use a similar model as in [28] such that the latency to initialize the transfer to/from main memory is 10 cycles and the latency per word is 2 cycles.

We tested the allocation algorithm for multiple benchmarks from UTDSP, MediaBench, and CHStone suites. We present 8 kernels from these suites. We avoided benchmarks that have the following criteria: (1) benchmarks with system calls, as we cannot analyze their WCET without the OS code; (2) benchmarks that access only the stack or have very small sizes for static and local objects. Note that the stack always resides in the SPM as its size



(a) aes **(b)** compress **(c)** histogram

(d) g272 **(e)** spectral **(f)** gsm

(g) lpc **(h)** edge detect

Figure 7 Ideality factor.

becomes small after converting stack variables to globals as discussed in Section 8 and its access rate is usually high.

As the benchmarks available for real-time systems are usually small kernels, we focus on the performance of the prefetching algorithm compared to dynamic allocation rather than the total profit of the allocation. We were not able to apply the algorithm to other suites with more realistic applications, *e.g.* SPEC2000, as they have system calls, recursion, unknown loop bounds, and calls to standard libraries which makes it unsuitable to derive WCET estimation. We plan to explore other benchmarks in the future.

We consider three cases: (1) dynamic allocation without prefetching (*base*); (2) dynamic allocation with prefetching (*pref*); (3) and dynamic allocation with no cost (*ideal*). We define the *base* case as the worst case for prefetching where the CPU has to stall for every memory transfer. We also define the *ideal* case as the dynamic allocation with no cost for memory transfer. Figure 7 shows the *ideality factor* as a function of the size of the SPM. The ideality factor is computed as: $\textit{ideality factor} = \frac{WCET(base)-WCET(pref)}{WCET(base)-WCET(ideal)}$. The denominator of the ideality factor represents the best hypothetical improvement in WCET that prefetching can achieve relative to the base dynamic allocation and the numerator is the improvement for the prefetching case. The ideality factor is an indication for the performance of the prefetching approach, with a value of 1 indicating a performance equivalent to the ideal case, *i.e.*, there is no allocation cost as all memory transfers are overlapped with CPU

execution. For each benchmark, we vary the range of the SPM sizes starting from the size in which at least one object can fit in the SPM to the size that can fit all objects.

The solving time for the allocation algorithm depends on the number of regions in the program, the number of objects that can fit in the SPM and the genetic algorithm parameters. In the experiments, we used a population of 100 chromosomes and termination parameters $k = 500$, $n = 10$. The solving time varied between few seconds to around 15 minutes. Inserting the allocation commands increases the executable code size by at most 1.2% for the tested programs.

9.1 Results Analysis

Benchmark 'histogram' has two main arrays with size 1024 bytes each. When the size of the SPM is 1024 bytes, it can fit only one of them and dynamic allocation is able to arbitrate between the two arrays. Prefetching can overlap part of the cost needed for dynamic allocation. When the SPM size is 2048, both arrays can fit in the SPM and also prefetching technique can hide the whole memory time required to transfer the arrays as it can overlap the transfer of one array with the use of the other array in the SPM.

For benchmark 'g272', prefetching technique can only overlap part of the memory transfer as the live range of the used objects are overlapped, *i.e.*, the chance to transfer one object while using the others is low.

Benchmark 'edge_detect' has three arrays with size 64 Kbyte each and a small array with size 36 bytes. For small SPM size, only the small array can fit and prefetching can overlap its memory transfer time. When the SPM can fit one of the large arrays at any program point, prefetching can overlap 25% of memory transfer time. Similarly, prefetching can overlap 33% of the transfer time when the SPM can fit two large arrays. When the SPM can fit all the large arrays, all the memory transfer time can be hidden through prefetching.

The other benchmarks have more objects and the live ranges are more nested. The ideality factor varies as the SPM size increases and larger objects or more objects can fit in the SPM; hence more memory transfers are introduced. If the added space is used to arbitrate for objects, prefetching might not have enough time to overlap the memory transfers. If the space allows objects to exist in the SPM simultaneously, prefetching performs better as it has more opportunity to overlap the memory transfers.

Although the dynamic prefetching approach is able to exploit the opportunities to hide the latency of memory transfers, object-based allocation fails short in terms of space and time in some cases. That is, considering only objects that can fit entirely in the SPM at each program point limits the allocation efficiency. We argue that the benefit of our approach will increase for smaller allocation granularity. So, we plan to extend the framework to be able to allocate parts of an array or data structure to allow more allocation flexibility, increase the efficiency of allocation for smaller SPM sizes, and provide more chances for prefetching.

10 Conclusions

In this paper, we introduced a framework for predictable data SPM prefetching. Our approach is automated within a compilation flow that is integrated with the LLVM compiler. We provided a hardware/software design that includes an SPM controller, an allocation algorithm and a WCET analysis. The experiments have shown the potential of our prefetching technique to provide a predictable mechanism to hide the latency of main memory transfers and efficiently manage the data SPM with low overhead.

Our framework can be extended to handle pointer-based memory accesses for static, stack and dynamically allocated objects. Using techniques like data pipelining can enhance the efficiency of the allocation algorithm for small SPM sizes by allocating portions of an object. We plan to integrate these mechanisms in our framework in future work.

Acknowledgements. We would like to thank Ondřej Lhoták for the helpful discussions and support on compiler design and implementation.

References

1. 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.
2. Oren Avissar, Rajeev Barua, and Dave Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Trans. Embed. Comput. Syst.*, 1(1):6–26, November 2002.
3. P. Burgio, A. Marongiu, P. Valente, and M. Bertogna. A memory-centric approach to enable timing-predictability within embedded many-core accelerators. In *Real-Time and Embedded Systems and Technologies (RTEST), 2015 CSI Symposium on*, pages 1–8, Oct 2015.
4. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
5. M. Dasygenis, E. Brockmeyer, B. Durinck, F. Catthoor, D. Soudris, and A. Thanailakis. A combined dma and application-specific prefetching approach for tackling the memory latency bottleneck. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(3):279–291, March 2006.
6. Jean-Francois Deverge and Isabelle Puaut. Wcet-directed dynamic scratchpad memory allocation of data. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, ECRTS'07, pages 179–190, Washington, DC, USA, 2007. IEEE Computer Society.
7. Angel Dominguez, Sumesh Udayakumaran, and Rajeev Barua. Heap data allocation to scratch-pad memory in embedded systems. *J. Embedded Comput.*, 1(4):521–540, December 2005.
8. Poletti Francesco, Paul Marchal, David Atienza, Luca Benini, Francky Catthoor, and Jose M. Mendias. An integrated hardware/software approach for run-time scratchpad management. In *Proceedings of the 41st Annual Design Automation Conference*, DAC'04, pages 238–243, New York, NY, USA, 2004. ACM.
9. 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.*, 48(2):32:1–32:36, November 2015.
10. Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI'94, pages 171–185, New York, NY, USA, 1994. ACM.
11. Sungjun Kim. Using scratchpad memory for stack data in hard real-time embedded systems. In *Proceedings of the Memory Architecture and Organization Workshop*, 2011.
12. Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Gener-*

ation and Optimization: Feedback-directed and Runtime Optimization, CGO'04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

13 Thomas Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, School of Computer Science and Engineering, Chalmers University of Technology, Sweden, 2002.

14 R. Mancuso, R. Dudko, and M. Caccamo. Light-PREM: Automated software refactoring for predictable execution on cots embedded systems. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10, Aug 2014.

15 Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio Buttazzo. Memory-processor co-scheduling in fixed priority systems. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS'15, pages 87–96, New York, NY, USA, 2015. ACM.

16 Sparsh Mittal. A survey of recent prefetching techniques for processor caches. *ACM Comput. Surv.*, 49(2):35:1–35:35, August 2016.

17 Nghi Nguyen, Angel Dominguez, and Rajeev Barua. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. *ACM Trans. Embed. Comput. Syst.*, 8(3):21:1–21:32, April 2009.

18 David Patterson and John L. Hennessy. *Computer architecture: a quantitative approach*. Elsevier, 2012.

19 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.

20 Muhammad R. Soliman and Rodolfo Pellizzoni. Data Scratchpad Prefetching for Real-time Systems. Technical report, University of Waterloo, UWSpace, 2017. URL: `http://hdl.handle.net/10012/11837`.

21 V. Suhendra, T. Mitra, A. Roychoudhury, and Ting Chen. Wcet centric data allocation to scratchpad memory. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 10 pp.–232, Dec 2005.

22 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.

23 Stephan Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Universität des Saarlandes, 2004.

24 Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.*, 5(2):472–511, May 2006.

25 Jussi Vanhatalo, Hagen Völzer, and Jana Koehler. The refined process structure tree. In *Proceedings of the 6th International Conference on Business Process Management*, BPM'08, pages 100–115, Berlin, Heidelberg, 2008. Springer-Verlag.

26 J. Whitham and N. Audsley. Studying the applicability of the scratchpad memory management unit. In *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 205–214, April 2010.

27 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. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008. `doi:10.1145/1347375.1347389`.

28 Xuejun Yang, Li Wang, Jingling Xue, Tao Tang, Xiaoguang Ren, and Sen Ye. Improving scratchpad allocation with demand-driven data tiling. In *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES'10, pages 127–136, New York, NY, USA, 2010. ACM.

29 Y. Yang, M. Wang, Z. Shao, and M. Guo. Dynamic scratch-pad memory management with data pipelining for embedded systems. In *Computational Science and Engineering, 2009. CSE'09. International Conference on*, volume 2, pages 358–365, Aug 2009.

A Linux Real-Time Packet Scheduler for Reliable Static SDN Routing*†

Tao Qian[1], Frank Mueller[2], and Yufeng Xin[3]

1 **North Carolina State University, Raleigh, NC, USA**
`tqian2@ncsu.edu`
2 **North Carolina State University, Raleigh, NC, USA**
`mueller@cs.ncsu.edu`
3 **RENCI, The University of North Carolina, Chapel Hill, NC, USA**
`yxin@renci.org`

Abstract

In a distributed computing environment, guaranteeing the hard deadline for real-time messages is essential to ensure schedulability of real-time tasks. Since capabilities of the shared resources for transmission are limited, e.g., the buffer size is limited on network devices, it becomes a challenge to design an effective and feasible resource sharing policy based on both the demand of real-time packet transmissions and the limitation of resource capabilities. We address this challenge in two cooperative mechanisms. First, we design a static routing algorithm to find forwarding paths for packets to guarantee their hard deadlines. The routing algorithm employs a validation-based backtracking procedure capable of deriving the demand of a set of real-time packets on each shared network device, and it checks whether this demand can be met on the device. Second, we design a packet scheduler that runs on network devices to transmit messages according to our routing requirements. We implement these mechanisms on virtual software-defined network (SDN) switches and evaluate them on real hardware in a local cluster to demonstrate the feasibility and effectiveness of our routing algorithm and packet scheduler.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

Keywords and phrases real-time networks, packet scheduling, deadline guarantee

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.25

1 Introduction

In a distributed computing environment, multiple compute nodes share communication resources to transmit data in order to collaborate with each other. In such systems, employing an effective resource sharing mechanism is essential to meet the real-time requirements of tasks. These mechanisms can be divided into two categories. First, the compute nodes control their own behavior of how to utilize shared resources. Past research has studied mechanisms of shaping the resource access pattern to increase timing predictability, e.g., memory sharing based on limiting the memory bandwidth for different cores [31, 32] and network bandwidth limitation on compute nodes connected via Ethernet [23]. These mechanisms are *passive* since they can only reduce the probability of resource contention instead of preventing contention in the first place. Thus, passive mechanisms usually guarantee probabilistic deadlines. The second category includes *active* resource sharing mechanisms, which either assign the resource

* This work was supported in part by NSF grants 1525609, 1329780, 1239246, 0905181 and 0958311.
† Aranya Chakrabortty helped to scope the problem in discussions.

to exactly one task at a time to prevent contention (e.g., TDMA-based bus sharing on a multiprocessor platform [25]), or provide mechanisms on the shared resources to control its usage directly [1, 14]. One benefit of active mechanisms is that by controlling how resources are shared, one can build up a mathematical model to estimate the resource access time. We believe this model is a necessary condition to guarantee hard deadlines.

In our previous work, we implemented a cyclic executive based task scheduler on distributed nodes to guarantee probabilistic task deadlines [21] and a hybrid earliest-deadline-first (EDF) task and packet scheduler to guarantee hard deadlines [23]. Furthermore, we have implemented a real-time distributed hash table (RT-DHT) to support the scalability and resilience requirements of distributed wide-area measurement systems, which estimate power grid oscillation modes based on real-time power state data [22, 19]. A passive mechanism (bandwidth limitation on DHT nodes) was adopted to reduce variations of the network delay in order to increase the schedulability of our hybrid scheduler [23]. However, when the RT-DHT has to share network resources with other systems whose network utilization patterns are unpredictable, this passive mechanism is not enough since it cannot restrict the network access of other systems. In this work, we actively control network devices (e.g., switches) by enforcing a packet scheduling algorithm on the devices to guarantee hard message transmission deadlines for real-time packets, even when these devices are subject to unpredictable background traffic.

One of the challenges of adopting the active mechanism is to determine an accurate resource sharing policy that considers both the demands of the real-time tasks and the capacities of the shared resources. For example, an active memory controller needs to know the bandwidth demands of the running tasks to proactively reserve bandwidth for each of them. This challenge becomes even harder for a network since the topology of the network can be complex (e.g., network devices have to collaborate in order to guarantee the deadline). We need to derive an effective static routing algorithm to determine the forwarding paths for all real-time packets so that the end-to-end delay for such a packet through the corresponding path never exceeds its deadline. Without loss of generality, we use term *packet* and *message* interchangeably in this paper.

One possible approach to address this problem is to express it as an integer linear programming (ILP) problem if the objective function and constraints are linear. In our case, each real-time message could have multiple forwarding paths with different costs for each path. For example, one cost is the size of the buffer the message will occupy on every network device along the path. Then the problem becomes to assign a forwarding path for each real-time message under the condition that the constraints on each shared device can be met (in this case, the total size of messages is no more than the buffer size of the shared device at any time). The effectiveness of this model is based on the accuracy of the cost function. However, for the routing problem, the cost function on each device is dependent on the exact traffic that goes through that device (i.e., dependent on the assignment results). This cyclic dependency makes this a hard problem. Previous work has proposed to utilize different oracles to break the cyclic dependency [12]. For example, assuming an oracle that determines the transmission time on any device for any message at any time exists, the assignment problem then can be solved with ILP techniques. In this paper, we enforce an active message scheduling algorithm on network devices so we can derive the cost function for each message and each device on the path for all traffic that goes through the device. As a result, we use a validation-based backtracking algorithm to determine the forwarding path for each message (detailed in Section 2.3).

The objective of our scheduling algorithm is to avoid dropping real-time messages even when network congestion occurs, e.g., due to too many real-time messages and background

traffic (considered as non real-time). First, the routing algorithm needs to guarantee that when multiple real-time messages share a device on their forwarding paths, the aggregate message size does not exceed the buffer size of that device. Thus, the scheduling algorithm needs to control the timing when a message is transmitted from one device to the next. The transmission time is planned statically so that the aggregate size of real-time messages on a device can be calculated in our static routing algorithm. Second, the scheduling algorithm needs to drop background traffic when the available buffer on a device is insufficient for real-time messages. In this way, our scheduling algorithm guarantees real-time message deadlines while providing a best-effort service to background traffic.

In summary, the contributions of this work are: (1) We design a static routing algorithm to find forwarding paths for multiple real-time messages to guarantee the hard deadlines of messages. (2) We propose a deadline-based scheduling algorithm, which actively controls the time at which real-time messages are processed at each device and only drops background packets when the device is congested. (3) We implement the packet scheduler on virtual SDN switches and conduct experiments to evaluate our approach.

The rest of paper is organized as follows. Section 2 presents the design of our static routing algorithm and the active device scheduling algorithm. Section 3 presents the implementation details of the scheduling algorithm on virtual switches. Section 4 discusses the evaluation setup and results. Section 5 contrasts our work with related work. Section 6 presents the conclusion and on-going research.

2 Design

This section first presents the system model and the objectives. It then contributes the static routing algorithm that determines forwarding paths for multiple pairs of source and destination of real-time messages. After that, it contributes the scheduling algorithm that runs on network devices to guarantee that real-time messages are transmitted in a time predictable fashion.

2.1 System Model and Objectives

Network Model: We use notation (V, E, B, Pr) to represent the underlying network utilized by compute nodes in distributed real-time systems to exchange messages. V denotes a finite set of compute nodes and the networking hardware, which forwards messages between compute nodes. Without loss of generality, we use the term *node* to represent either a networking device or a compute node. Let B be buffer sizes on nodes, i.e., the aggregate size of messages that can be stored in the queue on each node. If a burst of messages arriving at a node exceeds the buffer size, a fraction of these messages will be dropped. Thus, one objective of our system is to guarantee that only background traffic (i.e., messages without deadline requirements) can be dropped in such a case. Matrix E represents the real-time links between nodes in the graph. Nodes v_1 and v_2 have a physical connection for real-time traffic iff $E[v_1, v_2] > 0$, where $E[v_1, v_2]$ is the interface speed. Matrix Pr represents the propagation delays on these links. Table 1 summarizes the notation. When a network runs in stable state, its nodes and links do not change over time. Thus, V, E, B, and Pr are constant. We require that clock times on nodes are synchronized. This can be achieved via the Network Time Protocol [18], running at system startup time and periodically as a real-time task, or hardware support (e.g., GPS of phasor measurement units in the power grid).

Table 1 Notation.

Name	Meaning
V	set of nodes
$B[v]$	buffer size on node v
$E[v_1, v_2]$	constant interface speed matrix for real-time traffic between nodes v_1 and v_2
$Pr[v_1, v_2]$	constant propagation delay matrix on links between v_1 and v_2
D_m	relative deadlines of a message flow m
T_m	periods of a message flow m
S_m	size of messages in flow m
$R[v]$	expected message response time on node v
$A[v]$	latest message transmission time on node v
$\delta[v]$	runtime message response time variance on node v
$\Delta[v]$	response time variance bound on node v, determined by all message flows on the node

Node Architecture: We consider a store-and-forward node model. A packet arriving at a node is first put into the input buffer. Then, the node processor (i.e., forwarding engine) processes the packet to determine the forwarding rule for that packet (e.g., time to forward the packet and output link) and forwards the packet to the corresponding output queue at scheduled time (e.g., Cisco Calalyst Switches [6]). This is further detailed in Section 2.2.

Message Release Model: We consider two types of messages. First are messages due to background traffic without deadlines. Second are real-time messages released periodically by real-time tasks running on one end node, which are subsequently transmitted to another node. These messages can be classified as message flows, where a flow contains all messages released by the same real-time task. Thus, messages in the same flow have the same source and destination nodes, same relative deadline, and are released periodically. We define a set of message flows as $M = (D, T, S)$, where D is the relative deadlines of the message flows, T is the periods, and S is the sizes. The flow id and the release time of a message are embedded in the message header upon transmission. Nodes use the flow id to look up the forwarding policy for that flow from their routing table and use the release time to calculate the exact forwarding time for a specific message in that flow.

Objectives: Given the configuration of real-time flows, our static routing algorithm shall derive a forwarding path for each flow, such that (1) the transmission time of a real-time message shall never exceed its relative deadline, (2) the aggregate size of real-time messages on any network device shall never exceed the buffer size of that device, (3) when multiple real-time messages share a network device on their paths, the network device shall have the computing capability to process them before their local deadlines. To achieve these goals, the static routing algorithm derives offline the expected response time R by which each node must have processed a real-time message. The goal for the scheduler is to guarantee end-to-end message deadlines by enforcing the expected response time on each node and considering the runtime response time variance δ caused by the interference due to scheduling and queueing at the output interface. Section 2.3 proves that the scheduler can adopt an EDF-based algorithm that uses R as local deadlines to achieve this goal.

Message Delay Model: We focus on real-time messages to be transmitted before their deadlines. Thus, given any real-time message flow, the objective is to find a forwarding path for which the end-to-end delay does not exceed the relative deadline for the flow. More formally, let the forwarding path for a flow be $(v_0, v_1, ..., v_k)$, where k is the path length and $v_i \in V (0 \leq i \leq k)$ are the nodes on the forwarding path. Let $t[v_i]$ be the time when the message arrives at node v_i. $t[v_0]$ is the message release time, which is determined by the property of the corresponding message flow. $t[v_i]$, $0 < i \leq k$, can be formalized as a function of the path and the underlying network as shown in Eq. 1, where $R[v_i]$ is the expected message response time on node v_i and $\delta[v_i]$ is the runtime variance for the message response time (i.e., node v_i starts to transmit the message at time $t[v_i] + R[v_i]$ and finishes the transmission at time $t[v_i] + R[v_i] + \delta[v_i])$. The first sum in Eq. 1 is the accumulated propagation delay on links from v_0 to v_i. The second sum is the accumulated message response time on nodes v_0 to v_{i-1}. The subscript m for the message flow is abbreviated when the equations are suitable for every message flow (e.g., $R[v_j]$ instead of $R_m[v_j]$).

$$\begin{aligned} t[v_i] &= t[v_{i-1}] + R[v_{i-1}] + \delta[v_{i-1}] + Pr[v_{i-1}, v_i] \\ &= t[v_0] + \sum_{j=0}^{i-1} Pr[v_j, v_{j+1}] + \sum_{j=0}^{i-1} (R[v_j] + \delta[v_j]). \end{aligned} \tag{1}$$

Let $A[v_i]$ be the worst case (i.e., the latest time) that node v_i has to start the transmission. On the first node, $A[v_0] = t[v_0] + R[v_0]$. We assume that $\delta[v_i] \geq 0$ on any node v_i. $\delta[v_i]$ can be bounded by a value $\Delta[v_i]$, which is determined by all message flows on node v_i and is the same for any individual message on this node. Then, the latest transmission time $A[v_i]$ $(0 < i \leq k)$ occurs when the message transmissions on all nodes before v_i have experienced the worst variation, which is expressed in Eq. 2. Thus, $A[v_i]$ can be calculated for a message once R and Δ are derived. Sections 2.2 and 2.3 detail the approach to derive R and Δ offline.

$$\begin{aligned} A[v_i] &= max\{t[v_i] + R[v_i]\} = max\{t[v_i]\} + R[v_i] \\ &= t[v_0] + \sum_{j=0}^{i-1} Pr[v_j, v_{j+1}] + \sum_{j=0}^{i} R[v_j] + \sum_{j=0}^{i-1} \Delta[v_j]\,. \end{aligned} \tag{2}$$

As a result, the static routing algorithm needs to find a forwarding path for every message flow so that the end-to-end delay in the worst case is bounded by its relative deadline as expressed in Eq. 3.

$$A[v_k] + \Delta[v_k] - t[v_0] \leq D\,. \tag{3}$$

2.2 Message Scheduler

A real-time message experiences three phases on one node. Fig. 1 illustrates these phases on node v, where the message is sent by node v' and received by node v, and then forwarded to node v''. Before the message is sent to node v, the processor on node v' has calculated the latest message transmission time for the next node (i.e., $A[v]$) from the forwarding table. In the first phase, the message arrives from link 1 at time $t[v]$ and is put in the input buffer. We assume that the time for an interface to put a packet into the input buffer can be ignored. In the second phase, the message is processed and moved to the intermediate queue. During processing, the latest message transmission time for the next node (i.e., $A[v'']$) is calculated and the output interface for the message is determined according to the forwarding table.

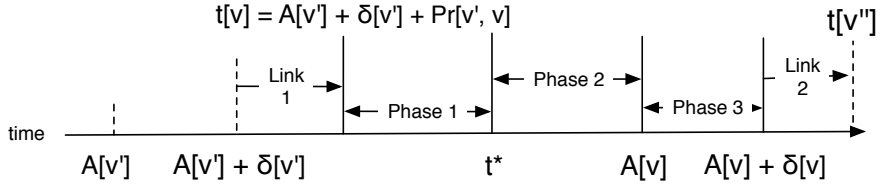


Figure 1 Message Phases on Node.

The message stays in the intermediate queue during the second phase. In the third phase, the message is forwarded to the corresponding interface at time $A[v]$ and then finally transmitted via link 2 at time $A[v] + \delta[v]$. The intuition of this design is to constrain the message arrival time at each node v to a time interval $(A[v'] + Pr[v', v], A[v'] + Pr[v', v] + \Delta[v'])$. In contrast, a background traffic packet is moved from the input buffer to the corresponding interface directly when no real-time messages need to be processed.

To guarantee that the processor can start the transmission in the third phase at time $A[v]$, the processor has to finish processing the message before $A[v]$ in the second phase (at time t^* in Fig. 1). Thus, our packet scheduler adopts an EDF-based algorithm to process the message before its local relative deadline $A[v] - t[v]$. Since $A[v] = A[v'] + Pr[v', v] + \Delta[v'] + R[v]$, which is derived from Eq. 2, the local relative deadline $A[v] - t[v]$ is no less than the expected response time $R[v]$ as derived in Eq. 4. If the processor has the computation capability to process every real-time message before its expected response time $R[v]$, the processor can forward it on time.

$$\begin{aligned} A[v] - t[v] &= (A[v'] + Pr[v', v] + \Delta[v'] + R[v]) - t[v] \\ &= (A[v'] + Pr[v', v] + \Delta[v'] + R[v]) \\ &\quad - (A[v'] + Pr[v', v] + \delta[v']) \\ &= R[v] + \Delta[v'] - \delta[v'] \\ &\geq R[v]. \end{aligned} \tag{4}$$

Algorithm 1 depicts the packet scheduler. The input buffer is organized into a real-time message section and a background traffic section. Each section maintains the front and tail of the corresponding queue. These sections are stored in a circular fashion. The growing of one end of a section can reach the other end of the other section. An incoming real-time message is stored at any end with sufficient space in the real-time message section. If neither ends have sufficient space, background packets at the tail of the background section are dropped until space is sufficient for the real-time message. An incoming background packet is dropped immediately if the input buffer has insufficient space. Otherwise, if the background traffic section has insufficient space at its tail, real-time messages are moved from the head to the tail of the real-time section. An incoming background packet is always stored at the tail of the background traffic section to support an FCFS service. The scheduler scans the real-time messages in the input buffer and processes the message with the smallest $A[v]$ but $t_{current} \geq A[v] - R[v]$. The second condition is to guarantee that a message will not be processed before its latest release time (see lines 4 to 31). In the algorithm, $\bar{A}$ represents the earliest transmission time of all real-time messages in the intermediate queue. The scheduler compares $\bar{A}$ with the current time to determine whether the transmission time of any message has passed. The scheduler transmits all messages whose transmission times have passed (see lines 32 to 36).

```
1  Message Scheduler (node v)
     Input : Packet Input Buffer Q
2    Ā ← ∞
3    while true do
4      t_current ← current time
5      target real message x ← nil
6      target background packet y ← nil
7      for each packet m in Q do
8        if m is a real-time message then
9          if t_current ≥ A_m[v] − R_m[v] then
10           if x = nil or A_m[v] < A_x[v] then
11             x ← m
12           end
13         end
14       else
15         if y = nil then
16           y ← m
17         end
18       end
19     end
20     if x ≠ nil then
21       if Ā > A_x[v] then
22         Ā ← A_x[v]
23       end
24       v′ ← next node for x
25       calculate latest transmission time for next node A_x[v′].
26       move message x into intermediate queue.
27     else
28       if y ≠ nil then
29         move y into output queue
30       end
31     end
32     t_current ← current time
33     while t_current >= Ā do
34       forward message with transmission time Ā to output interface.
35       update Ā.
36     end
37     T_s ← Ā − t_current
38     if Q is empty then
39       sleep (T_s) or wake up by packet arrival interrupt.
40     end
41   end
```

Algorithm 1: Pseudocode for message scheduler.

Our scheduler belongs to the class of non-preemptive EDF scheduling algorithms. The real-time messages can be considered as jobs of periodic tasks (i.e., real-time message flows) and instructions (lines 20 to 36) can be considered as the non-preemptive execution of the jobs. Since the expected response time R is not required to be equal to the message flow period, we utilize existing processor-demand analysis [2, 13] to perform a schedulability test for the message flows on each node to determine whether the real-time messages can be transmitted on time with the corresponding expected response time. This schedulability test is adopted by the static routing algorithm when deriving message forwarding paths in Section 2.3.

The runtime response time variation, δ, for a particular message consists of two parts. One part is the time to execute at most one iteration of the while loop (lines 4 to 31). The other part is the time to transmit other messages that have earlier transmission times in the intermediate queue than the transmission time of that particular message (lines 32 to 36). Thus, the worst case variation, Δ, is expressed by Eq. 5, where c is the worst case execution time in the first part, $B[v]$ is the buffer size of node v, and $E[v, v']$ is the bandwidth of the link that transmits any real-time messages from node v to node v'. The second part in Eq. 5 represents the time for the data to be transmitted on node v via the slowest link.

$$\Delta[v] = c + \frac{B[v]}{min\{E[v, v']\}}, \quad \textit{for all nodes v' linked to v for real-time messages.} \tag{5}$$

2.3 Routing Algorithm

Two significant differences between the objectives of our routing algorithm and other routing algorithms (see Section 5) are: (1) Our algorithm finds forwarding paths for multiple pairs of source and destination, and (2) it bounds the end-to-end delay for every pair by a predefined value (relative deadline of the flow) instead of minimizing the delay for a particular flow. Thus, we want to increase the predictability of the node behavior so that the message response time on each node is controlled and no real-time messages will be dropped.

Let us first describe the validation algorithm that checks if a set of real-time message flows with their corresponding paths is schedulable. Given a set of flows, a set of corresponding paths (one path for each flow) is schedulable if the following criteria can be met on every node in the network: (1) When a real-time message arrives at node v, the node is able to process it before time $A[v]$ (i.e., latest message response time). In addition, the node is able to transmit the message at a time in the range $(A[v], A[v] + \Delta[v])$. This is to guarantee that the message arrival time at the next node on the path can be modeled by Eq. 1 since this criterion infers $0 \leq \delta[v] \leq \Delta[v]$, which is the assumption for Eq. 1. This criterion is checked for the given message flows and the corresponding paths by the schedulability test described in Section 2.2.

(2) The residual buffer size of every node (i.e., the node buffer size minus the aggregate size of real-time messages that are resident on that node) cannot be negative at any time. Let us first consider the worst case for one message flow m on any node v. The longest time that any message in the flow is queued on this node is $\Delta[v'] + R_m[v] + \Delta[v]$, where v' is the predecessor of node v on the path. Let $\Delta[v'] = 0$ if v is the first node on the path. Thus, the number of messages of this flow on node v in the worst case is $\lceil \frac{\Delta[v'] + R_m[v] + \Delta[v]}{T_m} \rceil$. Then, the aggregate size of all message flows in the worst case must not exceed the buffer size of node v as expressed in Eq. 6. This buffer size limitation has to be validated on every node.

$$\sum_{all\ flows\ m\ on\ v} \lceil \frac{\Delta[v'] + R_m[v] + \Delta[v]}{T_m} \rceil * S_m \leq B[v]\,. \tag{6}$$

With this validation algorithm, our routing algorithm derives the forwarding paths for message flows with the following backtracking procedure. Assuming a schedulable forwarding path set is found for the first i flows, the routing algorithm checks every path candidate for flow $i+1$ until it finds a candidate that will result in a schedulable path set for all first $i+1$ flows according to the validation algorithm. If no such forwarding path is found for flow $i+1$, the algorithm backtracks to flow i and continues the process with the next candidate for flow i. A forwarding path candidate for a message flow is defined as a sequence of nodes that can transmit this flow from its source node to its destination with the expected response time on each node (i.e., $(v_0, R[v_0]), (v_1, R[v_1]), ..., (v_k, R[v_k])$). Section 2.4 describes the algorithm to find path candidates for message flows.

This algorithm always finds a schedulable forwarding path for each message flow if such a path exists since it checks all permutations. The complexity of this backtracking algorithm is exponential with the number of message flows (i.e., the same as an ILP algorithm) but this has no impact on real-time messaging as the calculations are performed offline. Nonetheless, we reduce the actual search time by optimizing the path search order. When checking the path candidates for flow $i+1$, the intuition is to give higher preference to the candidate that results in a larger residual buffer on the path if that candidate is chosen to be the forwarding path for flow $i+1$. The residual buffer size of a path is defined as the minimum residual buffer size of all nodes on the path. Since the residual buffer size of a node determines the size of background traffic that can be kept on the node, this strategy decreases the chance that certain nodes become the bottleneck for the background traffic. If two candidates result in the same residual buffer size, we give higher preference to the candidate with a shorter length. Furthermore, we give higher preference to the candidate with a shorter end-to-end delay if two candidates have the same length. The backtracking algorithm checks candidate paths for flow $i+1$ from highest preference to lowest.

2.4 Forwarding Path Candidate

Our routing algorithm requires that forwarding path candidates for message flow i are known when the algorithm attempts to find the path for that flow. A forwarding path candidate includes the nodes on the path and the expected message response time R on each node. First, we perform a breadth-first-search algorithm on the network graph to find all acyclic paths that connect the source and destination of the message flow. Then, for each path, we need to assign the expected message response time to each node based on the following three considerations.

First, the buffer size restriction described in Eq. 6 must be met on every switch. Second, the overall end-to-end delay in the worst case based on the expected message response time assignment must not exceed the relative deadline of the message flow (as shown in Eq. 3), which can also be expressed as:

$$\sum_{j=0}^{k-1} Pr[v_j, v_{j+1}] + \sum_{j=0}^{k} R[v_j] + \sum_{j=0}^{k} \Delta[v_j] \leq D.$$

$$\text{Thus,} \quad \sum_{j=0}^{k} R[v_j] \leq D - \sum_{j=0}^{k-1} Pr[v_j, v_{j+1}] - \sum_{j=0}^{k} \Delta[v_j]. \tag{7}$$

Third, the assigned expected response time must be long enough on every node v_j on the path so that these messages can be processed with their local relative deadline $R[v_j]$ as required by the validation algorithm. Since the message scheduler is non-preemptive, we use the worst

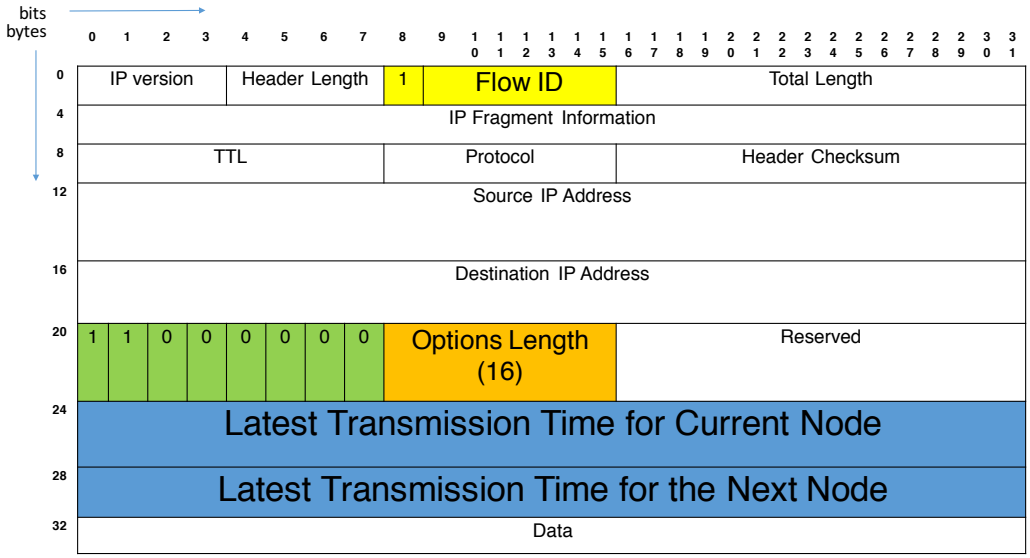


Figure 2 Message Header.

case execution time for the node to process one message (i.e., notation c as described in Section 2.2) as the unit to calculate the expected response time. Thus, $R[v_j] = c * x$, where x is at least 1 and upper bounded by Eq. 6 and 7. The assignment algorithm enumerates all values x in its range and performs processor-demand analysis for EDF scheduling to check if the corresponding response time assignment for the chosen x can be met on the network devices. As a result, multiple forwarding path candidates can be generated for each acyclic path found by the breadth-first-search algorithm.

3 Implementation

To support the three-phases message transmission (see Fig. 1), this section first presents the message structure and the extension to the Linux network stack on end nodes to specify the flow id and release time of real-time messages. It then presents the structure of forwarding tables on network devices and a scheduler implementation on virtual switches based on Open vSwitch [20, 5], a software-defined network (SDN) simulation infrastructure.

3.1 Message Structure and Construction

To support the scheduler in Algorithm 1, real-time messages need to carry the flow id, the latest transmission time for the current node $A[v]$, and the latest transmission time for the next node $A[v_{next}]$. We utilize the *Type of Service (ToS) field (i.e., bits 8–15)* and *Options field* in the standard IP Header to store them as illustrated in Fig. 2. For real-time messages, *bit* 8 is set to 1 and *bits* 9–15 are set to the message flow id. Then, a 16-bytes option (*bytes* 20–35) is used to carry the two time values in milliseconds. The type of the option (*byte* 20) is set to a value that has not been used by other protocols to prevent conflicts. For non real-time traffic, *bit* 8 is set to 0, which is the default value for the ToS field.

When a task running in user mode attempts to transmit a real-time message, it needs to specify the flow id to the network stack of Linux. The network stack software of the

node searches in the forwarding table to calculate the transmission times and determines the network interface for the transmission. Below are the most significant changes we made to Linux to support this functionality.

(1) We added a new field, fid, of type *char* in the kernel data structure *sock* so that the flow id of a socket provided by the task can be stored.

(2) We extended function *sock_setsockopt* of the Linux kernel and added a new option *SO_FLOW* so that the system call *setsockopt* assigns the message flow id when the task attempts to transmit a real-time message. *sock_setsockopt* keeps the value of the flow id in the fid field of the *sock* structure. The fid field is initialized to 0 when the *sock* structure is created.

(3) When the application transmits the real-time message, the kernel creates instance(s) of the *sk_buff* structure to store the data of the message. We added a new field, fid, in *sk_buff* so that the flow id of the message can be copied and passed down to the network layer. In addition, we added another field, *release_time* of type *ktime_t* in *sk_buff*, to store the current system time as the message release time.

(4) When the network header structure *network_header* in *sk_buff* is constructed in the network layer, we use fid to search in the forwarding table (see Section 3.2) and calculate the latest transmission time for the current node and for the next node. Then, fid and the transmission times are stored in the network header as shown in Fig. 2. *Bit* 8 is set to 1 to indicate real-time messages. If fid is 0, *bit* 8 is set to 0.

(5) We implemented a delay queue, which provides the standard interfaces of the Linux traffic control queue disciplines [11], so that real-time messages can be transmitted at its latest transmission time as required by our routing algorithm. The delay queue contains two components. The first one is a linked-list based FIFO queue to store non real-time packets. For real-time packets, we utilize the *cb* field, the control buffer in *sk_buff*, to implement a linked list-based min-heap. This linked list-based implementation does not have a limit on the number of messages that can be queued in the min-heap, which an array-based implementation of min-heap would have.

(6) When the clock reaches the latest transmission time of a real-time message, the queue discipline moves the value of the latest transmission time for the next node (i.e., *bytes* $28-31$) to *bytes* $24-27$. Then, the message is forwarded to the network interface for transmission. In this way, when the message arrives at the next node, *bytes* $24-27$ denote the latest transmission time for the new node.

3.2 Forwarding Table Structure

We utilize the default forwarding table for background traffic. The default forwarding table contains the mapping between the destination network IDs (i.e., network destination and network mask) and the local interfaces that reach the destinations. For real-time messages, we use the flow IDs to indicate the destinations in the forwarding table. The forwarding table contains the expected message response time ($R[v]$) on the current node v. In addition, it contains the sum of the worst case message transmission variation $\Delta[v]$ on the current node, the propagation delay ($Pr[v, v']$) on the link to the next node v', and the expected message response time ($R[v']$) on the next node. Table 2 depicts the table structure, where *Next Aggregate Delay* is the sum of $\Delta[v]$, $Pr[v, v']$, and $R[v']$. $Interface$ is the interface to forward messages in each flow.

As a result, the time variables of real-time messages required by the scheduler (see Algorithm 1) can be calculated from the data carried in the header and stored in the forwarding table. Table. 3 depicts these relations.

Table 2 Forwarding Table.

fid	Expected Response Time (ms)	Next Aggregate Delay (ms)	Interface
1	7	10	eth0
2	12	16	eth1

Table 3 Message Scheduler Variables.

Variable	Source
$A[v]$	Message header ($bytes$ $24-27$)
$R[v]$	Forwarding table
$A[v']$	$A[v]$ + *Next Aggregate Delay* in forwarding table

3.3 Virtual Switch Implementation

We extend the traffic control [11] and the OpenFlow implementation in Open vSwitch [20, 5] on Linux to implement our message scheduler. Our virtual switch implementation includes three components: (1) An ingress queue based on the traffic control mechanism (i.e., the input queue in our node model). When this queue is enabled, the incoming packets on related interfaces are put into this queue. Meanwhile, the packet arrival events trigger the scheduler to invoke the *dequeue* function if the scheduler is idle. The *dequeue* function finds the packet with the earliest expected response time and sends it to the downstream OpenFlow actions (lines 4 to 19 of Algorithm 1). (2) A new flow table structure (see Section 3.2) is constructed for real-time messages based on the OpenFlow hierarchy and the corresponding forwarding actions to execute (in lines 20 to 31). The scheduler invokes the forwarding actions for every packet. It then forwards real-time messages to the downstream intermediate queue and forwards background traffic to the corresponding interface. (3) A delay queue based on the traffic control mechanism (i.e., the intermediate queue in our node model). This queue has the same structure as the delay queue at the end nodes (see Section 3.1). The scheduler forwards any real-time messages removed from the delay queue with the *dequeue* function for the corresponding interface.

The buffer to store packets on a virtual switch is shared by the ingress queue and the delay queue. When an interface attempts to put a background packet into the ingress queue by invoking the *enqueue* function, we check if the available buffer is sufficient for the packet. If it is not, that background packet is dropped. When an interface attempts to put a real-time message into the ingress queue and the available buffer is insufficient for that message, the *enqueue* function drops background packets that are already in the queue so that the real-time message can be stored. In addition, as one of the objectives of our routing algorithm (see the second criterion of the validation algorithm described in Section 2.3), the algorithm guarantees that the buffer always has enough space to store real-time messages if the forwarding path is schedulable. The size of the buffer is a configurable parameter in our implementation, which is set to different values as part of our experimental assessment.

4 Evaluation

We evaluate our virtual switch on a local cluster. The experiments are conducted on 6 nodes, each of which features a 2-way SMP with AMD Opteron 6128 (Magny Core) processors and 8 cores per socket (16 cores per node). Each node has 32 GB DRAM and Gigabit Ethernet.

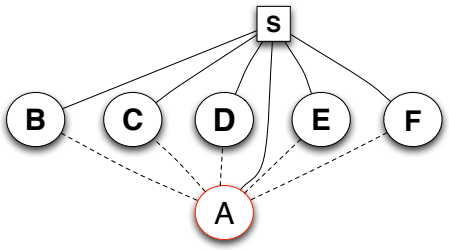


Figure 3 Experiment Network Setup (1).

These nodes are connected via a single network switch in the physical network topology, and we use different numbers of nodes as virtual SDN switch nodes in different experiments. Each node runs a modified version of Linux 2.6.32 and Open vSwitch 2.3.2, which includes the virtual switch implementation. The clocks on these nodes are synchronized with a centralized NTP server when we conduct the experiments. In the first part of this section, we present the experimental results to demonstrate the capabilities of the message scheduler in two aspects: (1) Under intense network congestion, the message scheduler on a single node can meet the hard deadline of every real-time message. (2) When the forwarding paths of real-time messages consist of multiple switches, our message schedulers on these switches can collaborate to guarantee end-to-end deadlines of all real-time messages. In the second part of this section, we present a demonstration for the routing algorithms.

4.1 Single Virtual SDN Switch Result

Background: To demonstrate the effectiveness of the packet scheduler, we measures the deadline miss rate for real-time messages and packet drop rate for background traffic on a single virtual SDN switch with different configurations.

All 6 nodes (marked as A to F in Fig. 3) are connected via a single physical switch (marked as S; physical links are indicated by solid lines). We use node A as the virtual switch. The traffic is transmitted through node A via the virtual links (dashed lines). To support this, a node generates test packets using the IP address of node A as the destination. When a test packet arrives at virtual switch A via the physical links, A uses the flow id carried in the packet header to determine its real destination. Thus, we add an extra column in the forwarding table to indicate the real destination of a message flow. In addition, the ToS field in background packets generated for test purposes is used to indicate their real destination. Virtual switch A fills the real destination of any packet into the IP header of the packet and then forwards it via the physical link to the central switch. In this way, test packets are transmitted to its destination via the virtual switch. This modification is due to the limitation of our experimental environment, which would be unnecessary if the virtual switch were deployed directly onto a physical switch.

We use the network benchmark Sockperf to generate test workloads. A Sockperf client transmits messages as UDP packets to the corresponding Sockperf server via the virtual switch. The client generates a log record to indicate the transmission time of a message while the server generates a log record to indicate the message arrival time. To simplify measurements, the data section for a real-time message contains the flow id and the sequence id of a particular message in that flow. The data section is padded by *0*s so that its total size is 1000 bytes. A packet has been dropped by the virtual switch if no corresponding record exists on the server side after the experiment. We assume that no packet drop occurs in

Table 4 Test Workload in Single Switch Experiment.

fid	Type	Source	Destination	(mps, burst)	Relative Deadline
1	Real Time	B	C	(250, 1)	24ms
2	Real Time	C	D	(200, 1)	24ms
3	Background	D	E	(200, 400)	NA
4	Background	E	F	(200, 400)	NA
5	Background	F	D	(200, 400)	NA

the network stack software of the end nodes or on the physical network links. Our platform meets this assumption since we do not limit the buffer size on end nodes and the physical network is reliable in our cluster.

Table 4 depicts the workload in the first experiment. The workload is controlled by Sockperf parameters $(mps, burst)$. mps indicates the number of frames per second, which affects the frame size. $burst$ defines the number of packets transmitted in each frame by the client. For example, $(250, 1)$ indicates 250 frames per second (i.e., a frame size of $4ms$) with 1 message transmitted per frame. Real-time message flows have fixed parameters to make them strictly periodic. The actual burst of a background flow is a random value uniformly drawn from the interval $[\frac{burst}{2}, burst]$ in each frame. If a real-time message does not arrive at the server (i.e., no corresponding record exists in the server logs), we consider it a deadline miss in this experiment. No re-transmission is performed. In addition, we calculate the end-to-end delay for real-time messages even if they arrive at the server side. If the delay of a message is longer than its deadline, we also consider it as a deadline miss. We set the expected response time on the virtual switch to $20ms$ for both real-time flows, which is the relative deadline of the message flows ($24ms$) minus the aggregate propagation delay and expected response time variations on the path (estimated as $4ms$). The case study in Section 4.3 demonstrates this calculation.

Analysis: Fig. 4 depicts the result. We adjust the buffer size and compare both deadline miss rate and packet drop rate when the message scheduler is turned on and off on the virtual switch. When the message scheduler is turned off, the virtual switch does not differentiate real-time packets from background packets and forwards them to the destination directly. The x-axis in Fig. 4 is the buffer size on the virtual switch. The y-axis is the packet drop rate for background traffic and the deadline miss rate for real-time flows.

The black and green lines depict the deadline miss rate for real-time messages when the scheduler is turned off and on, respectively. With $20ms$ as the expected response time, the message flows are schedulable under all buffer size configurations. Both message flows have a 0% deadline miss rate when the scheduler is on.

▶ **Observation 1.** *The packet scheduler can meet the deadline requirements of all real-time messages.*

The black and blue lines depict the deadline miss rate and packet drop rate when the scheduler is turned off. Since real-time packets and background packets are processed in the same way by the switch, the deadline miss rate and packet drop rate are close for all buffer size configurations. The virtual switch has an increasing tolerance to the burstiness of packets with a larger buffer size. As a result, both rates decrease when the buffer size is increased as depicted by the declining lines.

▶ **Observation 2.** *When the scheduler is off, a larger buffer size can decrease both the deadline miss rate for real-time messages and the packet drop rate for background traffic.*

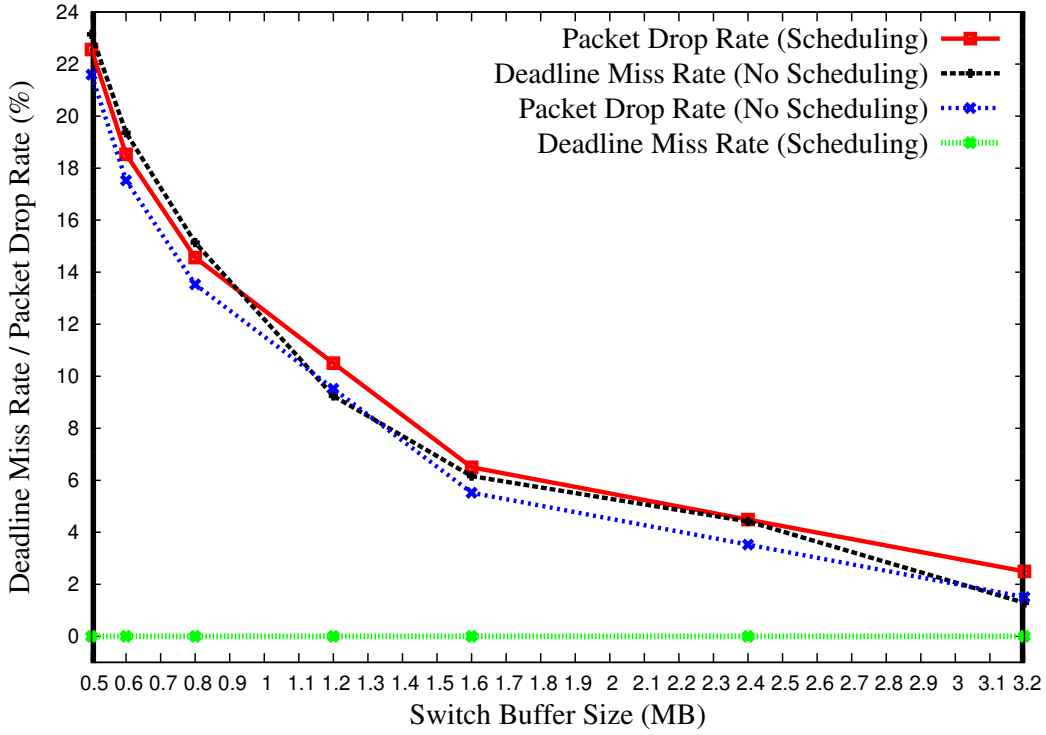


Figure 4 Deadline Miss Rate and Message Drop Rate.

Table 5 Expected Response Time vs Message Drop Rate.

Expected Response Time (ms)	Message Drop Rate (%)
10	10.80
20	10.89
40	11.05
80	11.46
160	11.81
320	12.33

When the scheduler is on, the expected message response time on the virtual switch is $20ms$, which means every real-time message is delayed in the switch buffer for $20ms$ (no delay is added in the end nodes where the real-time messages are released and received). Due to this delay, the available buffer size for background traffic shrinks. As a result, the packet drop rate for background traffic increases slightly, e.g., from 17.53% (blue line) to 18.53% (red line) for a buffer size of $0.6MB$. Table 5 shows an overall trend of an increasing drop rate for background traffic when the expected response time for real-time messages is increased from $10ms$ to $320ms$ for a buffer size of $1.2MB$. The results indicate that the longer real-time messages are delayed in the switch buffer, the smaller the available buffer is for background traffic. As a result, the packet drop rate increases when the delay time for real-time messages is increased.

▶ **Observation 3.** *Our scheduler increases the packet drop rate for background traffic.*

The measurement of end-to-end delays for real-time messages indicates that the message scheduler does not introduce a significant variation to the message transmission time (Δ is

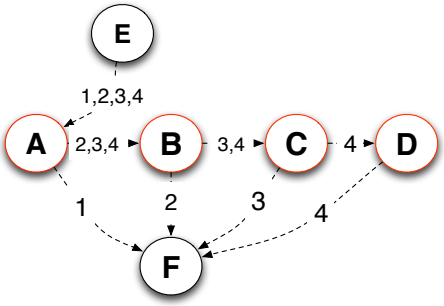


Figure 5 Experiment Network Setup (2).

small). The end-to-end delay includes the latency due to transmission from the source node to the virtual switch, the time the message spent on the virtual switch, and the transmission latency from the virtual switch to the destination node. When we set the buffer size to $1.2MB$ and the expected response time to $10ms$, the shortest end-to-end delay is $10.9ms$ and the longest is $12.1ms$ (the median value is $11.6ms$), of which $10ms$ are due to the software-induced delay on the virtual switch. Since messages are transmitted via a physical switch (and not via direct links in our system model), the physical switch also contributes to the delay variation.

4.2 Multiple Virtual Switches Result

Background: In the second experiment, we construct a virtual switch chain to transmit real-time messages. Nodes $A - D$ are configured as virtual switches with a buffer size of $1.2MB$. Node E uses a Sockperf client to transmit four message flows to the Sockperf server on node F. Fig. 5 depicts the forwarding path for each message flow. The periods of message flows are $1ms$. The expected response time is $5ms$ and the next aggregate delay is $7ms$ for each flow on these virtual switches. This setup suggests that the sum of the worst case response time variation, Δ, and the propagation delay, Pr, on the link between two virtual switches (i.e., two physical links connected via the central physical switch) is $2ms$. We measure the end-to-end delay for each message.

Our experimental results show that messages in all four flows experience a similar end-to-end delay variation ($[0.8ms, 1.8ms]$). The value of next aggregate delay includes the worst case response time variance on the previous node (as described in Eq. 3.2). The scheduler only considers a real-time message once the current time is greater or equal to the worst case response time (see line 9 in Algorithm 1). As a result, only the response time variation on the last virtual switch (plus the propagation delay variation on the physical link from the last virtual switch to the receiver in our experiment) contributes to the measured variation of the end-to-end delays even though the forwarding path has multiple virtual switches. This indicates the effectiveness of the message scheduler in terms of variance control.

▶ **Observation 4.** *Real-time messages do not accumulate response time variation (jitter) when forwarded by a chain of virtual switches.*

4.3 Routing Algorithm Demonstration

We demonstrate our routing algorithm to find forwarding paths for real-time message flows. Three message flows with attributes indicated in Table 6 are considered. These flows can

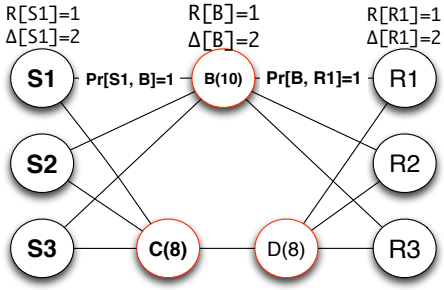


Figure 6 Routing Algorithm Demonstration.

Table 6 Real-time Message Flows in Demonstration.

fid	Source	Destination	Period (T)	Relative Deadline (D)	Message Size (S)
1	S1	R1	12	11	1
2	S2	R2	1	15	1
3	S3	R3	12	12	9

be either forwarded via intermediate switch B or switches C and D as depicted in Fig. 6, where the numbers in the switch circles are the buffer size of that switch. For simplicity, we consider that it takes $1ms$ for the message scheduler on every switch to process one message (i.e., $c = 1ms$ in Eq. 5). We consider a response time variation of $2ms$ on every node or switch and a propagation delay of $1ms$ ($Pr = 1ms$) on every link (i.e., $\Delta = 2ms$, $Pr = 1ms$).

We use the same notation as described in Section 2.3 to express forwarding path candidates. For example, candidate ((S1, 1), (B, 1), (R1, 1)) for flow 1 means that a real-time message of flow 1 is forwarded via nodes $S1$, B, and $R1$ with expected message response times of $1ms$, $1ms$, and $1ms$ on each node. We determine the schedulable routing paths for this setup in the following steps.

(1) Consider forwarding path candidate ((S1, 1), (B, 1), (R1, 1)) for flow 1 (see 1st row in of Table 7). The worst case end-to-end delay of this path candidate is $R[S1] + \Delta[S1] + Pr[S1, B] + R[B] + \Delta[B] + Pr[B, R1] + R[R1] + \Delta[R1] = 11ms$. Thus, this is the only forwarding path candidate for flow 1 according to the deadline constraint of Eq. 7. If the expected response time on any node of the path were increased, the end-to-end delay would exceed the relative deadline of flow 1, which is $11ms$.

The size of messages in flow 1 is 1. The maximum number of messages in flow 1 that could be on switch B at any time is $\lceil \frac{\Delta[S1]+R[B]+\Delta[B]}{T_1} \rceil = 1$. As a result, the residual buffer size of node B becomes 9, which is the buffer size of node B (10) minus the maximum buffer that could be occupied by messages in flow 1.

(2) The forwarding path candidates for flow 2 are shown with ID 2-6 in Table 7, where columns B, C, and D depict the residual buffer size on the corresponding nodes if flow 2 is forwarded. Candidates 2, 3, and 4 are considered first since the residual buffer size on these paths is 4 (i.e., $B - \lceil \frac{\Delta[S2]+R[B]+\Delta[B]}{T_2} \rceil * S_2 = 9 - 5 = 4$) if flow 2 is scheduled, which is larger than the candidates 5 and 6. However, switch B cannot schedule both flow 1 and flow 2, since the utilization of flow 2 is $\frac{c}{T_2} = 100\%$. Other path candidates for flow 2 with $R[B] = 1ms$, which are not shown in Table 7, are rejected for the same reason.

(3) Candidate 5 is considered next for flow 2 since it is a shorter path compared to candidate 6 even though they have the same residual buffer size on their paths. However, if

Table 7 Forwarding Path Candidates.

ID	Flow	Forwarding Path Candidate	End-to-end Delay	Residual Buffer[1)]			Result
				B	C	D	
1	1	(S1, 1), (B, 1), (R1, 1)	11	9	8	8	✔
2	2	(S2, 1), (B, 1), (R2, 1)	11	4	8	8	✕
3	2	(S2, 2), (B, 1), (R2, 1)	12	4	8	8	✕
4	2	(S2, 1), (B, 1), (R2, 2)	12	4	8	8	✕
5	2	(S2, 1), (B, 2), (R2, 1)	12	3	8	8	✕
6	2	(S2, 1), (C, 1), (D, 1), (R2, 1)	15	9	3	3	✔
7	3	(S3, 1), (B, 1), (R3, 1)	11	0	3	3	✔
8	3	(S3, 1), (B, 2), (R3, 1)	12	0	3	3	✕
9	3	(S3, 2), (B, 1), (R3, 1)	12	0	3	3	✕
10	3	(S3, 1), (B, 1), (R3, 2)	12	0	3	3	✕

1) After the corresponding flow is scheduled on the path.

flow 2 is transmitted via candidate 5, the residual buffer size of node B becomes 3. In this case, flow 3 has no forwarding path candidate since flow 3 requires that the buffer size of any node on the path is at least 9, the size of the message in flow 3, since the other forwarding path (via switches C and D) only has a residual buffer size of 8. Other path candidates for flow 2 with $R[B] \geq 2ms$, which are not shown in Table 7, are rejected for the same reason. Thus, the algorithm has to consider candidate 6 in Table 7, which is chosen.

(4) Candidates 7–10 for flow 3 meet the buffer size requirement and the relative deadline of flow 3. Candidate 7 has a higher preference since it has a shorter end-to-end delay. In addition, node B can schedule both flow 1 and 3, both with an expected response times of $1ms$. In summary, we found paths for all three flows.

5 Related Work

Our packet scheduler adopts per-node traffic control to guarantee transmission deadlines similar to other switched real-time Ethernet mechanisms, e.g., rate-controlled service disciplines (RCS) [8, 33], token-bucket traffic shaping [17], and EDF scheduling [9, 34]. Unlike these mechanisms, our scheduler considers multiple constraints imposed by network devices, namely link speed, computation speed, and buffer capacity. Table 8 details the comparison of the assumptions of these mechanisms. $defined$ in the table indicates that the corresponding mechanism considers the restriction in its model. Our work does not assume infinite buffer capacity and computation capacity. Our assumptions are more realistic on commodity switches connected by modern high speed links.

Past work has proposed different mechanisms to establish communication channels that recognize real-time requirements of packet flows at run time [7, 9, 26, 17]. In contrast, our work focuses on forwarding path planning (via a static routing algorithm) and forwarding policy enforcement (via packet scheduling). Other mechanisms guarantee soft deadlines of real-time packets while providing high throughput to other traffic [28], or adopt congestion avoidance algorithms when network contention occurs (e.g., buffer occupancy exceeds a certain threshold) [29]. Our work guarantees hard deadlines by dropping only non-realtime packets upon network congestion.

Our schedulability model does not depend on statistics as metrics of the network, e.g., bandwidth. Such metrics are dependent on multiple primitive factors of the switch: the

Table 8 Assumptions Comparison for Real-time Packet Schedulers.

Mechanism	Buffer Capacity	Computation Capability	Link Speed
RCS [8, 33]	infinite	infinite	defined
EDF [9, 34]	infinite	infinite	defined
Traffic shaping [17]	defined	infinite	defined
D^3 [29], D^2TCP [28]	defined	infinite	defined
Our work	defined	defined	defined

packet scheduling algorithm, buffer size, computing capability, and the state of other flows on the switch [10]. Instead, we have specifically considered these factors in two ways: (1) We formalize the dynamics of the network delay by introducing response time variations (δ) in our analysis and we aggressively control the variation by the message scheduler. (2) Our routing algorithm considers multiple constraints on switches and links when finding forwarding paths.

In contrast to TDMA-based real-time Ethernet channel implementations [4, 15], which rely on custom hardware to respond to control data frames, our scheduler is designed and implemented on Linux-compatible virtual switches and can be deployed on modern commodity SDN-compatible switches, which is more suitable for wide-area deployment.

Past work has studied routing algorithms to find packet forwarding paths with QoS support in different situations. This includes Dijkstra's algorithm to find the single-source path with shortest end-to-end delay under the assumption that per switch delay is known as priori [12], Suurballe's algorithm to find multiple disjoint paths with minimal total end-to-end delay [27], and methods to find multiple disjoint paths with the minimized delay of the shortest path [30]. Others have studied the routings to find an optimal forwarding path for one particular message flow under certain network assumptions [12]. Past work has also proposed to transmit messages over multiple paths simultaneously while the total time of the flow is constrained under the assumption that actual bandwidths are known [24]. Network Calculus has also been adopted to derive the forwarding path for a particular flow under the additional assumptions that no cross-over traffic exists or the pattern of cross-over traffic is known as priori [3]. Our work differs as follows from these prior work. It derives forwarding paths for multiple message flows to guarantee the hard deadline of every message. Our analysis depends on the link speed matrix, E, as described in Section 2. However, when the implementation is deployed on physical switches, E can be quantified by the medium speed, which is independent of the network traffic and scheduling algorithms. In addition, our routing algorithm belongs to the class of constraint-based path selection algorithms, where multiple constraints are considered on the transmission links [16]. We extend that by adding the constraint of switch buffers and formalizing the cost and evaluation functions for switch buffers and message scheduling, which are essential when applying any constraint-based path selection algorithm.

To find forwarding paths for multiple packets, past work has proposed to minimize the average packet delay [12]. However, the objective of real-time message transmission is to meet the deadline of each message flow (i.e., not to minimize the average delay). Our routing algorithm considers the hard deadline of every real-time flow when assigning a forwarding path.

6 Conclusion

We have presented a routing algorithm to determine forwarding paths for real-time message flows in a distributed computing environment and a scheduler to actively enforce a message forwarding policy on network devices. Our routing algorithm considers both the deadlines and the network resource demands of real-time messages. As a result, no real-time messages can be dropped due to network contention when handled by our message scheduler and no real-time messages miss their deadlines. We have implemented the scheduler on virtual switches and conducted experiments on a local cluster to prove the effectiveness of the scheduler. Our experimental results showed that deadline misses of real-time messages dropped to 0 when the message scheduler was turned on.

Future work includes porting the message scheduler implementation to physical network devices (e.g., physical switches). Modern SDN-compatible switches support customized packet forwarding protocols (e.g., OpenFlow), which could be utilized to run our virtual switch implementation based on Open vSwitch. Since physical switches have hardware that is dedicated to packet processing, we expect a better performance than the virtual switches. In addition, in the case where a set of real-time message flows cannot be scheduled on a network environment due to hardware limitations, we plan to extend our routing algorithm to produce suggestive information, e.g., the required increase in buffer size of a switch before the set of flows becomes schedulable. Furthermore, we plan to extend the routing algorithm to find disjoint forwarding paths for real-time messages to handle network device failures.

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 Karsten Albers and Frank Slomka. An event stream driven approximation for the analysis of real-time systems. In *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 187–195. IEEE, 2004.

3 Anne Bouillard, Bruno Gaujal, Sébastien Lagrange, and Éric Thierry. Optimal routing for end-to-end guarantees using network calculus. *Performance Evaluation*, 65(11):883–906, 2008.

4 Gonzalo Carvajal, Luis Araneda, Alejandro Wolf, Miguel Figueroa, and Sebastian Fischmeister. Integrating dynamic-tdma communication channels into cots ethernet networks. *IEEE Transactions on Industrial Informatics*, 12(5):1806–1816, 2016.

5 Martin Casado, Teemu Koponen, Daekyeong Moon, and Scott Shenker. Rethinking packet forwarding hardware. In *HotNets*, pages 1–6. Citeseer, 2008.

6 Catalyst Switch Architecture. URL: `http://www.cisco.com/networkers/nw03/presos/docs/RST-2011.pdf`.

7 Domenico Ferrari and Dinesh C. Verma. A scheme for real-time channel establishment in wide-area networks. *Selected Areas in Communications, IEEE Journal on*, 8(3):368–379, 1990.

8 Leonidas Georgiadis, Roch Guérin, Vinod Peris, and Kumar N. Sivarajan. Efficient network qos provisioning based on per node traffic shaping. *IEEE/ACM Transactions on Networking (TON)*, 4(4):482–501, 1996.

9 Hoai Hoang, Magnus Jonsson, Anders Kallerdahl, and Ulrik Hagström. Switched real-time ethernet with earliest deadline first scheduling-protocols and traffic handling. *Parallel and Distributed Computing Practices*, 5(1):105–115, 2002.

10 Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Multistage switches are not crossbars: Effects of static routing in high-performance networks. In *Cluster Computing, 2008 IEEE International Conference on*, pages 116–125. IEEE, 2008.

11 Bert Hubert, Thomas Graf, Greg Maxwell, Remco van Mook, Martijn van Oosterhout, P. Schroeder, Jasper Spaans, and Pedro Larroy. Linux advanced routing & traffic control. In *Ottawa Linux Symposium*, page 213, 2002.

12 Sushant Jain, Kevin Fall, and Rabin Patra. Routing in a delay tolerant network. In *SIGCOMM'04*, pages 145–158. ACM, 2004.

13 Kevin Jeffay, Donald F Stanat, and Charles U. Martel. On non-preemptive scheduling of period and sporadic tasks. In *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 129–139. IEEE, 1991.

14 D. D. Kandhlur, Kang G. Shin, and Domenico Ferrari. Real-time communication in multi-hop networks. *Parallel and Distributed Systems, IEEE Transactions on*, 5(10):1044–1056, 1994.

15 Hermann Kopetz and Günter Grünsteidl. TTP-A time-triggered protocol for fault-tolerant real-time systems. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers, The Twenty-Third International Symposium on*, pages 524–533. IEEE, 1993.

16 Fernando Kuipers, Piet Van Mieghem, Turgay Korkmaz, and Marwan Krunz. An overview of constraint-based path selection algorithms for QoS routing. *IEEE Communications Magazine, 40 (12)*, 2002.

17 Jork Loeser and Hermann Haertig. Low-latency hard real-time communication over switched ethernet. In *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 13–22. IEEE, 2004.

18 David L. Mills. Internet time synchronization: the network time protocol. *Communications, IEEE Transactions on*, 39(10):1482–1493, 1991.

19 Seyedbehzad Nabavi, Jianhua Zhang, and Aranya Chakrabortty. Distributed optimization algorithms for wide-area oscillation monitoring in power systems using interregional pmu-pdc architectures. *Smart Grid, IEEE Transactions on*, 6(5):2529–2538, 2015.

20 Ben Pfaff, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. Extending networking into the virtualization layer. In *Hotnets*, 2009.

21 T. Qian, F. Mueller, and Y. Xin. A real-time distributed hash table. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*, pages 1–10. IEEE, 2014.

22 Tao Qian, Aranya Chakrabortty, Frank Mueller, and Yufeng Xin. A real-time distributed storage system for multi-resolution virtual synchrophasor. In *Power & Energy Society General Meeting*. IEEE, 2014.

23 Tao Qian, Frank Mueller, and Yufeng Xin. Hybrid EDF Packet Scheduling for Real-Time Distributed Systems. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 37–46, July 2015.

24 Nageswara SV Rao and Stephen G Batsell. QoS routing via multiple paths using bandwidth reservation. In *INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 11–18. IEEE, 1998.

25 Jakob Rosen, Alexandru Andrei, Petru Eles, and Zebo Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 49–60. IEEE, 2007.

26 Rui Santos, Moris Behnam, Thomas Nolte, Paulo Pedreiras, and Luís Almeida. Multi-level hierarchical scheduling in ethernet switches. In *Proceedings of the ninth ACM international conference on Embedded software*, pages 185–194. ACM, 2011.

27 John W. Suurballe and Robert Endre Tarjan. A quick method for finding shortest pairs of disjoint paths. *Networks*, 14(2):325–336, 1984.

28 Balajee Vamanan, Jahangir Hasan, and T. N. Vijaykumar. Deadline-aware datacenter tcp (d2tcp). *ACM SIGCOMM Computer Communication Review*, 42(4):115–126, 2012.

29 Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. *ACM SIGCOMM Computer Communication Review*, 41(4):50–61, 2011.

30 Dahai Xu, Yang Chen, Yizhi Xiong, Chunming Qiao, and Xin He. On finding disjoint paths in single and dual link cost networks. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 1. IEEE, 2004.

31 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 299–308. IEEE, 2012.

32 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), 2013 IEEE 19th*, pages 55–64. IEEE, 2013.

33 Hui Zhang and Domenico Ferrari. Rate-controlled service disciplines. *J. High Speed Networks*, 3(4):389–412, 1994.

34 Kai Zhu, Yan Zhuang, and Yannis Viniotis. Achieving end-to-end delay bounds by edf scheduling without traffic shaping. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1493–1501. IEEE, 2001.

Write-Back Caches in WCET Analysis*

Tobias Blaß[1], Sebastian Hahn[2], and Jan Reineke[3]

1 **Saarland Informatics Campus, Saarland University, Saarbrücken, Germany**
`s9toblas@stud.uni-saarland.de`
2 **Saarland Informatics Campus, Saarland University, Saarbrücken, Germany**
`sebastian.hahn@cs.uni-saarland.de`
3 **Saarland Informatics Campus, Saarland University, Saarbrücken, Germany**
`reineke@cs.uni-saarland.de`

Abstract

Write-back caches are a popular choice in embedded microprocessors as they promise higher performance than write-through caches. So far, however, their use in hard real-time systems has been prohibited by the lack of adequate worst-case execution time (WCET) analysis support.

In this paper, we introduce a new approach to statically analyze the behavior of write-back caches. Prior work took an "eviction-focussed perspective", answering for each potential cache miss: May this miss evict a dirty cache line and thus cause a write back? We complement this approach by exploring a "store-focussed perspective", answering for each store: May this store dirtify a clean cache line and thus cause a write back later on?

Experimental evaluation demonstrates substantial precision improvements when both perspectives are combined. For most benchmarks, write-back caches are then preferable to write-through caches in terms of the computed WCET bounds.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

Keywords and phrases write-back caches, real-time systems, WCET analysis, cache analysis

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.26

1 Introduction

The timely execution of programs is critical for hard real-time systems. Static worst-case execution time (WCET) analysis provides upper bounds on programs' execution times in all possible execution scenarios. These upper bounds can then be used to verify that all timing constraints of a given system are met prior to deployment in the field.

The execution time of a program heavily depends on the state of the underlying hardware platform, in particular on the state of caches, which are intended to bridge the gap between slow main memory and fast cores. WCET analysis has to precisely account for cache behavior to obtain useful time bounds.

Multiple parameters affect a cache's behavior and thus the latency of memory accesses, e.g. its capacity, associativity, block size, and replacement policy. A parameter that has so far received little attention in the literature is the write policy which determines the handling of write memory accesses. There are two common choices for the write policy: write through and write back. In a write-through cache, upon a store, the data is written to main memory directly. In a write-back cache, the data is only written to the cache and the corresponding

* This work was partially supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Project PEP, and by the Saarbrücken Graduate School of Computer Science which receives funding from the DFG as part of the Excellence Initiative of the German Federal and State Governments.

cache line is marked *dirty*. Once a dirty cache line is evicted from the cache its data is written back to main memory. In this way, multiple stores to the same cache line may be consolidated into a single costly main-memory access. Due to the possible performance benefits [17, 18], many embedded systems employ caches following the write-back policy, e.g. in the MPC603e [10] and the ARM946 [3] processor.

Write-back caches are commonly considered hard to analyze because write backs are decoupled in time from the corresponding stores in the program. Most literature on cache analysis is targeted at caches following the write-through policy. We are aware of only two exceptions: Alt et al. [1] and Sondag and Rajan [27] both introduce static analyses to safely approximate the set of dirty cache lines at each program point. Upon a potential cache miss, WCET analysis can use this information to determine whether or not the evicted cache line may be dirty and thus trigger a write back. As this approach tries to exclude potential write backs at cache evictions, we term this approach "eviction-focussed write-back analysis".

In this paper, we identify a complementary approach, which we term "store-focussed write-back analysis". The approach is based on two simple observations:

- each write back is preceded by a store to the cache line written back, and
- stores to dirty cache lines do not increase the number of write backs.

Thus we introduce an analysis that identifies and bounds the number of "dirtifying stores", i.e. stores to previously clean cache lines. We demonstrate at the hand of examples and later experimentally that the store-focussed approach is required to obtain good execution time bounds for caches following the write-back policy.

Our main contributions are the following:

- We introduce the "store-focussed write-back analysis" approach and the corresponding *dirtifying store* analysis, which is essential to turn write-back caches into a favorable choice for hard real-time systems.
- We present the first experimental evaluation of *both* eviction- and store-focussed write-back analyses. In particular, we demonstrate that with our new analysis write-back caches are preferable to write-through caches in terms of WCET bounds.

2 Background: Write-back Caches

Caches are fast but small memories that store a subset of the main memory's contents to bridge the latency gap between the CPU and main memory. To profit from spatial locality and to reduce management overhead, main memory is logically partitioned into a set of *memory blocks* $\mathcal{B}$. Each block is cached as a whole in a cache line of the same size.

When accessing a memory block, the cache logic has to determine whether the block is stored in the cache ("cache hit") or not ("cache miss"). For efficient look-up, each block can only be stored in a small number of cache lines known as a *cache set*. A subset of the bits of a memory block's address determines the cache set it maps to. The cache is partitioned into equally-sized cache sets. The size of a cache set in blocks is called the *associativity* k of the cache.

Since the cache is much smaller than main memory, a *replacement policy* must decide which memory block to replace upon a cache miss. Importantly, almost all replacement policies treat sets independently, so that accesses to one set do not influence replacement decisions in other sets. Well-known replacement policies are least-recently-used (LRU), used, e.g., in various Freescale processors such as the MPC603e and the TriCore17xx; pseudo-LRU (PLRU), a cost-efficient variant of LRU; and first-in first-out (FIFO). In this article we focus

Listing 1 Motivation for Store-focussed Write-back Analysis

```
x = f(x);
sum = 0;
for (i=0; i<N; i++) {
    sum += arr[read_sensor()];
}
...
```

exclusively on LRU. The application of our ideas to other policies is left as future work. As the name suggests, LRU replaces the least-recently-used block upon misses.

LRU naturally gives rise to a notion of *ages* for memory blocks: The age of a block b is the number of pairwise different blocks that map to the same cache set as b that have been accessed since the last access to b. If a block has never been accessed, its age is k. Then, a block is cached if and only if its age is less than the cache's associativity k.

Given this notion of ages, the state of an LRU cache can be modeled by a mapping that assigns to each memory block its age, where ages are truncated at k, i.e., we do not distinguish ages of uncached blocks. So the set of cache states is $age : \mathcal{B} \to \{0, \ldots, k\}$. Then, the effect of an access to memory block x can be formalized as follows:

$$update(age, x) \triangleq \lambda b. \begin{cases} 0 & \text{if } x = b \\ age(b) & \text{else if } age(x) \leq age(b) \vee set(x) \neq set(b) \\ \min(k, age(b) + 1) & \text{else if } age(x) > age(b) \wedge set(x) = set(b) \end{cases} \quad (1)$$

where $set(x)$ denotes the cache set that x maps to.

There are several choices regarding the implementation of stores, determined by the answers to the following two questions:

1. When is the data written back to main memory?
   One option, known as *write through*, is to perform any store immediately in main memory. The other option, known as *write back* is to buffer the changes in the cache, marking the modified cache line as dirty. Upon an eviction of a dirty cache line its data is then written back to main memory.
2. What happens upon a write miss?
   If the memory block being modified is not contained in the cache, one can either bypass the cache and make the modification in main memory (*no write allocate*) or one can allocate a cache line and write to that line (*write allocate*).

Write-back caches usually employ *write allocate*, and write-through caches usually employ *no write allocate*. In the remainder of the paper we will only state whether a cache is write through or write back and assume the usual allocation policy.

For a write-through cache, the LRU update defined in (1) does not cover the *store miss* case; then the update is simply the identity function; assuming that the non write allocate policy is applied in write-through caches.

3 Motivating Examples

Consider the example program in Listing 1. For the sake of readability, we use C-style example programs, while the analysis is performed at the level of machine instructions. Assume that all variables, i.e., `x`, `sum`, and `i`, are kept in separate memory blocks rather than

Listing 2 Motivation for Eviction-focussed Write-back Analysis

```
if (...)
    x = 0;
if (...)
    x = 1;
b = c;
```

in registers, and that `N` is a constant held in a register. For simplicity, let us analyze the write-back behavior of this example on a fully-associative cache of size 4.

The address accessed by `arr[read_sensor()]` cannot be predicted statically as it depends on sensor readings, which only become available at runtime. In particular, it is impossible to determine whether the accessed memory block is cached or not, assuming the array is larger than the cache. Then, each of the array accesses may result in a miss and may thus potentially trigger a write back.

After the first access to the array within the loop, the variable `x` is stored in the least-recently-used cache line, as `sum`, `i`, and `arr[read_sensor()]` have been used more recently. Due to the store `x = f(x)` the cache line holding `x` is dirty. Depending on whether the following iterations access the same cache line as the first loop iteration, `x` is evicted or not. Applying the eviction-focussed approach [1, 27] it is impossible to exclude a write back in any of the loop's iterations but the first. Such an analysis would thus have to assume at least $\mathtt{N} - 1$ write backs.

Now let us adopt a store-focussed view: `x` is written to only once, and so there may be at most one corresponding write back. However, simply using the number of stores in the example program is also not beneficial: including the assignments to `i` and `sum` in the loop, there are $2\mathtt{N} + 2$ stores in the program, which is even greater than the number of write backs an eviction-focussed analysis would derive. Besides, simply counting the number of stores defeats the purpose of write-back caches, which is to consolidate multiple stores to the same cache line before writing the data back to main memory.

To account for this mechanism, we introduce the notion of *dirtifying stores*. A dirtifying store is a store to a clean cache line. The number of dirtifying stores is a bound on the number of write backs. In the example, the variable `sum` is first written to in line 2, turning its cache line dirty. All the stores to `sum` inside the loop are non-dirtifying: as `sum` is accessed in every loop iteration, it may never be evicted and thus it remains dirty throughout the entire loop execution. Similarly, the first store to `i` in the loop is dirtifying, but all subsequent stores are non-dirtifying, as `i` remains cached throughout.

We therefore conclude that as a consequence of this program's execution at most three write backs may occur, namely of `x` (committing the store from line 1), `i` (committing the stores in line 3) and `sum` (committing the stores in line 2 and 4). In Section 6, we introduce an approach to bound the number of dirtifying stores.

Although the store-focussed approach often works very well, it is not always superior to the eviction-focussed approach. Consider the example in Listing 2, which includes two potentially dirtifying stores to variable `x`. Assuming a fully-associative cache of size 2, an eviction-focussed approach may recognize that `x` can only be evicted when accessing `b` (following the access to `c`) and can therefore safely account for a single write back. For optimal results, both approaches should therefore be combined. In Section 5, we describe an eviction-focussed write-back analysis, including a dirtiness analysis, which is also required to precisely bound the number of dirtifying stores in Section 6.

4 Background: Static WCET and Cache Analysis

4.1 Static WCET Analysis

In this section, we briefly present a state-of-the-art approach to WCET analysis, which our write-back analysis is based upon. It consists of three phases: (1) value analysis, (2) microarchitectural analysis, and (3) path analysis. All analyses are performed on binary-level machine programs.

The first phase performs analyses that are independent of the underlying microarchitecture. Examples are value analyses such as constant propagation or interval analysis. The main purpose of this phase is to compute loop bounds and addresses of load and store instructions. This information is used in the following phases.

The second phase analyzes the program at the microarchitectural level. The microarchitectural analysis calculates a *state graph*. Each node in the state graph contains the state of the microarchitecture including the pipeline and the cache. Nodes are connected via edges, which represent the execution behavior of the system over time. Each edge is weighted with its transition time in clock cycles, which we refer to as $time(e)$. Additional weights may be defined if required, e.g. modeling that a loop back edge is taken, which can be constrained by a corresponding loop bound, or that a write back occurs when an edge is taken.

Unfortunately, enumerating all reachable concrete microarchitectural states is infeasible; there are simply too many. It is therefore necessary to construct the state graph using *abstract* states. Each abstract state corresponds to many concrete states at once, thereby reducing the state space. Such an abstraction comes with two functions: the *update* function and the *join* function.

- The *update* function computes the successor of an abstract state. For a correct analysis, this function has to be consistent with the concrete transition function.
- The *join* function merges two abstract states into one. The resulting state must represent all concrete states represented by its arguments, but it may contain more. The ensuing imprecision is the price for the reduced state space.

Due to the uncertainty within the abstract states, there might be multiple successor states, e.g. one for the cache hit case and one for the cache miss case. This uncertainty is represented by nondeterministic choices inside the graph, i.e. multiple edges originating from the same state.

Finally, in the path-analysis phase, the worst-case path through the state graph is determined. To do so, the state graph is encoded via linear constraints and integer linear programming (ILP) is used to determine the path through the state graph with the greatest execution time. More precisely, a *frequency variable* f_e is introduced for each edge e of the state graph, modeling how often edge e is taken in an execution. The structure of the graph is encoded via *constraints*, which restrict the valuations of the frequency variables to those corresponding to valid paths through the state graph: in particular, the sum of the frequencies of incoming edges needs to be equal to the sum of the frequencies of outgoing edges at each node of the state graph. The results of the loop bound analysis in the first phase are used to further restrict the possible paths via *loop bound constraints*. A bound on the program's execution time can then be obtained via the following objective function: $\max \sum_{\text{edge } e} time(e) \cdot f_e$.

Further constraints can be added to the ILP to exclude infeasible execution paths. We use such constraints in the store-focussed write-back analysis introduced in Section 6.

Listing 3 Motivation: Persistence analysis

```
for (int i=0; i<N; i++) {
    k = read_sensor();
    sum[k] = sum[k] + arr[k];
}
```

4.2 Cache Analysis

The goal of cache analysis is to statically prove that a memory access hits or misses the cache. Here, we limit our exposition to a brief summary of the cache analyses required in order to understand our work. A more complete and detailed overview can be found in [22].

May and Must Analysis

In order to predict cache hits and misses LRU cache analyses compute upper and lower bounds on the *ages* of memory blocks. The age of memory block b is the number of distinct memory blocks mapping to the same cache set as b accessed since the last access to b. A block is in the cache if and only if its age is less than the cache's associativity k.

Thus, upper bounds on ages may be used to predict cache hits and lower bounds may be used to predict cache misses. The two analyses computing upper and lower bounds on ages are commonly known as must and may analysis [1]. Abstract states for both analyses map blocks to their respective bounds: $\mathcal{S}_{must} = \mathcal{S}_{may} = \mathcal{B} \to \{0, \ldots, k\}$.

The two functions $update_{must}$ and $update_{may}$ define the successor of an abstract cache state when loading or storing to x. For reasons of brevity we only define $update_{must}$ here:

$$update_{must}(must, x) \triangleq \lambda b. \begin{cases} 0 & \text{if } x = b \\ must(b) & \text{else if } must(x) \leq must(b) \vee set(x) \neq set(b) \\ \min(k, must(b) + 1) & \text{else if } must(x) > must(b) \wedge set(x) = set(b) \end{cases}$$

where $set(x)$ denotes the cache set that x maps to.

As in the concrete case, for a write-through cache, the above update does not cover the *store miss* case; then the identity function is applied instead. Since the must analysis maintains upper bounds, the join function takes the maximum of both bounds: $join(S, T) = \lambda b.\ \max(S(b), T(b))$. Similarly the join for the may analysis takes the point-wise minimum of the bounds.

Persistence Analysis

May and must analysis try to classify memory accesses at a given program point as cache hits or cache misses under all circumstances. In some cases, such a classification is impossible even though the cache is highly effective. Consider the array *sum* in Listing 3. Assuming all memory blocks accessed inside the loop completely fit into the cache, none of the cache lines of *sum* can be evicted within this loop once they have been loaded. They *persist* in the cache. However, must analysis is unable to prove this assuming each call of `read_sensor` may deliver an arbitrary value within the bounds of the arrays: each access may be the first to its cache line.

One approach to persistence analysis is the *conflict-set* analysis (also called conflict counting in [6]). The conflict-set analysis simply accumulates all memory blocks that may

have been accessed. If all of these blocks simultaneously fit into the cache, then all of them can safely be classified as *persistent*. There are more sophisticated persistence analyses described in [6], however, they offer little additional precision in practice, and thus we make use of the simple conflict-set analysis in this work.

For the example above, the conflict-set analysis computes a conflict set containing all blocks of `sum` and `arr` as well as `i` and `k`. Since this set fits entirely into the cache, all accesses are classified as persistent.

If a memory block is persistent, accesses to this block may only result in a single cache miss during program execution. This property can be encoded as a linear constraint, limiting the execution paths explored to those in which the respective block causes at most one miss. If an entire array is persistent, a similar constraint can be formulated to capture that the sum of the misses upon accesses to the array is bounded by the number of cache lines the array occupies.

Applying persistence analysis globally to large programs is bound to fail as memory blocks are rarely persistent throughout the entire program execution. Instead, persistence analysis is usually performed on smaller, contiguous parts of the program called *persistence scopes*. A common choice for persistence scopes, which we adopt in our experiments, are loops – they are executed more than once and they often reuse memory blocks across iterations. Persistence scopes can be nested, such that a block might be persistent in an inner loop but not persistent in the surrounding loop.

5 Eviction-focussed Write-back Analysis

In this section, we describe a simple eviction-focussed write-back analysis. The aim of this analysis is to determine for each potential cache miss in the program whether the cache miss may evict a dirty cache line and thus cause a write back.

Without further information, the analysis has to assume that every cache miss evicts a dirty cache line. To provide useful bounds, the write-back analysis has to track which memory blocks may be dirty. If the analysis can prove that all of the memory blocks that a cache miss might evict are clean, then no write back may occur.

5.1 Formalizing the Behavior of Write-back Caches

Before defining an abstraction for tracking the dirtiness of memory blocks, let us formalize the concrete behavior of a write-back cache. To this end, the dirtiness of memory block is represented by a predicate $dirty : \mathcal{B} \to \{C, D\}$, where C stands for "clean" and D for "dirty". Concrete cache states S then are pairs consisting of $S.age : \mathcal{B} \to \{0, \ldots, k\}$ and $S.dirty : \mathcal{B} \to \{C, D\}$, which together map each block to its age and its dirtiness status.

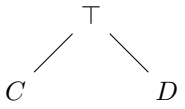
The update of ages has been defined in (1). The update of the dirtiness predicate depends on whether the access is a load or a store. Let us first consider the load case:

$$update_{load}(S.dirty, x) \triangleq \lambda b. \begin{cases} C & \text{if } evicts(S.age, x, b) \\ S.dirty(b) & \text{otherwise} \end{cases} \tag{2}$$

where $evicts(S.age, x, b)$, defined below, captures that the access to x causes the eviction of memory block b. If the access to x causes the eviction of block b (case 1), then b becomes clean. Otherwise, b's dirtiness remains the same (case 2).

The access to x evicts memory block b from the cache if they map to the same cache set, the access to x causes a miss, and b is evicted:

$$evicts(S.age, x, b) \triangleq set(x) = set(b) \wedge S.age(x) = k \wedge S.age(b) = k - 1.$$



⊔	C	D	⊤
C	C	⊤	⊤
D	⊤	D	⊤
⊤	⊤	⊤	⊤

Figure 1 Hasse diagram of the partial order on dirtiness states and the corresponding join function on dirtiness states.

In case of a store, the block that is written to becomes dirty. Other blocks may become clean if they get evicted, as modeled by the load update:

$$update_{store}(S.dirty, x) \triangleq \lambda b. \begin{cases} D & \text{if } x = b, \\ update_{load}(S.dirty, x)(b) & \text{otherwise}. \end{cases} \tag{3}$$

An access to block x causes a write back if and only if it evicts a dirty block:

$$writeback(S, x) \triangleq \exists b \in \mathcal{B}.\ S.dirty(b) = D \wedge evicts(S.age, x, b)\,.$$

5.2 Dirtiness Analysis

Dirtiness analysis [1, 27] tracks the dirtiness of memory blocks. During analysis, each block can be in one of three dirtiness states:

1. clean (C), meaning the block is definitely clean,
2. dirty (D), meaning the block is definitely dirty, and
3. unknown ($\top$), meaning the block might be either clean or dirty.

The three dirtiness states are partially ordered according to the Hasse diagram in Figure 1. The domain of the overall dirtiness analysis is thus $\mathcal{B} \to \{C, D, \top\}$ mapping each block to its dirtiness state.

To specify the abstract update function of the dirtiness analysis, we need to know when a memory block *might get evicted* from the cache and when a block *has definitely been evicted* from the cache. We define both predicates based on the information from the must and the may cache analysis. The must analysis tells us the *earliest* point in time when a block may be evicted, while the may analysis tells us the *latest* point in time when a block may be evicted. If there is any uncertainty about a block's exact age we obtain an interval of points in time at which an eviction may happen.

We use $S.may$ and $S.must$ to refer to the state of the may and must analysis *before* the current update. The predicate $may\text{-}evict(S, x, b)$ determines whether accessing x might evict b from cache state S:

$$\begin{aligned} may\text{-}evict(S, x, b) \triangleq\ & set(x) = set(b) \wedge S.must(x) = k \\ & \wedge S.may(b) < k \wedge update_{must}(S.must, x)(b) = k\,. \end{aligned}$$

Paraphrasing the formula above, block b might be evicted, if the access to x may result in a miss ($S.must(x) = k$) and b may have been cached prior to the access ($S.may(b) < k$), and b may be out of the cache after the access ($update_{must}(S.must, x)(b) = k$).

The predicate $\mathit{evicted}(S, x, b)$ determines whether block b has definitely been evicted from the cache after accessing x[1]:

$$\mathit{evicted}(S, x, b) \triangleq \mathit{update}_{\mathit{may}}(S.\mathit{may}, x)(b) = k\,.$$

Using the above predicates, we can define the abstract dirtiness update function for loads:

$$\widehat{\mathit{update}_{\mathit{load}}}(S, x) \triangleq \lambda b.\ \begin{cases} C & \text{if } \mathit{evicted}(S, x, b), \\ S(b) \sqcup C & \text{else if } \mathit{may\text{-}evict}(S, x, b), \\ S(b) & \text{otherwise}\,. \end{cases} \tag{4}$$

If a block must have been evicted (case 1, above) it is definitely clean. If it may have been evicted, it may be clean (case 2) after the access. If it was clean before $S(b) \sqcup C = C$, otherwise $S(b) \sqcup C = \top$ and the dirtiness status of b is unknown. If b may not have been evicted by the access to x, then its dirtiness status does not change (case 3).

Here, it becomes evident that the precision of the dirtiness analysis depends on the precision of may and must analysis: The more uncertainty there is about when blocks are evicted, due to uncertainty about blocks' ages, the more uncertainty there is about their dirtiness status.

Upon stores, dirtiness is updated as follows:

$$\widehat{\mathit{update}_{\mathit{store}}}(S, x) \triangleq \lambda b.\ \begin{cases} D & \text{if } x = b, \\ \widehat{\mathit{update}_{\mathit{load}}}(S, x)(b) & \text{otherwise}\,. \end{cases} \tag{5}$$

The block that is stored becomes dirty (case 1). The effect on other blocks is the same as in case of a load, and thus the update function for loads can be applied for those blocks (case 2). This closely matches the concrete update defined in (3).

Dirtiness analysis states are joined by applying the join depicted in Figure 1 to the dirtiness of each memory block:

$$\mathit{join}(S.\mathit{dirty}, T.\mathit{dirty}) \triangleq \lambda b.\ (S.\mathit{dirty}(b) \sqcup T.\mathit{dirty}(b)).$$

Based on a dirtiness analysis state we define the *may-wb* predicate, which determines whether an access might induce a write back:

$$\mathit{may\text{-}wb}(S, x) \triangleq \exists b \in \mathcal{B}.\ S.\mathit{dirty}(b) \neq C \wedge \mathit{may\text{-}evict}(S, x, b)\,. \tag{6}$$

This is the case, if a block may be evicted that is not guaranteed to be clean.

Initial State

The results produced by our WCET analysis may serve as inputs to a later schedulability analysis that checks whether a set of tasks can be scheduled on a given platform. Such an analysis for fixed-priority scheduling and platforms featuring write-back caches has recently been proposed by Davis et al. [7]. The analysis by Davis et al. accounts for the effects of initially dirty cache blocks and additional effects due to preemptions. Using this schedulability analysis, the WCET analysis can safely assume an initially clean cache, i.e. the proposed dirtiness analysis may start from the initial state $\lambda b.\, C$. If required, a potentially dirty initial cache can also be modeled by the initial state $\lambda b.\, \top$.

[1] We do not call this predicate *must-evict*, because it also captures cases where b may have been evicted earlier, and is thus not guaranteed to be evicted by the current access.

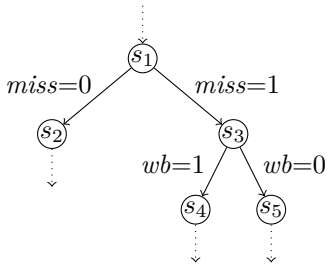


Figure 2 State graph upon cache-state uncertainty.

5.3 Integration into Microarchitectural Analysis

Microarchitectural analysis constructs the state graph that represents the cycle-level execution behavior of a program, as described in Section 4.1. The nodes in the state graph are abstract microarchitectural states. These abstract states encompass the state of the pipeline, the state of the memory controller, and the state of the caches including the abstract dirtiness state introduced above.

Assume memory block x is accessed during a cycle transition from abstract state S. If the may and must analysis cannot classify the access as cache hit or miss the analysis creates a successor state for both cases, one for the hit and one for the miss case. In general, both cases need to be considered due to *timing anomalies* [21, 25].

If a write-back cache is employed, the cache line loaded upon the cache miss will evict another cache line. The evicted cache line may be either dirty, which causes a write-back, or clean. Naively, the microarchitectural analysis would follow both cases. However, the analysis can use the *may-wb* predicate to rule out that a write back happens. If *may-wb* is false, no write back can happen and only one case needs to be considered. If *may-wb* is true, again both cases need to be considered due to the possibility of timing anomalies.

An excerpt of a state graph that arises due to cache and dirtiness uncertainty is depicted in Figure 2. After a few cycle transitions, the successors of s_2, s_4, and s_5 are likely to converge to a similar state. In that case, these successor states are joined into a single analysis state to keep the complexity of the microarchitectural analysis at an acceptable level.

6 Store-focussed Write-back Analysis

As we have shown by example in Section 3, it can be beneficial to bound the number of write backs by analyzing the stores that may occur during program execution rather than the evictions, which eventually trigger write backs. While stores are trivial to locate, the exact position of the write back of a particular dirty memory block depends on the cache state and thereby on the history of the program. Uncertainty about the program flow therefore affects eviction-focussed analyses more than store-focussed ones.

The simplest store-focussed analysis only exploits the fact that a write-back cache defers stores but never generate additional ones. The number of write backs is therefore bounded by the number of stores.

This property, which we call the *store bound*, can be expressed in the path analysis ILP formulation. As can be seen in Figure 2, microarchitectural analysis annotates transitions in the state graph with multiple edge weights, including $wb(e)$ and $st(e)$. $wb(e)$ denotes the number of write backs on edge e in the state graph, and $st(e)$ denotes the number of stores on edge e. Usually, these edge weights are either 0 or 1 for a given edge, but in principle

Listing 4 Listing 1 revisited. The dirtiness analysis state is shown on the right. Dirtifying stores are underlined.

```
x = f(x);                                          x ↦ D, sum ↦ C, i ↦ C
sum = 0;                                           x ↦ D, sum ↦ D, i ↦ C
for (i=0; i<N; i++) {
    sum += arr[read_sensor()];                     x ↦ ⊤, sum ↦ D, i ↦ D
}
...
```

they can be larger if several consecutive edges are merged for efficiency reasons. Given these edge weights, the following linear constraint corresponds exactly to the fact that the number of stores bounds the number of write backs:

$$\sum_{\text{edge } e} wb(e) \cdot f_e \leq \sum_{\text{edge } e} st(e) \cdot f_e, \tag{7}$$

where f_e is the frequency variable that captures how often edge e is taken.

With this constraint, path analysis implicitly only considers executions in which the number of write backs is bounded by the number of stores. We note, however, that the constraint does not exclude those infeasible executions in which write backs occur *before* stores. This limitation appears difficult to eliminate without generating a much larger ILP.

6.1 Dirtifying Stores

Simply bounding the number of write backs by the number of stores may yield unsatisfactory results. In particular, the store bound never predicts fewer memory accesses than would occur in a system with a write-through cache. The crucial advantage of write-back caches is the ability to consolidate multiple stores into a single write back. An effective analysis has to capture this behavior. Approaching the problem from a store-focussed perspective, this means that a write-back analysis has to recognize whether a store targets a clean cache line. We call such stores *dirtifying*, since they cause a cache line to become dirty.

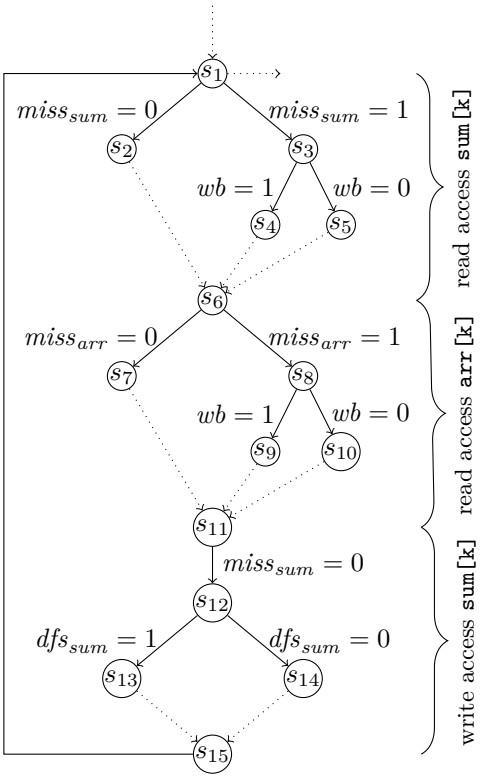
The dirtifying store analysis is based on the observation that only dirtifying stores may cause write backs. After the first store to a cache line, this line has to be written back no matter what, so additional stores to the cache line do not increase the future number of write backs. We use the dirtiness analysis described in Section 5.2 to identify definitely dirty cache lines, and refine the $st(e)$ weight to the dirtifying store weight $dfs(e)$ defined as follows:

$$dfs(e) \triangleq |\{b \in stores(e) \mid S.dirty(b) \neq D\}|$$

where $stores(e)$ is the set of memory blocks targeted by stores on e and S is the abstract cache state at the source of e. By construction $dfs(e) \leq st(e)$. Given $dfs(e)$, the constraint in Equation 7 can then be improved as follows:

$$\sum_{\text{edge } e} wb(e) \cdot f_e \leq \sum_{\text{edge } e} dfs(e) \cdot f_e\,. \tag{8}$$

To understand the effect of this analysis consider Listing 4. The dirtiness analysis classifies `x` and `sum` as definitely dirty in the two lines leading up to the loop. Variable `i` is classified as dirty following the loop initialization. Both `i` and `sum` are guaranteed to remain in the cache, and so remain definitely dirty during the entire loop execution. Thus the stores to



Persistence constraints:

$$\sum_{\text{edge } e} miss_{sum}(e) \cdot f_e \leq \#cachelines_{sum}$$

$$\sum_{\text{edge } e} miss_{arr}(e) \cdot f_e \leq \#cachelines_{arr}$$

Write-back constraint:

$$\sum_{\text{edge } e} wb(e) \cdot f_e \leq \sum_{\text{edge } e} \left(\sum_{\text{variable } v} dfs_v(e) \right) \cdot f_e$$

Persistent dirtifying store constraint:

$$\sum_{\text{edge } e} dfs_{sum}(e) \cdot f_e \leq \#cachelines_{sum}$$

Figure 3 Simplified state graph for Listing 3 and generated constraints.

`i` and `sum` inside the loop are *non*-dirtifying. The loop itself might evict `x`, so during the loop `x`'s dirtiness is classified as $\top$. Since all accesses to `arr` may evict `x`, each access to `arr` generates a write-back edge. Enforcing the store bound yields a vacuous constraint, since there are $2\mathtt{N} + 2$ stores in the program but only N potential write backs. The dirtifying store analysis, however, recognizes three dirtifying stores (one each to `x`, `sum`, and `i`) and therefore only allows for three write backs as a consequence of the code visible in the listing.

6.2 Stores to Persistent Blocks

The analysis of dirtifying store described above relies on the must-dirty information obtained from the dirtiness analysis. Consequently, the approach suffers from the same shortcomings as the must analysis does in predicting cache hits. Reconsider Listing 3, which motivated persistence analysis. Since the precise blocks accessed by the store to `sum` are unknown at analysis time, no block belonging to `sum` is guaranteed to be cached in any loop iteration or to have been written to. Consequently, the analysis classifies the dirtiness of the array's memory blocks as $\top$ and therefore cannot exclude any of the stores to be dirtifying. As a consequence, the analysis must account for N write backs due to the stores to `sum`.

Persistence analysis exists to remedy this shortcoming of the must analysis; assuming that all blocks accessed in the loop together fit into the cache, persistence analysis proves that `sum` and `arr` are never evicted and therefore cause at most one cache miss for each cache line in `sum` and `arr`. A similar argument holds for the number of dirtifying stores: If `sum` is never evicted, there can be at most one dirtifying store to each cache line of `sum`. For every persistence constraint, we therefore also generate a corresponding constraint bounding

the number of dirtifying stores. Instead of N potential write backs, the path analysis now accounts for at most as many write backs as there are cache lines spanned by `sum`.

Implementing such a persistence-like bound requires dedicated edges for the dirtifying and the non-dirtifying case within the state graph. Figure 3 shows a simplified version of the state graph produced by the microarchitectural analysis for the program in Listing 3. For the sake of readability, we only show the relevant abstract states and relevant, non-zero edge weights. The edge weight $miss_v(e)$ denotes the number of misses to variable v on edge e, $dfs_v(e)$ the number of dirtifying stores to variable v, and $wb(e)$ the number of dirty cache lines written back on edge e. The constant $\#cachelines_v$ refers to the number of cache lines that variable v maps to, which is 1 for basic types such as integers, and may be larger for arrays, depending on the number of array elements and the size of the base type. Next to the state graph, we give the linear constraints obtained from persistence analysis and our store-focused write-back analysis.

Initial State

In the presented analysis, we again assumed an initially clean cache. The analysis can be adjusted to correctly account for initial unknown dirtiness. In that case, one needs to add the number of lines in the cache to the right-hand side of the constraint in Equations 7 and 8.

7 Experimental Evaluation

7.1 Executive Summary

In this section, we evaluate our write-back analysis. First, we determine its overall effectiveness by comparing WCET bounds for a system with a write-back cache (WB) with those obtained for an otherwise equivalent system featuring a write-through cache (WT). In many cases, WCET bounds for the write-back cache are more than 15% lower, making write-back caches not only a feasible but also a desirable choice in hard real-time systems.

Second, we evaluate the individual components of our write-back analysis. We demonstrate that all components presented in this work improve the analysis. We also evaluate a pure eviction-focussed analysis, following Alt et al.'s dirtiness analysis. We observe that it is ineffective in most benchmarks. In the few cases in which it is effective, however, it yields significant improvements over a pure store-focussed analysis.

Third, we evaluate the analysis cost of our analysis for write-back caches. We show that the analysis comes at reasonable cost in terms of both analysis time and memory consumption. Often it even turns out to be cheaper than an analysis for a write-through cache.

Finally, we evaluate how the memory latency affects write-back performance. Since a write backs transfers a whole cache line, while a direct store to memory only transfers a single word, write backs usually take slightly longer. We identify how much longer a write back may take before write-back caches cease to be profitable.

7.2 Experimental Setup

We base our evaluation on the Mälardalen benchmark suite [11]. Due to restrictions of our analyzer, we exclude those benchmarks that do not compile, have external function calls, or use recursion. We additionally generated seven benchmarks with *SCADE* [26], an industry-strength commercial model-based design tool.

We apply our analysis to a processor with a classic 5-stage in-order pipeline that supports a subset of the ARM instruction set architecture. The modeled processor features separate

data and instruction caches. Each of them is a 2-way LRU cache with 16-byte line size and 32 cache sets. As the benchmarks have a small memory footprint, we consequently choose such a small cache size. Finally, the processor contains a single-entry write buffer that buffers stores on their way to main memory. The processor therefore does not wait for stores to complete unless another memory operation is already pending.

The parameters of main memory, such as the latency of an access, are an important factor in our evaluation. While a write-through cache transfers one word (4 bytes) during a store, a write-back cache writes back an entire cache line (in our case 16 bytes). Memory chips support accesses in *burst* mode for this purpose: multiple words are transferred in a single access, one word per cycle. Thus, a line-wide access is slightly more expensive than a word-wide access, but a lot cheaper than accessing all words of a line individually. We choose 10 cycles as the latency of the first word accessed and 1 cycle for each following word in burst mode. These are realistic timings for DRAM chips, e.g. the Micron MT46V16M16 Automotive DDR SDRAM [24]. At the start of an access the previously open row is closed ($t_{RP} = 15ns$), the new row is opened ($t_{RCD} = 15ns$), and the respective column is accessed ($CL = 3$ cycles). For a clock rate of 200 MHz, this amounts to 9 cycles needed to setup the access. Each consecutive word within the burst access is then transferred in one cycle, resulting in the above timing.

All evaluations were performed using our own timing analyzer *llvmta* first used and described in [16]. The analyzer operates on the binary-level program representation generated by the LLVM compiler infrastructure. We employ trace partitioning [23] to perform context-sensitive analyses. We choose contexts to distinguish different call sites of a function and to peel the first iteration of each loop. This is necessary to obtain useful must- and may cache information. Our address analysis determines intervals of potentially accessed addresses for each load and store instruction. We make use of the LLVM-internal scalar evolution analysis to obtain loop bounds.

Since our analyzer cannot handle some of the more involved features used by the LLVM optimizer, all benchmarks are compiled without optimization. This is so far a common choice for safety-critical systems [9]. However, the lack of efficient register allocation frequently causes needless spills and reloads. While a write-back cache can handle both accesses inside the cache, a write-through cache accesses memory on each register spill. Thus, it is not obvious whether our evaluation results also apply to highly optimized machine code.

7.3 Write-through versus Write-back Caches

Table 1 shows the WCET bounds obtained for all benchmarks for a system with unblocked stores (i.e. with a write buffer, which is the standard configuration) and for a system with blocked stores. These results are summarized in Figure 4 via two histograms: both histograms depict the number of benchmarks that fall into a particular bin of ratios between the WCET under write through (WT) and write back (WB). The histogram on the left corresponds to the case with unblocked stores, while the histogram on the right corresponds to the case with blocked stores. Benchmarks on the left of the line at 1.0 have smaller WCET bounds under WB than under WT, and vice versa benchmarks on the right of the line have larger WCET bounds under WB than under WT. A logarithmic scale is applied on the x-axis as the values are ratios, and so a ratio of 2.0 is at the same distance to 1.0 as a ratio of 0.5 is in the opposite direction.

The first observation is that WB is preferable to WT on most benchmarks (32/36 and 34/36, respectively) in both cases. The second observation is that the ratio WB/WT is usually smaller in case of blocked stores than it is in case of unblocked stores. A larger

Table 1 WCET bounds (in 1000 cycles) for write-through and write-back caches with blocked and unblocked stores and ratios between the WCET bounds in the two cases. The memory share is an estimate on the time the write-back system spends accessing memory. WB free is the write-back WCET assuming write backs take no time.

		Blocked stores			Unblocked stores				Unblocked stores	
	Benchmark	WT	WB	$\frac{WB}{WT}$	WT	WB	$\frac{WB}{WT}$	D-cache misses	Mem. share	$\frac{WB\ \text{free}}{WB}$
Mälardalen	adpcm	1453	858	59%	1018	857	84%	1186	4%	99%
	bs	2	2	67%	2	2	76%	13	31%	100%
	bsort100	1911	1102	58%	1264	1102	87%	58	0%	100%
	cnt	52	31	58%	40	30	76%	64	6%	98%
	compress	360	341	95%	316	330	104%	6296	50%	83%
	crc	767	464	61%	532	449	84%	1875	11%	95%
	expint	270	170	63%	174	170	98%	9	0%	100%
	fdct	10	11	114%	9	11	127%	325	82%	81%
	fft1	78	37	46%	59	37	62%	39	3%	100%
	fibcall	5	3	54%	4	3	76%	3	3%	100%
	fir	3940	2738	69%	3437	2652	77%	79807	78%	89%
	insertsort	8	5	57%	6	5	86%	4	2%	100%
	janne-complex	19	12	63%	13	12	91%	4	1%	100%
	jfdctint	17	15	90%	13	14	115%	287	54%	85%
	lcdnum	4	3	79%	3	3	88%	23	23%	96%
	lms	4818	3403	71%	3716	3376	91%	23990	18%	97%
	ludcmp	110	96	87%	99	95	96%	3521	97%	88%
	matmult	1543	1615	105%	1348	1547	115%	57213	96%	85%
	minver	39	28	73%	32	28	87%	359	34%	89%
	ndes	529	325	61%	414	319	77%	2796	23%	95%
	ns	105	84	80%	86	84	98%	791	25%	100%
	nsichneu	104	100	96%	99	97	98%	739	20%	92%
	prime	82	56	68%	56	56	100%	5	0%	100%
	qsort-exam	100	67	67%	67	67	100%	371	14%	96%
	qurt	29	16	54%	22	16	70%	41	7%	100%
	select	53	44	83%	47	43	93%	613	37%	91%
	sqrt	9	7	71%	8	7	83%	6	3%	100%
	statemate	21	17	82%	20	17	85%	26	4%	100%
	ud	63	53	85%	53	53	100%	1468	73%	92%
	geo. mean			71%			89%			95%
SCADE	cruise-control	162	122	75%	158	122	77%	94	2%	100%
	digital-stopwatch	3031	1793	59%	2321	1772	76%	5354	8%	97%
	es-lift	160	133	83%	155	132	85%	1773	35%	98%
	flight-control	3608	2271	63%	2951	2257	76%	9216	11%	96%
	pilot	226	192	85%	205	189	92%	3488	48%	89%
	roboDog	440	407	93%	424	402	95%	6587	43%	90%
	trolleybus	1150	1022	89%	1127	1005	89%	18736	48%	89%
	geo. mean			77%			84%			94%
	geo. mean (Mäl. + SCADE)			72%			88%			95%

number of store accesses to main memory makes write buffers more profitable. Thus, systems with write-back caches profit less from the additional write buffer because they write less often to main memory than write-through caches. As write buffers are common in modern processors, this demonstrates that it is essential to account for their presence for a fair comparison between the two write policies.

7.4 Analysis Precision: Decomposition into Contributions

We now evaluate the effects of the individual analysis components. Starting at the naive analysis that assumes a write back on every cache miss, Figure 5 shows the incremental improvement up to the final analysis.

As predicted in Section 6, the store bound has next to no effect on the WCET bound. Recognizing dirtifying stores is required to achieve appreciable bound reductions, as can be

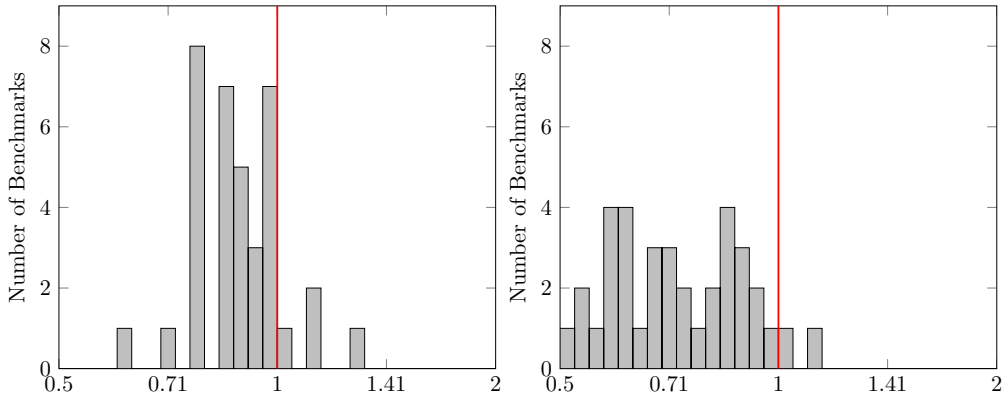
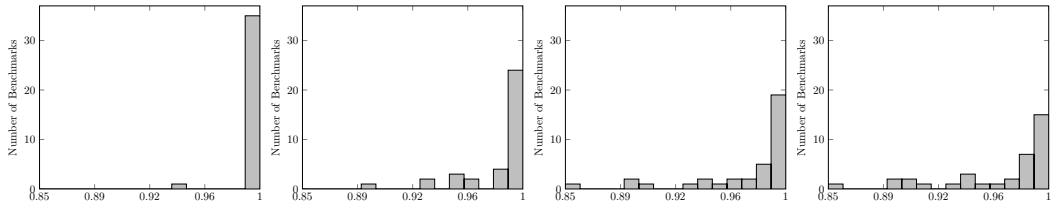


Figure 4 Histogram of ratios of WCET bounds compared to the write-through system. Left: Unblocked stores via a single-entry write buffer. Right: Blocked stores and thus no write buffer.



(a) Store Bound **(b)** + Dirtifying stores **(c)** + Store-persistence **(d)** + Eviction analysis

Figure 5 Histogram of ratios of WCET bounds compared to the naive analysis that assumes a write back on every cache miss.

observed in part (b) of Figure 5. Most benchmarks are still unaffected by the analysis, though. We conjecture that non-dirtifying stores often occur in loops, where multiple iterations target the same cache line. The regular dirtifying store analysis fails to recognize this unless the accessed block is known precisely. We therefore developed store-persistence constraints, which recognize stores to persistent blocks (which may be part of an array) and bound the number of dirtifying stores appropriately. Figure 5(c) indicates that the conjecture about non-dirtifying stores is correct; adding the persistence constraints improves the results significantly and achieves the greatest speedup, e.g. reducing *ludcmp*'s bound by 15%.

With Figure 5(c) we have reached the the final result for a pure store-focussed analysis. We propose to combine the store-focussed analysis with an eviction-focussed analysis. Figure 6 shows the reason: the two approaches are orthogonal, i.e. programs are usually either suited to one or the other. This effect can also be observed in Figure 5(d): eviction-focussed analysis yields crucial bound improvements on a few benchmarks but has no effect on the majority of benchmarks. Since one needs a microarchitectural dirtiness analysis to implement the dirtifying store bound anyway, performing an eviction-based analysis has only a negligible effect on the analysis runtime.

Surprisingly, about half of the benchmarks in Figure 5(d) are not affected by either analysis. One reason is that many benchmarks spend little time accessing memory; write-back analysis therefore cannot possibly have a large effect on the overall WCET. We provide an estimate of the amount of time a benchmarks spends accessing memory in the *Memory share* column of Table 1. This amount of time is estimated by multiplying the number of data cache misses with the latency of *two* memory accesses (i.e. a cache miss and a write back). Dividing this time by the WCET yields the memory share.

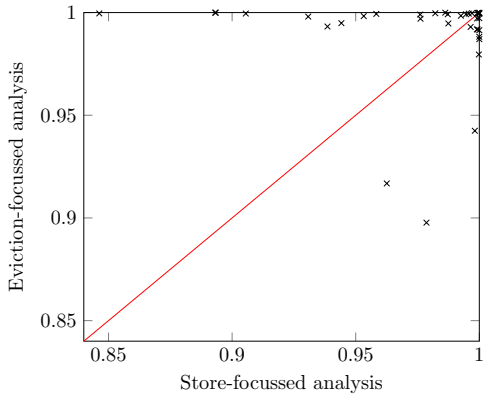


Figure 6 Comparison between the store-focussed analysis and the eviction-focussed analysis. The WCETs are normalized to the naive write-back analysis.

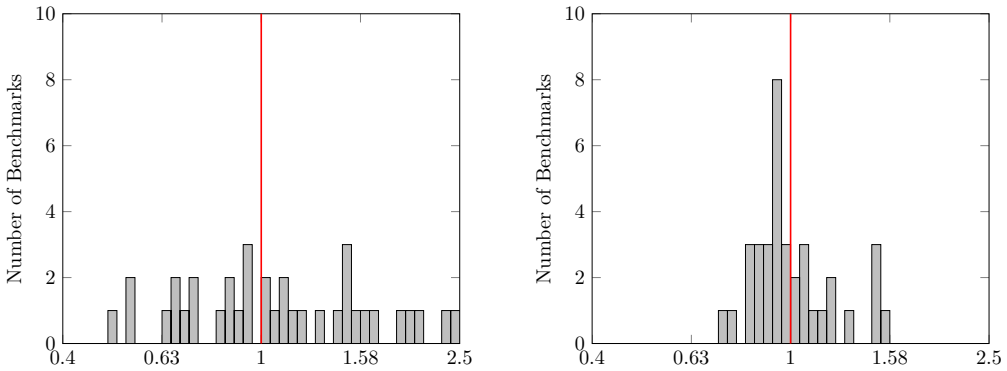


Figure 7 Histogram of ratio of runtimes (left) and memory consumptions (right) of the write-back analysis relative to the write-through analysis.

Consider for example *bsort100*. As one can see in Table 1, it spends less than 0.5% of its runtime accessing memory, *even if all cache misses are write backs*. This means that even a perfect write-back analysis would have no noticeable impact on *bsort100*'s WCET bound. In total 10 benchmarks spend $\leq 3\%$ of their runtime accessing memory.

This property of benchmarks can also be observed by considering WCET bounds obtained under the assumption that write backs are free (called "WB free" in the table). This value represents a fundamental lower bound on what an analysis can achieve by proving write backs impossible, if the number of cache misses accounted for remains constant. Table 1 shows that 13 benchmarks are already within 3% of this boundary. In total, our analysis achieves 95% of this boundary on average. Significant further improvements can therefore only stem from improving the underlying cache analysis, thereby reducing the number of cache misses accounted for.

7.5 Analysis Performance

Although analysis performance has not been a major goal in this work, it is important that the analysis is not prohibitively expensive. Figure 7 shows a histogram of the ratio of analysis runtime and memory consumption for the analysis of the write-back cache compared to the analysis of the write-through cache for each of the 36 benchmarks. The results vary widely: some programs take half the analysis time, some double analysis time. This may be

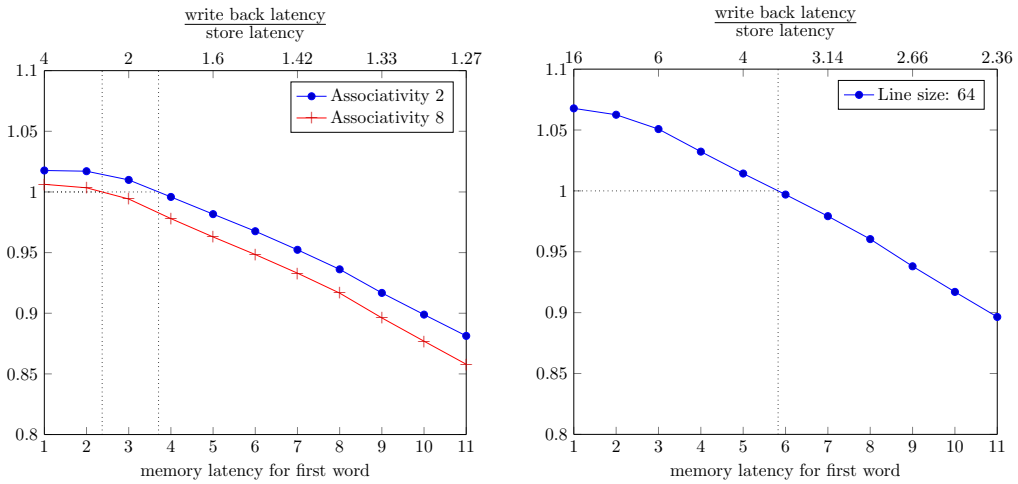


Figure 8 Geometric mean of WB/WT ratios for different latencies to access the first word of a cacheline. Each consecutive word always requires one additional cycle.

explained by two opposing effects: (i) On the one hand, it seems natural that the analysis of write-back caches is more expensive: Besides the increased size of abstract cache states due to dirtiness information, the microarchitectural analysis performs additional splits due to uncertainty about whether a write back happens or not. (ii) On the other hand, write-back caches reduce execution times which manifest in smaller state graphs with fewer states to explore for microarchitectural analysis whenever the eviction-focussed analysis is successful in excluding writebacks. All in all, we conclude that the write-back analysis comes at a reasonable cost.

7.6 Influence of Memory Latency on Write-back Execution Times

Recall our cache and memory parameters: cache line size of 16 bytes, i.e., 4 words of 4 bytes, a 10-cycle latency for the first word accessed, and a 1-cycle burst latency for consecutive words within a burst access. A write back then costs $\frac{13}{10} = 1.3$ times as much as a write-through store. On average, programs should therefore perform 0.3 stores on each dirty cache line before its eviction to compensate for the increased cost.

Clearly the profitability of the write-back cache crucially depends on these memory timing parameters. We therefore consider different latency scenarios to see how the picture might change for a main memory with different latency characteristics. To this end, we vary the initial latency, i.e. the latency to access the first word within a burst access from 1 to 11. The burst latency, i.e. the latency to access additional consecutive words, is kept constant at one additional cycle. An initial latency of 1 models the case of a pure random-access memory, where the latency of each word is the same. The higher the initial latency, the smaller the gap between the latency of a write back and the latency of a write-through store. We used an initial latency of 10 in all other experiments in this paper because it is realistic for modern embedded memory as discussed in Section 7.2.

Figure 8 shows the geometric mean of the WB/WT ratios for different initial latencies. We consider three cache configurations: (1) the standard two-way set-associative 1 KiB cache configuration described in Section 7.2 (“Associativity 2” in the figure), (2) an eight-way set-associative 4 KiB cache (“Associativity 8”), and (3) a two-way set-associative 4 KiB cache with a line size of 64 bytes (“Line size: 64”).

It is informative to check where the curves reach a ratio of 1, the break-even point between write back and write through. In the standard configuration write back is profitable for initial latencies above around 3.7. A slightly lower value is obtained for the larger cache with higher associativity (beneficial above 2.4). In case of a line size of 64 bytes the break-even point is around 5.8. Modern memories should lie well to the right of each of these points, rendering write-back caches profitable.

8 Related Work

In this section, we discuss related work on WCET and response-time analysis for systems with instruction and data caches; in particular work targeting write-back caches. All approaches described below assume caches with LRU replacement.

Analysis for write-back caches inherits all the challenges imposed by data-cache analysis compared with instruction-cache analysis. The main additional difficulty of data-cache analysis is obtaining information about the accessed addresses. While addresses of instructions inside a binary are easy to obtain, the addresses of data accesses can often not be pinned down to a single value. Examples include array accesses within a loop or input-dependent accesses resulting in a range of possibly accessed addresses.

Alt et al. [1] first introduced cache analysis based on abstract interpretation. They proposed must and may analysis to classify accesses as always hit/miss. They considered data cache analysis only for scalar accesses whose addresses could be precisely determined. For write-back caches, they extend the may analysis to track whether a block may be dirty. Based on this may-dirtiness information they locally exclude a write back if only clean blocks may get evicted. No experimental results concerning the proposed write-back analysis are given in Alt et al. [1] nor in the subsequent journal paper by Ferdinand and Wilhelm [8].

In [28], White et al. extend their prior work on static cache simulation to data caches. In cache simulation, they use abstract cache states, comparable to may-cache states, and (post-)dominator information to derive always hit/miss and first hit/miss classifications.

Ferdinand and Wilhelm [8] present a persistence analysis to improve the analysis of data caches. A memory block is deemed persistent if it cannot be evicted once it is loaded into the cache. The persistence analysis can handle ranges of possibly accessed addresses.

Huynh et al. [15] introduce the notion of temporal scope to improve data cache persistence analysis. The temporal scope of a memory block denotes the loop iterations in which the block might be accessed. Memory blocks with non-overlapping temporal scopes can thus not conflict within the persistence analysis. Their persistence analysis could be used to further increase the precision of our approach. It is, however, not obvious how widely applicable it is. In addition, Huynh et al. fix a problem in the original persistence analysis by Ferdinand and Wilhelm. In the same year, Cullmann [5] also provided a corrected persistence analysis. Later, Cullmann [6] describes the conflict-set analysis (termed "conflict counting" analysis there), which we use as a persistence analysis in this paper.

Sondag and Rajan [27] propose an analysis of multi-level caches and contribute the notion of *live caches* to reason about cache blocks that must be in one of the cache levels but are not guaranteed to be in any particular one. Analyzing write backs from the L1 cache to the L2 cache is necessary to correctly model the behavior of the L2 cache. Their eviction-focussed analysis uses a must/may-dirty analysis, similar to our dirtiness analysis. However, it remains unclear how their analysis works exactly; for instance, their update function does not distinguish between loads and stores. They do not use the must-dirty information to derive a set of dirtifying stores which we consider essential to precise write-back analysis.

They evaluate the impact of the live caches on the provable multi-level cache performance, but they give no results concerning their write-back analysis.

Lesage et al. [20] also consider the analysis of multi-level data caches. However, they limit themselves to the analysis of a write-through and write-no-allocate policy to avoid the complications induced by write-back caches.

Hahn and Grund [12] present relational cache analysis to overcome the necessity of exact absolute address information. Instead of using absolute addresses, they use relations between referenced addresses, such as *same block* or *different set*, to analyze a task's cache behavior. With such a relational analysis consecutive accesses to the same but unknown address can be classified as hits. Such an analysis could be used to derive sharper bounds on the number of dirtifying stores.

A detailed survey on cache analysis is given by Lv et al. [22].

The work discussed above targets the timing and cache analysis of individual tasks. Based on per-task characteristics, schedulability analyses determines whether a set of tasks can be scheduled together on a hardware platform. Some work on schedulability analysis takes platform-induced overheads into account, such as the effects of preemptive scheduling on the cache behavior [4, 19, 2].

Davis et al. [7] consider the effect of write-back caches on a preemptively scheduled fixed-priority system. Their response-time analysis uses a per-task characterization of dirty cache blocks (DCBs) and final dirty cache blocks (FDCBs). The evaluation is based on simulated execution traces of the tasks, because no WCET analysis accounting for write-back caches was available at the time of publication. Basing the analysis on simulation results, however, avoids any uncertainty that arises within static analysis. The analysis techniques presented in this paper could be used to provide the needed per-task WCET characterization by static analysis. With some further effort, the dirtiness analysis could also be used to extract DCBs and FDCBs. Davis et al. and other approaches to response-time analysis rely on timing compositionality [14]. As an example, Davis et al. assume that the cost of each additional write back is bounded by the memory latency. Due to amplifying timing anomalies, this assumption does not hold for most hardware platforms rendering naive compositional analysis unsound. However, the approach introduced in [13] to enable compositional timing analysis even in the presence of anomalies, is applicable also to the WCET analysis presented in this paper.

9 Conclusions and Perspectives

We have discussed and fleshed out the existing eviction-focussed approach to write-back analysis. We have also introduced a new store-focussed approach. As both approaches are beneficial on almost disjoint sets of tasks it is beneficial to combine both in a single analysis, as we have done.

To the best of our knowledge, we have conducted the first experimental evaluation of any WCET analysis for write-back caches. It shows that write-back caches are preferable to write-through caches from a WCET perspective for most benchmarks. The evaluation also demonstrates that write buffers are much more valuable in conjunction with write-through caches than with write-back caches. As write buffers are common in modern processors, it is essential to account for their presence for a fair comparison between the two write policies. It is future work to evaluate whether larger write buffers further shift results in favor of write-through caches.

References

1 Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In Radhia Cousot and David A. Schmidt, editors, *Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, September 24-26, 1996, Proceedings*, volume 1145 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 1996. doi:10.1007/3-540-61739-6_33.

2 Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium, RTSS 2011, Vienna, Austria, November 29 – December 2, 2011*, pages 261–271, 2011. doi:10.1109/RTSS.2011.31.

3 ARM Limited. *ARM946E-S Technical Reference Manual. Available at* http://infocenter.arm.com/help/topic/com.arm.doc.ddi0201d/DDI0201D_arm946es_r1p1_trm.pdf.

4 José V. Busquets-Mataix, Juan José Serrano, Rafael Ors, Pedro J. Gil, and Andy J. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceedings of the IEEE Real-Time Embedded Technology and Applications (RTAS)*, pages 204–212, June 1996.

5 Christoph Cullmann. Cache persistence analysis: a novel approach – theory and practice. In *Proceedings of the ACM SIGPLAN/SIGBED 2011 conference on Languages, compilers, and tools for embedded systems, LCTES 2011, Chicago, IL, USA, April 11-14, 2011*, pages 121–130, 2011. doi:10.1145/1967677.1967695.

6 Christoph Cullmann. Cache persistence analysis: Theory and practice. *ACM Trans. Embed. Comput. Syst.*, 12(1s):40:1–40:25, March 2013. doi:10.1145/2435227.2435236.

7 Robert I. Davis, Sebastian Altmeyer, and Jan Reineke. Analysis of write-back caches under fixed-priority preemptive and non-preemptive scheduling. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS 2016, Brest, France, October 19-21, 2016*, pages 309–318, 2016. doi:10.1145/2997465.2997476.

8 Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2):131–181, 1999. doi:10.1023/A:1008186323068.

9 Ricardo Bedin França, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Towards formally verified optimizing compilation in flight control software. In *PPES 2011: Predictability and Performance in Embedded Systems*, volume 18, pages 59–68. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2011.

10 Freescale Semiconductor, Inc. *MPC603e RISC Microprocessor User's Manual. Available at* http://www.nxp.com/files/32bit/doc/ref_manual/MPC603EUM.pdf.

11 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. In Björn Lisper, editor, *WCET2010*, pages 137–147, Brussels, Belgium, July 2010. OCG.

12 Sebastian Hahn and Daniel Grund. Relational cache analysis for static timing analysis. In *24th Euromicro Conference on Real-Time Systems, ECRTS 2012, Pisa, Italy, July 11-13, 2012*, pages 102–111, 2012. doi:10.1109/ECRTS.2012.14.

13 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, RTNS 2016, Brest, France, October 19-21, 2016*, pages 299–308, 2016. doi:10.1145/2997465.2997471.

14 Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards compositionality in execution time analysis: definition and challenges. *SIGBED Review*, 12(1):28–36, 2015. doi:10.1145/2752801.2752805.

15 Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. Scope-aware data cache analysis for WCET estimation. In *17th IEEE Real-Time and Embedded Technology and Applications*

Symposium, RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011, pages 203–212, 2011. doi:10.1109/RTAS.2011.27.

16 Michael Jacobs, Sebastian Hahn, and Sebastian Hack. WCET analysis for multi-core processors with shared buses and event-driven bus arbitration. In *Proceedings of the 23rd International Conference on Real Time Networks and Systems, RTNS 2015, Lille, France, November 4-6, 2015*, pages 193–202, 2015. doi:10.1145/2834848.2834872.

17 Norman P. Jouppi. Cache write policies and performance. In Alan Jay Smith, editor, *Proceedings of the 20th Annual International Symposium on Computer Architecture. San Diego, CA, May 1993*, pages 191–201. ACM, 1993. doi:10.1145/165123.165154.

18 Daniel Kröning and Silvia M. Müller. The impact of write-back on the cache performance. In *Proceedings of the IASTED International Conference on Applied Informatics, Innsbruck*, pages 213–217. ACTA Press, 2000.

19 Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

20 Benjamin Lesage, Damien Hardy, and Isabelle Puaut. WCET analysis of multi-level set-associative data caches. In Niklas Holsti, editor, *9th Int'l Workshop on Worst-Case Execution Time Analysis, WCET 2009, Dublin, Ireland, July 1-3, 2009*, volume 10 of *OASICS*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, Germany, 2009. URL: http://www.dagstuhl.de/dagpub/978-3-939897-14-9, doi:10.4230/OASIcs.WCET.2009.2283.

21 Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium, Phoenix, AZ, USA, December 1-3, 1999*, pages 12–21, 1999. doi:10.1109/REAL.1999.818824.

22 Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. A survey on static cache analysis for real-time systems. *LITES*, 3(1):05:1–05:48, 2016. doi:10.4230/LITES-v003-i001-a005.

23 Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In Shmuel Sagiv, editor, *14th European Symposium on Programming (ESOP) 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer, 2005. doi:10.1007/978-3-540-31987-0_2.

24 Micron Technology, Inc. *Automotive DDR SDRAM MT46V32M8, MT46V16M16. Available at* https://www.micron.com/~/media/documents/products/data-sheet/dram/mobile-dram/low-power-dram/lpddr/256mb_x8x16_at_ddr_t66a.pdf.

25 Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *6th Int'l Workshop on Worst-Case Execution Time (WCET) Analysis, July 4, 2006, Dresden, Germany*, 2006. doi:10.4230/OASIcs.WCET.2006.671.

26 SCADE suite. URL: http://www.esterel-technologies.com/products/scade-suite/.

27 Tyler Sondag and Hridesh Rajan. A more precise abstract domain for multi-level caches for tighter WCET analysis. In *Proceedings of the 31st IEEE Real-Time Systems Symposium, RTSS 2010, San Diego, California, USA, November 30 – December 3, 2010*, pages 395–404, 2010. doi:10.1109/RTSS.2010.8.

28 Randall T. White, Christopher A. Healy, David B. Whalley, Frank Mueller, and Marion G. Harmon. Timing analysis for data caches and set-associative caches. In *3rd IEEE Real-Time Technology and Applications Symposium, RTAS'97, Montreal, Canada, June 9-11, 1997*, pages 192–202, 1997. doi:10.1109/RTTAS.1997.601358.