35th Euromicro Conference on Real-Time Systems

ECRTS 2023, July 11–14, 2023, Vienna, Austria

Edited by

Alessandro V. Papadopoulos

Editors

Alessandro V. Papadopoulos
Mälardalen University, Västerås, Sweden
alessandro.papadopoulos@mdu.se

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

ISBN 978-3-95977-280-8

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-280-8.

Publication date
July, 2023

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

License
This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0): https://creativecommons.org/licenses/by/4.0/legalcode.

In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:
- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.ECRTS.2023.0

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.

Editorial Board

- Luca Aceto (*Chair*, Reykjavik University, IS and Gran Sasso Science Institute, IT)
- Christel Baier (TU Dresden, DE)
- Mikolaj Bojanczyk (University of Warsaw, PL)
- Roberto Di Cosmo (Inria and Université de Paris, FR)
- Faith Ellen (University of Toronto, CA)
- Javier Esparza (TU München, DE)
- Daniel Král' (Masaryk University - Brno, CZ)
- Meena Mahajan (Institute of Mathematical Sciences, Chennai, IN)
- Anca Muscholl (University of Bordeaux, FR)
- Chih-Hao Luke Ong (University of Oxford, GB and Nanyang Technological University, SG)
- Phillip Rogaway (University of California, Davis, US)
- Eva Rotenberg (Technical University of Denmark, Lyngby, DK)
- Raimund Seidel (Universität des Saarlandes, Saarbrücken, DE and Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern, DE)

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

Preface
Peter Puschner and Alessandro V. Papadopoulos 0:vii–0:viii

Organizers
.... 0:ix–0:xi

Authors
.... 0:xiii–0:xvi

Papers

Scheduling and Compiling Rate-Synchronous Programs with End-To-End Latency Constraints
Timothy Bourke, Vincent Bregeon, and Marc Pouzet 1:1–1:22

Towards Efficient Explainability of Schedulability Properties in Real-Time Systems
Sanjoy Baruah and Pontus Ekberg 2:1–2:20

The Safe and Effective Use of Low-Assurance Predictions in Safety-Critical Systems
Kunal Agrawal, Sanjoy Baruah, Michael A. Bender, and Alberto Marchetti-Spaccamela 3:1–3:19

Memory Latency Distribution-Driven Regulation for Temporal Isolation in MPSoCs
Ahsan Saeed, Denis Hoornaert, Dakshina Dasari, Dirk Ziegenbein, Daniel Mueller-Gritschneder, Ulf Schlichtmann, Andreas Gerstlauer, and Renato Mancuso 4:1–4:23

Quasi Isolation QoS Setups to Control MPSoC Contention in Integrated Software Architectures
Sergio Garcia-Esteban, Alejandro Serrano-Cases, Jaume Abella, Enrico Mezzetti, and Francisco J. Cazorla 5:1–5:25

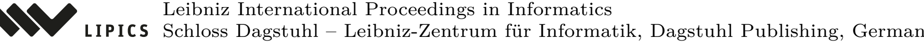
FusionClock: Energy-Optimal Clock-Tree Reconfigurations for Energy-Constrained Real-Time Systems
Eva Dengler, Phillip Raffeck, Simon Schuster, and Peter Wägemann 6:1–6:23

CAWET: Context-Aware Worst-Case Execution Time Estimation Using Transformers
Abderaouf N Amalou, Elisa Fromont, and Isabelle Puaut 7:1–7:20

Precise Scheduling of DAG Tasks with Dynamic Power Management
Ashikahmed Bhuiyan, Mohammad Pivezhandi, Zhishan Guo, Jing Li, Venkata Prashant Modekurthy, and Abusayeed Saifullah 8:1–8:24

Bounding the Data-Delivery Latency of DDS Messages in Real-Time Applications
Gerlando Sciangula, Daniel Casini, Alessandro Biondi, Claudio Scordino, and Marco Di Natale 9:1–9:26

On the Equivalence of Maximum Reaction Time and Maximum Data Age for

35th Euromicro Conference on Real-Time Systems (ECRTS 2023).
Editor: Alessandro V. Papadopoulos

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

Cause-Effect Chains
Mario Günzel, Harun Teper, Kuan-Hsun Chen, Georg von der Brüggen, and Jian-Jia Chen 10:1–10:22

A New Perspective on Criticality: Efficient State Abstraction and Run-Time Monitoring of Mixed-Criticality Real-Time Control Systems
Tim Rheinfels, Maximilian Gaukler, and Peter Ulbrich 11:1–11:26

Isospeed: Improving (min,+) Convolution by Exploiting (min,+)/(max,+) Isomorphism
Raffaele Zippo, Paul Nikolaus, and Giovanni Stea 12:1–12:24

Low-Overhead Online Assessment of Timely Progress as a System Commodity
Weifan Chen, Ivan Izhbirdeev, Denis Hoornaert, Shahin Roozkhosh, Patrick Carpanedo, Sanskriti Sharma, and Renato Mancuso 13:1–13:26

Consensual Resilient Control: Stateless Recovery of Stateful Controllers
Aleksandar Matovic, Rafal Graczyk, Federico Lucchetti, and Marcus Völp 14:1–14:27

Impact of Transient Faults on Timing Behavior and Mitigation with Near-Zero WCET Overhead
Pegdwende Romaric Nikiema, Angeliki Kritikakou, Marcello Traiola, and Olivier Sentieys 15:1–15:22

Optimal Multiprocessor Locking Protocols Under FIFO Scheduling
Shareef Ahmed and James H. Anderson 16:1–16:21

A Tight Holistic Memory Latency Bound Through Coordinated Management of Memory Resources
Shorouk Abdelhalim, Danesh Germchi, Mohamed Hossam, Rodolfo Pellizzoni, and Mohamed Hassan 17:1–17:25

Replication-Based Scheduling of Parallel Real-Time Tasks
Federico Aromolo, Geoffrey Nelissen, and Alessandro Biondi 18:1–18:23

Invited Paper

From FMTV to WATERS: Lessons Learned from the First Verification Challenge at ECRTS
Sebastian Altmeyer, Étienne André, Silvano Dal Zilio, Loïc Fejoz, Michael González Harbour, Susanne Graf, J. Javier Gutiérrez, Rafik Henia, Didier Le Botlan, Giuseppe Lipari, Julio Medina, Nicolas Navet, Sophie Quinton, Juan M. Rivas, and Youcheng Sun 19:1–19:18

Preface

Message from the Chairs

We welcome you to ECRTS 2023, in Vienna, Austria. Alongside RTSS and RTAS, ECRTS ranks as one of the top three international conferences on real-time systems and is the premier European conference series on this topic. Given the lessons learned during the pandemic, we continue even for this year ECRTS to include the possibility to participate online, as this is an opportunity to make the community more diverse. We are delighted to have you join the second hybrid ECRTS, for an exciting program consisting of both scientific talks and opportunities for socializing and collaborative work.

ECRTS has been at the forefront of recent innovations in the real-time systems community such as artifact evaluation, open-access proceedings, and a flexible page limit. This year we have consolidated the experience and repeated a double-blind submission process with flexible page limit, that does not constrain the authors and allows them to put the effort into optimizing the content of their submission, rather than space utilization.

ECRTS 2023 received a total of 65 submissions from Asia, Europe, and North America. Each submission was reviewed by at least three expert members of the program committee and discussed both on the discussion board of the submission website and at the program committee meeting that took place on April 19 and 20, 2023. The program committee accepted 18 papers for publication and presentation, which translates to an acceptance rate of 27.7%. An additional paper has been invited to discuss the lessons learned from the first verification challenge at ECRTS.

In addition, on the scientific side, the ECRTS industrial challenge will be presented and discussed at the conference, following a long-lasting tradition of industrial challenges coming from the WATERS workshop.

A major conference like ECRTS 2023 is the result of the hard work of many people involved in the conference organization. First of all, we would like to thank the program committee members, for their hard work despite all the burdens of yet another challenging year. Similarly, thanks to all external and secondary reviewers, who provided many valuable perspectives and important feedback. We are especially grateful to those PC members and additional reviewers who went “above and beyond” serving as shepherds. We would also like to extend our thanks to the Artifact Evaluation Chairs, Julien Forget and Matthias Becker, and their board of Artifact Evaluators for running the AE process, and to the Real-Time Pitches Chair, Alexander Züpke, for bringing fresh new ideas and discussions to the conference. Without the dedicated support of all these colleagues, we would not have been able to put together such an excellent program. Many thanks to all of you!

We would like to thank the ECRTS Executive Committee, Yasmina Abdeddaïm, Sebastian Altmeyer, Steve Goddard, and Marcus Völp, for their outstanding service to the community. Last but not least, we thank all authors for submitting to ECRTS 2023. Whether or not the submission was ultimately accepted for publication, we deeply appreciate your fine work and the tremendous effort and care that has gone into it; this conference would not be possible without you.

35th Euromicro Conference on Real-Time Systems (ECRTS 2023).
Editor: Alessandro V. Papadopoulos
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Finally, we would like to thank all authors who submitted their work to ECRTS 2023, whether it was accepted or not. Without them, this conference would not have been possible.

We are looking forward to an inspiring scientific program in Vienna and online. Please join us in enjoying both the technical content and everything around it, especially with the return to in-person events.

Peter Puschner	Alessandro V. Papadopoulos
General Chair ECRTS 2023	Program Chair ECRTS 2023

Organizers

Euromicro Real-Time Technical Committee

General Chair

Peter Puschner, TU Wien, Austria

Program Chair

Alessandro V. Papadopoulos, Mälardalen University, Sweden

Publicity Chair

Steve Goddard, University of Iowa, USA

Social Media Chair

Kuan H. Chen, University of Twente, Netherlands

Artifact Evaluation Chairs

Matthias Becker, Royal Institute of Technology (KTH), Sweden
Julien Forget, Université de Lille, France

Industrial challenge chairs

Andrea Bastoni, Technical University of Munich, Germany
Paolo Burgio, Universitá di Modena e Reggio Emilia, Italy

Real-time pitches chair

Alexander Züpke, Technical University of Munich, Germany

Program Committee

Yasmina Abdeddaïm, LIGM, Univ Gustave Eiffel, CNRS, France
Luca Abeni, Scuola Superiore Sant'Anna, Pisa, Italy
Luis Almeida, University of Porto, Portugal
Sebastian Altmeyer, Augsburg University, Germany
Jim Anderson, University of North Carolina at Chapel Hill, NC, USA
Matthias Becker, Royal Institute of Technology (KTH), Sweden
Marko Bertogna, Universitá di Modena e Reggio Emilia, Italy
Enrico Bini, University of Turin, Italy
Gedare Bloom, University of Colorado Colorado Springs, CO, USA
Anne Bouillard, Huawei, France
Timothy Bourke, INRIA, France
Daniel Bristot de Oliveira, Red Hat, Italy
Thomas Carle, University Toulouse III, France
Laura Carnevali, University of Florence, Italy
Daniel Casini, Scuola Superiore Sant'Anna, Pisa, Italy

35th Euromicro Conference on Real-Time Systems (ECRTS 2023).
Editor: Alessandro V. Papadopoulos
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Silviu Craciunas, TTTech, Austria
Pontus Ekberg, Uppsala University, Sweden
Gerhard Fohler, TU Kaiserslautern, Germany
Julien Forget, Université de Lille, France
Marisol Garcia Valls, Universitat Politecnica de Valencia, Spain
Steve Goddard, University of Iowa, IA, USA
Giovani Gracioli, Federal University of Santa Catarina, Brazil
Nan Guan, University of Hong Kong, Hong Kong
Arpan Gujarati, University of British Columbia, Canada
Tingting Hu, University of Luxembourg, Luxembourg
Mehdi Kargahi, University of Teheran, Iran
Angeliki Kritikakou, University of Rennes 1, France
Lucia Lo Bello, University of Catania, Italy
Yehan Ma, Shanghai Jiao Tong University, China
Renato Mancuso, Boston University, MA, USA
Filip Markovic, MPI-SWS, Germany
Saad Mubeen, Mälardalen University, Sweden
Frank Mueller, North Carolina State University, NC, USA
Geoffrey Nelissen, Eindhoven University of Technology, Netherlands
Catherine E. Nemitz, Davidson College, NC, USA
Risat Mahmud Pathan, University of Gothenburg, Sweden
Rodolfo Pellizzoni, University of Waterloo, Canada
Edoardo Quinones, Barcelona Supercomputing Center, Spain
Jan Reineke, Saarland University, Germany
Christine Rochange, University of Toulouse, France
Jean-Luc Scharbarg, University of Toulouse, France
Marcus Völp, University of Luxembourg, Luxembourg
Georg von der Brüggen, TU Dortmund, Germany
Bryan Ward, Vanderbilt University, TN, USA
Heechul Yun, University of Kansas, KS, USA
Dirk Ziegenbein, Robert Bosch GmbH, Germany
Marco Zimmerling, University of Freiburg, Germany

External reviewers

Ibrahim Alkoudsi, TU Kaiserslautern, Germany
Federico Aromolo, Scuola Superiore Sant'Anna, Pisa, Italy
Vijay Banerjee, University of Colorado Colorado Springs, CO, USA
Patrick Carpanedo, Boston University, MA, USA
Ö"zgür Ceyhan, University of Luxembourg, Luxembourg
Weifan Chen, Boston University, MA, USA
Giorgiomaria Cicero, Scuola Superiore Sant'Anna, Pisa, Italy
Francesco Ciraolo, Boston University, MA, USA
Bassel El Mabsout, Boston University, MA, USA
Gautam Gala, TU Kaiserslautern, Germany
Golsana Ghaemi, Boston University, MA, USA
Sandro Grebant, Université de Lille, France
Denis Hoornaert, Technical University of Munich, Germany
Sena Hounsinou, Metropolitan State University, MN, USA

Omolade Ikumapayi, University of Colorado Colorado Springs, CO, USA
Ivan Izhbirdeev, Boston University, MA, USA
Martina Maggio, Saarland University, Germany
Luiz Maia Neto, TU Kaiserslautern, Germany
Jean Malm, Mälardalen University, Sweden
Brayden McDonald, North Carolina State University, USA
Swastik Mittal, North Carolina State University, USA
Amin Naghavi, University of Luxembourg, Luxembourg
Mattia Nicolella, Boston University, MA, USA
Habeeb Olufowobi, University of Texas at Arlington, TX, USA
Miloš Ojdanić, University of Luxembourg, Luxembourg
Francesco Paladino, Scuola Superiore Sant'Anna, Pisa, Italy
Paolo Pazzaglia, Robert Bosch GmbH, Germany
Paulo Pedreiras, University of Aveiro, Aveiro, Portugal
Shahin Roozkhosh, Boston University, MA, USA
Mouhammad Sakr, University of Luxembourg, Luxembourg
Ehsan Shahri, University of Aveiro, Aveiro, Portugal
Mohsen Shekarisaz, University of Tehran, Iran
Parul Sohal, Boston University, MA, USA
Alexander Stegmeier, University of Augsburg, Germany
Mário Sousa, University of Porto, Porto, Portugal
Tilmann Unte, University of Augsburg, Germany
Hao Zhang, North Carolina State University, USA

List of Authors

Shorouk Abdelhalim (17)
McMaster University, Hamilton, Canada

Jaume Abella (5)
Barcelona Supercomputing Center (BSC), Spain

Kunal Agrawal (3)
Washington University in Saint Louis, MO, USA

Shareef Ahmed (16)
University of North Carolina at Chapel Hill, NC, USA

Sebastian Altmeyer (19)
Universität Augsburg, Germany

Abderaouf N Amalou (7)
Univ. Rennes, INRIA, CNRS, IRISA, France

James H. Anderson (16)
University of North Carolina at Chapel Hill, NC, USA

Étienne André (19)
Université Sorbonne Paris Nord, LIPN, CNRS UMR 7030, F-93430 Villetaneuse, France

Federico Aromolo (18)
Scuola Superiore Sant'Anna, Pisa, Italy

Sanjoy Baruah (2, 3)
Washington University in Saint Louis, MO, USA

Michael A. Bender (3)
Stony Brook University, NY, USA

Ashikahmed Bhuiyan (8)
Department of Computer Science, West Chester University, PA, USA

Alessandro Biondi (9, 18)
TeCIP Institute, Scuola Superiore Sant'Anna, Pisa, Italy; Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Pisa, Italy

Timothy Bourke (1)
Inria Paris, France; Ecole normale supérieure, PSL University, CNRS, Paris, France

Vincent Bregeon (1)
Airbus Operations S.A.S., Toulouse, France

Patrick Carpanedo (13)
Boston University, MA, USA

Daniel Casini (9)
TeCIP Institute, Scuola Superiore Sant'Anna, Pisa, Italy; Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Pisa, Italy

Francisco J. Cazorla (5)
Barcelona Supercomputing Center (BSC), Spain; Rapita Systems S.L., Barcelona, Spain

Jian-Jia Chen (10)
Lamarr Institute for Machine Learning and Artificial Intelligence, Dortmund, Germany; Department of Computer Science, TU Dortmund University, Germany

Kuan-Hsun Chen (10)
University of Twente, The Netherlands

Weifan Chen (13)
Boston University, MA, USA

Silvano Dal Zilio (19)
Univ. de Toulouse, INSA, LAAS, F-31400 Toulouse, France

Dakshina Dasari (4)
Robert Bosch GmbH, Stuttgart, Germany

Eva Dengler (6)
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Marco Di Natale (9)
TeCIP Institute, Scuola Superiore Sant'Anna, Pisa, Italy; Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Pisa, Italy

Pontus Ekberg (2)
Uppsala University, Sweden

Loïc Fejoz (19)
RealTime-at-Work (RTaW), 615, Rue du Jardin Botanique, F-54600 Villers-lès-Nancy, France

Elisa Fromont (7)
Univ. Rennes, IUF, INRIA, CNRS, IRISA, France

Sergio Garcia-Esteban (5)
Polytechnic University of Catalonia, Barcelona, Spain; Barcelona Supercomputing Center (BSC), Spain

35th Euromicro Conference on Real-Time Systems (ECRTS 2023).
Editor: Alessandro V. Papadopoulos
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Maximilian Gaukler (11)
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Danesh Germchi (17)
University of Waterloo, Canada

Andreas Gerstlauer (4)
The University of Texas at Austin, TX, USA

Rafal Graczyk (14)
Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg

Susanne Graf (19)
Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, F-38000 Grenoble, France

Zhishan Guo (8)
Department of Computer Science, North Carolina State University, Raleigh, NC, USA

J. Javier Gutiérrez (19)
Universidad de Cantabria, Santander, Spain

Mario Günzel (10)
Department of Computer Science, TU Dortmund University, Germany

Michael González Harbour (19)
Universidad de Cantabria, Santander, Spain

Mohamed Hassan (17)
McMaster University, Hamilton, Canada

Rafik Henia (19)
Thales Research & Technology, F-91767 Palaiseau, France

Denis Hoornaert (4, 13)
Technische Universität München, Germany

Mohamed Hossam (17)
McMaster University, Hamilton, Canada

Ivan Izhbirdeev (13)
Boston University, MA, USA

Angeliki Kritikakou (15)
Univ Rennes, Inria, IRISA, CNRS, France

Didier Le Botlan (19)
Univ. de Toulouse, INSA, LAAS, F-31400 Toulouse, France

Jing Li (8)
Department of Computer Science, New Jersey Institute of Technology, Newark, NJ, USA

Giuseppe Lipari (19)
Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Federico Lucchetti (14)
Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg

Renato Mancuso (4, 13)
Boston University, MA, USA

Alberto Marchetti-Spaccamela (3)
Sapienza Università di Roma, Italy

Aleksandar Matovic (14)
Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg

Julio Medina (19)
Universidad de Cantabria, Santander, Spain

Enrico Mezzetti (5)
Barcelona Supercomputing Center (BSC), Spain; Rapita Systems S.L., Barcelona, Spain

Venkata Prashant Modekurthy (8)
Department of Computer Science, University of Nevada, Las Vegas, NV, USA

Daniel Mueller-Gritschneder (4)
Technische Universität München, Germany

Nicolas Navet (19)
University of Luxembourg, Luxembourg

Geoffrey Nelissen (18)
Eindhoven University of Technology, Eindhoven, The Netherlands

Pegdwende Romaric Nikiema (15)
Univ Rennes, Inria, IRISA, CNRS, France

Paul Nikolaus (12)
Distributed Computer Systems Lab (DISCO), TU Kaiserslautern, Germany

Rodolfo Pellizzoni (17)
University of Waterloo, Canada

Mohammad Pivezhandi (8)
Department of Computer Science, Wayne State University, Detroit, MI, USA

Marc Pouzet (1)
Ecole normale supérieure, PSL University, CNRS, Paris, France; Inria Paris, France

Isabelle Puaut (7)
Univ. Rennes, INRIA, CNRS, IRISA, France

Sophie Quinton (19)
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, F-38000 Grenoble, France

Phillip Raffeck (6)
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Tim Rheinfels (11)
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Juan M. Rivas (19)
Universidad de Cantabria, Santander, Spain

Shahin Roozkhosh (13)
Boston University, MA, USA

Ahsan Saeed (4)
Robert Bosch GmbH, Stuttgart, Germany

Abusayeed Saifullah (8)
Department of Computer Science, Wayne State University, Detroit, MI,USA

Ulf Schlichtmann (4)
Technische Universität München, Germany

Simon Schuster (6)
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Gerlando Sciangula (9)
TeCIP Institute, Scuola Superiore Sant'Anna, Pisa, Italy; Huawei Research Center, Pisa, Italy

Claudio Scordino (9)
Huawei Research Center, Pisa, Italy

Olivier Sentieys (15)
Univ Rennes, Inria, IRISA, CNRS, France

Alejandro Serrano-Cases (5)
Barcelona Supercomputing Center (BSC), Spain; Rapita Systems S.L., Barcelona, Spain

Sanskriti Sharma (13)
Boston University, MA, USA

Giovanni Stea (12)
Dipartimento di Ingegneria dell'Informazione, University of Pisa, Italy

Youcheng Sun (19)
The University of Manchester, UK

Harun Teper (10)
Department of Computer Science, TU Dortmund University, Germany

Marcello Traiola (15)
Univ Rennes, Inria, IRISA, CNRS, France

Peter Ulbrich (11)
TU Dortmund, Germany

Georg von der Brüggen (10)
Department of Computer Science, TU Dortmund University, Germany

Marcus Völp (14)
Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg

Peter Wägemann (6)
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Dirk Ziegenbein (4)
Robert Bosch GmbH, Stuttgart, Germany

Raffaele Zippo (12)
Dipartimento di Ingegneria dell'Informazione, University of Firenze, Italy; Dipartimento di Ingegneria dell'Informazione, University of Pisa, Italy; Distributed Computer Systems Lab (DISCO), TU Kaiserslautern, Germany

Scheduling and Compiling Rate-Synchronous Programs with End-To-End Latency Constraints

Timothy Bourke
Inria Paris, France
Ecole normale supérieure, PSL University, CNRS, Paris, France

Vincent Bregeon
Airbus Operations S.A.S., Toulouse, France

Marc Pouzet
Ecole normale supérieure, PSL University, CNRS, Paris, France
Inria Paris, France

Abstract

We present an extension of the synchronous-reactive model for specifying multi-rate systems. A set of periodically executed components and their communication dependencies are expressed in a Lustre-like programming language with features for load balancing, resource limiting, and specifying end-to-end latencies. The language abstracts from execution time and phase offsets. This permits simple clock typing rules and a stream-based semantics, but requires each component to execute within an overall base period. A program is compiled to a single periodic task in two stages. First, Integer Linear Programming is used to determine phase offsets using standard encodings for dependencies and load balancing, and a novel encoding for end-to-end latency. Second, a code generation scheme is adapted to produce step functions. As a result, components are synchronous relative to their respective rates, but not necessarily simultaneous relative to the base period. This approach has been implemented in a prototype compiler and validated on an industrial application.

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

Keywords and phrases synchronous-reactive, integer linear programming, code generation

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.1

1 Introduction

Embedded control software is often designed as a set of components that each repeatedly sample inputs, compute a transition function, and update outputs. Such components must be scheduled so as to share processor resources while respecting timing and communication requirements. Scheduling determines how data propagates along chains of components from sensor acquisitions, through successive computations, to corresponding actuator emissions. The end-to-end latencies of such chains are crucial to overall system performance.

We characterize and extend an approach for developing avionics software based on the synchronous-reactive languages Lustre [29] and Scade [13]. Our application model comprises (i) a set of components whose execution rates are specified as unit fractions ($1/n$) of a base rate, and (ii) a graph of data flow between components. The Worst-Case Execution Time (WCET) of each component must be less than the base period. This is a significant restriction, but one that is acceptable for safety-critical avionics applications. The implementation target is one or more sequential step functions called cyclically to, in turn, call individual component step functions. Data is exchanged by reading and writing static variables.

Besides providing a way to specify real-time behavior, execution rates allow implementations to balance requirements and resources. For example, in the absence of other constraints, a component at rate $1/3$ can be scheduled to run every 3 cycles with any of the phases 0, 1,

© Timothy Bourke, Vincent Bregeon, and Marc Pouzet;
licensed under Creative Commons License CC-BY 4.0

or 2. Such choices are made to distribute total computation load over successive cycles, to implement the dataflow specification by ordering variable reads and writes, and to respect resource bounds such as, for example, the capacity of an avionics bus. We show how it is also possible to choose component phases so as to satisfy overall end-to-end latency requirements.

In our approach, scheduling occurs in two stages. The first assigns components to phases and the second orders the components within a cycle. We realize the first stage by generating and solving an Integer Linear Programming (ILP) problem and the second by adapting an algorithm [4] used to compile Lustre and Scade. Using the ILP encoding presented in this article, offline scheduling is restricted by end-to-end latency constraints declared in source programs. As a result, if scheduling succeeds, these source-level constraints are respected by the generated code. We have implemented the presented techniques in a prototype compiler and validated them on a large flight control program.

Industrial context

This article characterizes an industrial approach to developing avionics software. We focus on a flight control and guidance system that is developed as follows. The control laws and monitoring functions are specified in the Scade language. The resulting design comprises approximately 5000 individual components communicating over 120 000 named signals. A component is a *block diagram* comprising blocks and lines: blocks represent basic arithmetic operations, unit delays, filters, etcetera; lines connect block outputs to block inputs. A code generator transforms each Scade component into a C function that reads and writes static variables corresponding to its input signals, output signals, and internal states. Preemption and dynamic scheduling are rigorously avoided to simplify reasoning and testing. The resulting code is then implemented on an embedded platform.

Scade programs are compiled following the synchronous paradigm: code generation produces *step functions* for cyclic execution. Since it is not feasible to execute all 5000 components in a single cycle of the platform, each is executed at an integer multiple (2, 4, 8, 24, or 48) of the base period of 5 ms and scheduled to distribute the computational load. It is crucial that (a) the aggregate of computations executed within a cycle does not exceed the base period; and (b) the final system strictly respects end-to-end latency constraints on servo control loops. The first constraint is taken into account during scheduling and validated on the generated executable using WCET analysis. End-to-end latencies are currently specified indirectly by application engineers who assign execution orders and bounds to certain function sequences. These indirect constraints, and platform limitations related to the avionics bus, are encoded by software engineers into an ILP problem that is solved to give a schedule.

Our work systematizes and streamlines the process described above. The idea is to specify the system as a single program that instantiates the 5000 individual functions together with constraints on resources and end-to-end latencies. In this way, we clarify the overall semantics, allow direct specification of end-to-end latencies, and automate compilation. We have successfully applied our approach to the avionics system, but industrial constraints prevent us reporting the details, so we instead focus on the ROSACE case study [51]. It has only 11 components but is representative of the domain and makes for a good example.

Although our work is guided by a specific application, we believe it is applicable more generally. For instance, many companies develop control applications as Simulink block diagrams and either manually reprogram or automatically generate code for *single-tasking execution* [46, §4-6]. Other applications are developed using a similar non-preemptive tasking model even though they are not specified explicitly as block diagrams. Examples include the open-source ArduPilot [1] and Paparazzi UAV [5] projects.

2 A rate-synchronous model

The first part of this section presents a variant of Lustre [29] with unit-fraction clocks. Unlike in similar programming languages [10, 15, 33, 45, 55], our clocks specify a rate without a phase. This is natural for real-time scheduling where release times may implement data dependencies [9 | 6, §3.5.2]. Our proposition is inspired by Prelude [22, 24] but restricts communication primitives and generates sequential code rather than real-time tasks.

The last part of this section presents our version of ROSACE and the results of the scheduling and code generation techniques which are detailed in the remainder of the article.

2.1 Syntax

As in any Lustre-like language, a function is defined by a set of mutually recursive equations.

$$eq ::= x \texttt{ = } e \mid x^* \texttt{ = } f\texttt{(}e^*\texttt{)}$$

A *basic equation* defines a variable using an expression. An *instantiation* of a function f defines the function inputs using a list of expressions and associates each output to a variable.

An *expression* is a constant, a variable, an application of a unary or binary operator, a conditional expression, the previous value of a variable, a sampling of a faster variable, a sampling of the previous value of a faster variable, or a buffering of a slower variable.

$$\begin{aligned} e ::= &\ c \mid x \mid \diamond\, e \mid e \oplus e \mid \texttt{if } e \texttt{ then } e \texttt{ else } e \mid \texttt{last } x \\ \mid &\ x \texttt{ when } s \mid \texttt{(last } x\texttt{) when } s \mid \texttt{current(}x\texttt{, } s\texttt{)} \end{aligned}$$

$$s ::= \texttt{(}c \texttt{ \% } c\texttt{)} \mid \texttt{(? \% } c\texttt{)}$$

The `last` operator [12, 53] can only be applied to a variable, but otherwise has the same meaning as Lustre's `pre` operator: it delays a signal by one cycle relative to its rate. Unlike `pre`, the `last` operator can be directly translated into flow graphs, as will be seen when they are introduced in Section 3.1, and directly implemented by a shared variable. It may only be applied to a variable declared with an initial last value.

For similar reasons, our `when` operator only applies to a variable or `last` expression. The *sample choice* argument s defines how to sample incoming values. For instance, $1 \% 4$ selects the second, numbered from zero, of every four values and $? \% 3$ lets offline scheduling determine which of every three values to sample. This choice is then fixed throughout all executions of the generated code. For $m \% n$, it must be that $0 \leq m < n$ and $1 < n$.

Our `current` operator buffers the value of a variable x, which must have an initial last value. The sample choice s determines how to hold incoming values. For instance, $2 \% 4$ specifies to repeat the initial value twice before repeating each input value four times and $? \% 4$ lets scheduling determine how many times to repeat the initial value.

A program comprises a list of declarations of resources, external functions, and functions.

$$p ::= (d \texttt{ ;})^*$$

$$\begin{aligned} d ::= &\ \texttt{resource } x \texttt{ : } ty \\ \mid &\ \texttt{node } f\texttt{(} (x \texttt{ : } ty \texttt{ ;})^* \texttt{) returns (} (x \texttt{ : } ty \texttt{ ;})^* \texttt{) requires (} (x \texttt{ = } c \texttt{ ;})^* \texttt{)} \\ \mid &\ \texttt{node } f\texttt{(} (x \texttt{ : } ty \texttt{ :: } ck \texttt{ [last = } c\texttt{] ;})^* \texttt{) returns (} (x \texttt{ : } ty \texttt{ :: } ck \texttt{ [last = } c\texttt{] ;})^* \texttt{)} \\ &\ \texttt{var } (x \texttt{ : } ty \texttt{ :: } ck \texttt{ [last = } c\texttt{] ;})^* \texttt{ let } (((pragmas\ eq) \mid cst) \texttt{ ;})^+ \texttt{ tel} \end{aligned}$$

$$pragmas ::= [\texttt{label(}x\texttt{)}]\ [\texttt{phase(}c \texttt{ \% } c\texttt{)}]$$

A *resource declaration* introduces a name x for an integer or floating-point quantity that the scheduler must take into account. For example, `busout` for the number of digital outputs to be sent on an avionics bus, or `cpu` for a measure of processor load.

An *external function declaration* specifies the name f of an external function together with its input/output interface and its resource requirements. The inputs and outputs are each specified by a list of variable names and their types. The resource requirements are a list of resource names, each paired with a constant quantity.

A *function definition* specifies the name f and input/output interface and defines its implementation as a list of local variables and a list of equations and constraints. Each variable x has a declared type ty, clock type ck, and, optionally, an initial last value c (denoted $x_{\text{-}1} = c$). We only consider primitive types, namely ty ::= `bool` | `int` | `float`, and unit-fraction clocks ck ::= `1/`c, where `1/1`, normally written as `1`, represents the base rate of a function and `1/`c represents a fraction of the base rate. Expressions may refer to input, local, and output variables. Equations define local and output variables only. Each local and output variable must appear at left of exactly one equation. An equation may be preceded by pragmas: `label` specifies a unique identifier and `phase` fixes the schedule. The label of an instantiation x^* `=` f `(` e^* `)` defaults to f when not ambiguous. In addition to equations, the definition may also include resource and timing constraints.

cst ::= `resource balance` x
  | `resource` x *rel* c
  | `latency` (`exists` | `forward` | `backward`) *rel* c `(` x `,` x (`,` x)* `)`

rel ::= `<=` | `<` | `=` | `>` | `>=`

A *balance constraint* directs the scheduler to minimize, across cycles, differences in the sum of a given resource, like `cpu`. A *resource constraint* places a constant bound c on the sum of a resource, like `busout`, in a single cycle. A *latency constraint* sets a constant bound `c` on the end-to-end latency of one instance, or all forward or backward instances, of a chain. A *chain* is a sequence of equations, eq_0, eq_1, ..., eq_{n-1} where at least one of the variables defined at left of eq_i appears in an expression at right of equation eq_{i+1}.

2.2 Semantics

The focus here is not on programming languages, so we only outline the main principles. In a dataflow semantics [34], expressions are associated with sequences of values, equations associate variables to sequences, and functions map sequences to sequences. In a synchronous dataflow semantics [7], infinite sequences, or streams, are aligned. The idea is that they are calculated together over successive rounds. Streams are often presented in grids with rows for expressions and columns for rounds. Alignment may be shown in a grid by leaving gaps. In our model, slower streams are shown by placing their values in wider columns: they are synchronous at their rate and "simultaneous" with multiple values of faster streams.

Consider a simple example adapted from Forget [21, Figure 5.1] and sketched in Figure 1a. There is a fast component that executes every cycle and a slow one that executes every three cycles. We instantiate the two components below.

```
node eg1() returns ()
var vf : int :: 1;
    vs : int :: 1/3 last = 0;
     n : int :: 1 last = 0;
let
  n = (last n) + 1;
  vf = n + current(vs, (2 % 3));
  vs = (vf when (1 % 3)) + 5;
tel
```

fast :: 1 —vf→ slow :: 1/3 —vs→ fast :: 1

(a) Fast and slow components.

vf	1	2	10	11	12	23	24	25	39	...
n	1	2	3	4	5	6	7	8	9	...
vs	7			17			30			...

(b) Synchronized streams in the rate-synchronous model.

Figure 1 Structure and trace of `node eg1()`.

$$[\![e_1 \oplus e_2]\!]\,(i) = [\![e_1]\!]\,(i) \oplus [\![e_2]\!]\,(i)$$

$$[\![\texttt{last}\ x]\!]\,(i) = \begin{cases} x_{\text{-}1} & \text{if } i = 0 \\ [\![x]\!]\,(i-1) & \text{otherwise} \end{cases}$$

$$[\![x\ \texttt{when}\ (s \,\%\, n)]\!]\,(i) = [\![x]\!]\,(n \cdot i + s)$$

$$[\![\texttt{current}(x, (s \,\%\, n))]\!]\,(i) = \begin{cases} x_{\text{-}1} & \text{if } i < s \\ [\![x]\!]\,(\lfloor \frac{i-s}{n} \rfloor) & \text{otherwise} \end{cases}$$

$$[\![(\texttt{last}\ x)\ \texttt{when}\ (s \,\%\, n)]\!]\,(i) = \begin{cases} x_{\text{-}1} & \text{if } i + s = 0 \\ x(n \cdot i + s - 1) & \text{otherwise} \end{cases}$$

(a) Stream-based semantics (x_{-1} is a declared initial last value).

$$\frac{e_1 :: 1/n \qquad e_2 :: 1/n}{e_1 \oplus e_2 :: 1/n}$$

$$\frac{x :: 1/n}{\texttt{last}\ x :: 1/n}$$

$$\frac{x :: 1/m}{x\ \texttt{when}\ (\cdot \,\%\, n) :: 1/mn}$$

$$\frac{x :: 1/mn}{\texttt{current}(x, (\cdot \,\%\, n)) :: 1/m}$$

(b) Clock typing rules.

Figure 2 Key formal definitions for expressions.

The `vf` and `vs` signals are declared with their execution rates. The local variable `n` is a counter used by the fast component. Both `vs` and `n` are declared with initial last values, so that they can be used with the `last` and `current` operators. Rate transitions are expressed with `current` (slow-to-fast) and `when` (fast-to-slow) operators. The fast value, `vf`, sums the counter with the initial last value of `vs` repeated twice and then each of its values repeated three times. The slow value, `vs`, adds five to the second of every three fast values. Figure 1b shows the resulting streams.

The grid shown in Figure 1b gives the meaning of the program. Each row in the grid associates a variable with a stream. This intuitive idea is made precise in Figure 2a by a semantic function $[\![\cdot]\!]$ that maps each syntactic element to a stream ($\mathbb{N}_0 \to \mathbb{V}$); that is, to a function from a cycle number i to the value of the expression in that cycle. The semantic function distributes over expressions to return a constant function for literals c or the value of an input or equation for variables x (not shown). A binary operator $\oplus$ is applied pointwise to its input streams. The `last` operator returns the initial last or preceding value of the named stream. The `when` operator selects one of every n input values. The `current` operator repeats values from the input stream with a special case for the first s values. The rule for $(\texttt{last}\ x)\ \texttt{when}\ (s \,\%\, n)$ can be derived by substitution from the rules for `when` and `last`. If a program contains $?\,\%\,n$, the s in the corresponding stream equation can take any value in $[0, n)$ and the semantic rules may admit more than one solution.

An expression like $x + (x\ \texttt{when}\ (0 \,\%\, 3))$ has a well-defined value according to the semantic rules but cannot be implemented without an unbounded buffer. Programs that require unbounded memory are unsuited to embedded control applications. In Lustre-like languages, a *clock type system* prescribes how streams may be combined [7] and thereby which programs are accepted for compilation. Every expression is associated with a rate by syntax-directed rules that define acceptable expressions. Figure 2b shows the main rules for the presented

language. For binary operators, the two input expressions (above the line) and the output expression (below the line) must all have the same rate ($1/n$). A similar rule, not shown, applies to function instances. The `last` operator does not change the rate of its argument, and the rate transition operators respectively divide or multiply the rate of an input argument.

Communications between non-harmonic rates require an intermediate equation at a common multiple of both rates. For example, between a writer `w :: 1/6` and a reader `r :: 1/8`, one could add an explicit buffer `b :: 1/2`.

```
b = current(w, (? % 3));
r = b when (? % 4);
```

Finally, there are well-clocked programs that have no semantics. For instance, in the `eg1` node, replacing `last n` by `n` or exchanging the $2 \% 3$ and $1 \% 3$ sample choices results in cyclic definitions for which there are no solutions. There are also correct programs that our current code generator cannot handle. For instance, changing the definition of `vs` to `vf when` $(0 \% 2) +$ `vf when` $(1 \% 2)$ gives a valid program that cannot be implemented using a single shared variable for `vf`. We do not pursue these issues here: scheduling simply fails for such programs.

2.3 Compilation

The source language allows specifying and reasoning about programs in terms of streams of values. Generating code for such programs and defining end-to-end latencies requires a change of perspective. We now want to consider a program as a set of components that repeatedly read and write shared variables. Each iteration of the program is termed a *cycle*. We will also refer to the *hyperperiod* of a set of equations hp, which is the Least Common Multiple (LCM) of their periods, that is, of the denominators of their rates. In the source semantics, each variable is associated with a single value for the duration of its *round* (that is, during one period / within one dataflow grid column), but the corresponding shared variable is updated in a specific cycle when the code generated for its defining equation executes. Execution order now becomes paramount: each equation must be assigned a phase relative to its execution rate and ordered relative to other equations for execution within a cycle.

The following C code was generated for `eg1`.

```
static int c = 0;

void step() {
  static int vs = 0, n = 0;
  int vf;

  n = n + 1;
  vf = n + vs;
  if (c == 1) { vs = vf + 5; }
  c = (c + 1) % 3;
}
```

A static variable `c` is introduced to count off successive phases of the hypercycle ($hp = 3$). Static variables are declared for `c` and `vs`. Their values persist across cycles. A local variable is declared for `vf` since its value is only needed within a cycle. In every cycle, `n` is updated first, then `vf`, and then, but only in the second of every three cycles, `vs`. The new value of `vs` is not used until the subsequent cycle.

The assignment of equations to phases, termed *scheduling*, and the ordering of equations within a cycle, termed *microscheduling*, are central to the compilation scheme presented in the following sections on constraint and code generation. First, though, we apply the specification language to an existing case study which will serve as a running example.

2.4 Example: ROSACE

The ROSACE case study [51] considers the development process of a longitudinal flight controller. Figure 3a shows our reimplementation of the original Prelude program [51, Figure 3 and §III.B]. Since the fastest components run at 200 Hz, that is, with a period of 5 ms, we set the cycle period to 2.5 ms to allow load balancing. Inputs are declared for the desired altitude `h_c` and speed `va_c`, both at 10 Hz. Outputs are declared for the throttle `d_th_c` and elevator deflection `d_e_c` commands, both at 50 Hz. Local signals are declared with explicit rates. The node body contains three groupings of components. At 200 Hz are the `elevator`, `engine`, and `dynamics` components that provide a discretized model of the environment. Such components would not normally be included in a controller, but we maintain them for the sake of the example. At 100 Hz are five filters on altitude `h`, vertical acceleration `az`, pitch rate `q`, vertical speed `vz`, and true airspeed `va`. At 50 Hz are the control laws for tracking the requested altitude `alt_hold`, vertical speed `vz_control`, and airspeed `va_control`. We use free sample choices (?) in `when` and `current` rate transitions to give greater scheduling freedom at the cost of underspecification.

Figure 3c graphs the data flows between components. There is a vertex for each equation. An arc indicates that a signal defined by the "tail" equation is used in the "head" equation. For example, there is an arc from `az_filter` to `vz_control` due to the `az_f` signal.

Strictly speaking, either the clock types of the local variables or the sample rates of the `when` and `current` operators could be inferred by the compiler; as in Prelude or Simulink. In this work, we prefer to state them explicitly. The compiler detects and signals any discrepancies when it checks clock types after parsing a source file.

The node body also contains two constraints. A given task chain must execute within 5 ms every 20 ms [51, §III.C], that is, with an end-to-end latency ≤ 2 cycles. The `exists` keyword specifies that only one instance of the chain per hypercycle need satisfy the bound. A resource called `ops` must be balanced across cycles. We calculated `ops` values in a simple way from the Prelude definitions: +10 for each libm function, node instance, or if/then/else; +3 for each multiplication or division; +1 for other operators and each equation. The weights, see Figure 3d, are added to external function declarations for each component, for example,

```
node alt_hold (h_c, h_f : float)
returns (vz_c : float)
requires (ops = 201);
```

Figure 3d also shows the scheduled phases. Since the `dynamics` node requires many more operations than the others, it has been scheduled with `elevator` in odd cycles. These two components are scheduled in the same cycle due to the tight latency constraint. The other components are scheduled in even cycles such that dependency and latency constraints are satisfied. Better load balancing could be obtained by inlining the `dynamics` and `alt_hold` nodes, and halving the cycle period while doubling all the rates.

The schedule is realized in the generated code, Figure 3b, via the guards of `if` and `switch` statements. The optimization of conditionals can be reduced so that `if (c % 4 == 2) {···}` is after, rather than within, `if (c % 2 == 0) {···}`, and the grouping by period is retained. Microscheduling determines the order of function calls for a given value of the counter `c`. Notably, `elevator` runs before `dynamics`; the `*_filter` run before `va_control`, `vz_control`, and `alt_hold`; and `alt_hold` runs before `vz_control`. In these cases, an output calculated by one component is propagated in a cycle to become the input for another. Conversely, the outputs calculated by `dynamics` are sampled less often by the `*_filter` and with a delay of one cycle. In this code, signals are implemented by reading and writing static variables inside components. The compiler can also generate code that passes values on the stack.

```
node assemblage(
   h_c : float :: 1/40 last = 0.;
  va_c : float :: 1/40 last = 0.)
returns (
  d_th_c : float :: 1/8 last = 1.6402;
  d_e_c  : float :: 1/8 last = 0.0186)
var
  vz_c : float :: 1/8;
  d_e, th, h, az, va, q, vz : float :: 1/2;
  vz_f, va_f, h_f, az_f, q_f : float :: 1/4;
let
  (* 200Hz = 1/2 *)
  d_e = elevator(current(d_e_c, (? % 4)));
  th = engine(current(d_th_c, (? % 4)));
  (va, az, q, vz, h) = dynamics(th, d_e);
  (* 100Hz = 1/4 *)
   h_f =  h_filter( h when (? % 2));
  az_f = az_filter(az when (? % 2)); ...
  (* 50Hz = 1/8 *)
  vz_c = alt_hold(current(h_c, (? % 5)),
                h_f when (? % 2));
  d_e_c = vz_control(vz_c,
              vz_f when (? % 2),
              q_f when (? % 2),
              az_f when (? % 2));
  d_th_c = va_control(
              current(va_c, (? % 5)),
              va_f when (? % 2),
              q_f when (? % 2),
              vz_f when (? % 2));

  latency exists <= 2 (dynamics, h_filter,
              alt_hold, vz_control, elevator);
  resource balance ops;
tel
```

(a) Source program.

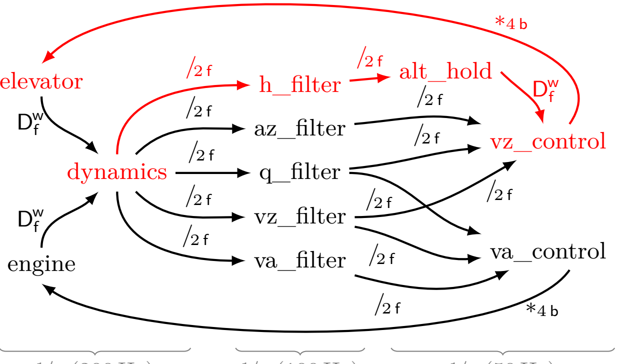
```
static int c = 0;

static float h_c = 0,
    va_c = 0;
    d_th_c = 1.6402,
    d_e_c = 0.0186
    vz_c, ..., q_f;

void step0()
{
  if (c % 2 == 0) {
    engine();
    if (c % 4 == 2) {
      vz_filter();
      h_filter();
      va_filter();
      q_filter();
      az_filter();
    }
  } else {
    elevator();
    dynamics();
  }
  switch (c) {
  case 2:
    va_control();
    break;
  case 6:
    alt_hold();
    vz_control();
    break;
  }

  c = (c + 1) % 8;
}
```

(b) Generated code.



(c) Flow graph.

	ops	*phase*	
elevator	98	1 % 2	(p_e)
engine	82	0 % 2	
dynamics	1174	1 % 2	(p_d)
h_filter	38	2 % 4	(p_h)
az_filter	37	2 % 4	
q_filter	37	2 % 4	
vz_filter	37	2 % 4	
va_filter	38	2 % 4	
alt_hold	201	6 % 8	(p_a)
vz_control	88	6 % 8	(p_v)
va_control	90	2 % 8	

(d) Schedule.

Figure 3 Main ROSACE [51, Figure 3] node.

3 Constraint generation

A source program is transformed into sequential code in three steps. First, the compiler is invoked to generate an ILP encoding of the constraints on equation phases. Second, the encoding is passed to an external solver like the IBM ILOG CPLEX Optimizer [32]. Third, if the solver finds a solution, it is returned to the compiler which then generates code. Both compiler invocations manipulate a flow graph constructed from the program source. We first describe the flow graphs before presenting the ILP encodings of data dependencies, resource constraints, and latency chains. The second compiler invocation is addressed in Section 4.

3.1 Flow graphs

As an intermediate step in the passage from dataflow programs to sequential code, we adapt the standard definition of flow graphs by labeling edges with sampling and microscheduling information. Subsequent definitions and reasoning are in terms of flow graphs, which thus provide a way to apply the results independently of the source language.

A *flow graph* is a directed graph (V, A) with labeled arcs $A \subseteq V \times S \times C \times V$. The first label specifies the *sampling*: $S = \{\mathsf{D}^{\mathsf{w}}, \mathsf{D}^{\mathsf{r}}, \mathsf{D}^{?}, /_n, /^{\mathsf{L}}_n, *_n, \ldots\}$ with $n \in \{2, 3, \ldots\}$; and the second one specifies the *concomitance*: $C = \{\mathsf{f}, \mathsf{b}\}$. Vertexes V represent equation labels, which for simplicity we conflate with the equations themselves. An arc from eq_w to eq_r indicates that a variable defined by eq_w appears in the defining expression of eq_r. That is, values flow from eq_w to eq_r, or, in implementation terms, eq_w writes shared variables that eq_r reads. The flow graph for the ROSACE example is shown in Figure 3c.

There are three types of sampling. *Direct sampling* indicates equations at the same rate: D^{w} is the standard case where writing must occur before reading; D^{r} is used to encode the `last` operator, where variables must be read before they are written; and $\mathsf{D}^{?}$ indicates that writing and reading may be scheduled in any order, but that the concomitance is important, otherwise the arc could simply be omitted. *Fast-to-slow sampling*, $/_n$, indicates that the writer has a higher rate than the reader, the subscript, n, is the relative ratio. The $/^{\mathsf{L}}_n$ form additionally specifies sampling of the `last` operator. *Slow-to-fast sampling*, $*_n$, indicates that the writer is slower than the reader. The symbols are adapted from Prelude [21, §4.2.2] and recall the clocking rules of Figure 2b. A `when` divides the writer rate by sampling a subset of its values. A `current` multiplies the writer rate by duplicating (buffering) input values.

For scheduling and microscheduling to work coherently they must agree on the order of related equations within a single phase. The concomitance labels C solve this problem by fixing the intraphase ordering prior to scheduling. They specify how two equations will be ordered if ever they execute in the same cycle: f, *forward concomitance*, constrains the write to occur before the read, immediately communicating a value; and b, *backward concomitance*, constrains the write to occur after the read, delaying the communication by one period.

For our source language, the generation of a flow graph from a function definition is immediate. For each equation eq_r, we simply descend into the defining expression $(x = e)$ or argument expressions $(x^* = f(e^*))$, ignoring constants and continuing recursively through operators and conditionals. The remaining cases all involve a variable x. If the variable is an input, or `last` y for y defined by eq_r, nothing is done, otherwise its defining equation eq_w is identified and an arc is added to the flow graph as per the cases:

x	$eq_w \xrightarrow{\mathsf{D}^{\mathsf{w}}_{\mathsf{f}}} eq_r$	$(\texttt{last}\ x)\ \texttt{when}\ (\cdot \,\%\, n)$	$eq_w \xrightarrow{/^{\mathsf{L}}_{n}\,\mathsf{b}} eq_r$	
$\texttt{last}\ x$	$eq_w \xrightarrow{\mathsf{D}^{\mathsf{r}}_{\mathsf{b}}} eq_r$	$\texttt{current}(x, (\cdot \,\%\, n))$	$eq_w \xrightarrow{*_{n}\,\mathsf{f}} eq_r$	(by default)
$x\ \texttt{when}\ (\cdot \,\%\, n)$	$eq_w \xrightarrow{/_{n}\,\mathsf{f}} eq_r$		$eq_w \xrightarrow{*_{n}\,\mathsf{b}} eq_r$	(if "fast-first")

The `current` operator normally has forward concomitance, but a "fast-first" option that makes it backward permits equations to be ordered within a cycle from fastest to slowest.

Clock typing almost guarantees the absence of arcs with different labels between two equations, but it is still necessary to reject equations that read both x and `last` x, for example, $y = x + \texttt{last}\ x$. Their compilation requires extra buffering: $lx = \texttt{last}\ x$, $y = x + lx$.

3.1.1 Circular dependencies

Equations with circular dependencies, or "algebraic loops" [45, §3–39], are normally rejected since a unique solution does not exist or cannot be found without iteration. For example, no streams satisfy both $x = y + 1$ and $y = x + 1$, and all pairs of identical streams satisfy $x = y$ and $y = x$. Typically, circular dependencies are broken by manually adding `last` operators. For example, the definitions $x = \texttt{last}\ y + 1$ and $y = x + 1$ are valid.

A Lustre program is analyzed by checking a graph of its static dependencies for cycles [29, §III.A] [4, §3.1]. A *dependency graph* is obtained from a flow graph by reversing all edges with backward concomitance. There are two kinds of cycles in such graphs. A *direct cycle* only contains arcs with direct sampling labels. All the equations in such a cycle have the same rate, will necessarily be scheduled in the same phase, and cannot be microscheduled. An *inter-rate* cycle contains at least one slow-to-fast and one fast-to-slow arc.

3.1.1.1 Direct cycles

In the industrial context that motivates our work, the standard solution of breaking direct cycles by manually adding `last` operators is untenable. There are simply too many cycles, mostly due to feedback from monitoring and maintenance features, and too many variables. Furthermore, it does not usually matter whether the most recent or last value is read as the extra delay usually has no consequence on overall system behaviour. Many variables carry samples of signals that change slowly, or their contribution to feedback and output calculations is minimal. Manually breaking cycles complicates development and overconstrains scheduling.

As a solution to this problem, Wyss et al. [57] propose a *don't-care* operator `dc x` that the compiler resolves to `x` or `last x`. Iooss et al. [33, §5.1] adopt a similar solution with their `last`? x operator. We considered adding such an operator to our source language, but for the flight control system described in the introduction, we found that programmers would simply add this operator to all direct reads. For large programs, this complicates the source text without providing any real advantages.

Instead, we provide three compiler options that transform a flow graph prior to scheduling: (i) *relax same period*, (ii) *relax same period cycles*, and (iii) *cut same period cycles*. The first drops all direct arcs that are not needed for latency chains and relabels those in latency chains with $\mathsf{D}^?$. It has the same effect as replacing all variables x in expressions e with `last`? x. The second is similar but only applies to arcs within a strongly connected component (SCC) of the dependency graph. The third calculates a *minimum feedback arc set* of the dependency graph and inverses the concomitance of those arcs in the flow graph to eliminate all cycles. That is, it replaces some variable instances x with `last`? x. Finding the smallest such set is NP-hard for general graphs [35], so we adapt a heuristic [18, 19] that executes in time $O(|V| \cdot |A|)$ and produces a minimum feedback arc set that is at worst twice the size of a minimal one. In limited experiments, neither of the three transformations proved better than the others in terms of scheduling time or result quality. The compiler options provide a pragmatic solution to a practical problem but require deforming the source-level specification. It remains to be seen whether such applications can be specified in a more principled manner.

3.1.1.2 Inter-rate cycles

Identifying cyclic dependencies between equations at multiple rates is challenging. For example, the equations $x = y$ `when` $(1 \,\%\, 2)$ and $y =$ `current`$(x, (0 \,\%\, 2))$ have no solution since y must buffer the value of x in the first of every two rounds, but x may only sample y in the second one. Dependencies may even change over the course of a hypercycle. For now, we simply accept that constraint solving will fail for such programs.

When the generated constraints have a solution, however, we must ensure that microscheduling will succeed. We do so by transforming the original flow graph, prior to scheduling, to inverse the concomitance of all forward slow-to-fast arcs ($*_{n\,\mathsf{f}}$) between vertexes in the same SCC of the dependency graph. Then, even if interdependent equations are scheduled in the same phase, they can still be microscheduled. The semantics of the program is unchanged. This treatment occurs in Figure 3c where the arcs from `vz_control` to `elevator`, and from `va_control` to `engine` have backward concomitance. According to the schedule of Figure 3d, only `va_control` and `engine` may execute together in the same cycle, and `engine` then goes first, as in Figure 3b.

3.2 Dependency constraints

The flow graph produced from a program and modified as described above is translated into a set of ILP constraints. The encoding is unsurprising. The *phase* of an equation *eq* is represented by an integer variable $0 \leq \mathtt{p}_{eq} < \mathrm{period}(eq)$, where $\mathrm{period}(eq) = n$ for $eq :: {}^{1}\!/n$. Any solution found for the phase variables is a valid schedule. A variable is unnecessary if $\mathrm{period}(eq) = 1$. The substitution $\mathtt{p}_{eq} = 0$ is then applied to constraints and solutions.

In the following, we consider arbitrary pairs of equations eq_w and eq_r, where eq_w defines a variable w that eq_r uses to define a variable r. We will reason in terms of w and r, conflating variables and equations and ignoring the details of expressions. The generalization is trivial.

For equations of the form $r = w$, writing must occur before reading. For those of the form $r =$ `last` w, reading must occur before writing. Whether reading and writing may occur in the same phase or not, and thus the strictness of inequalities, depends on the concomitance. For example, for $\mathsf{D}^{\mathsf{w}}_{\mathsf{b}}$, w and r must not be scheduled in the same phase, since backward concomitance requires that microscheduling place the computation of r before that of w. Flow-graph arcs are thus translated to phase constraints as follows.

$$w \xrightarrow{\mathsf{D}^{\mathsf{w}}_{\mathsf{f}}} r \ \text{ becomes } \ \mathtt{p}_w \leq \mathtt{p}_r \qquad\qquad w \xrightarrow{\mathsf{D}^{\mathsf{r}}_{\mathsf{f}}} r \ \text{ becomes } \ \mathtt{p}_r < \mathtt{p}_w$$
$$w \xrightarrow{\mathsf{D}^{\mathsf{w}}_{\mathsf{b}}} r \ \text{ becomes } \ \mathtt{p}_w < \mathtt{p}_r \qquad\qquad w \xrightarrow{\mathsf{D}^{\mathsf{r}}_{\mathsf{b}}} r \ \text{ becomes } \ \mathtt{p}_r \leq \mathtt{p}_w$$

For an equation $r = w$ `when` $(i \,\%\, n)$, where $0 \leq i < n$, with forward concomitance, the read must occur with or after the ith write and strictly before any subsequent write:

$$i \cdot \mathrm{period}(w) + \mathtt{p}_w \quad \leq \quad \mathtt{p}_r \quad < \quad (i+1) \cdot \mathrm{period}(w) + \mathtt{p}_w \quad \text{for } w \xrightarrow{/_{n\,\mathsf{f}}} r.$$

For backward concomitance, $w \xrightarrow{/_{n\,\mathsf{b}}} r$, the strictnesses are reversed: $\cdots < \mathtt{p}_r \leq \cdots$.

For free sample choices, $r = w$ `when` $(? \,\%\, n)$, the rule is *write before first read*. Only the lower bound is needed and $i = 0$, giving $\mathtt{p}_w \leq \mathtt{p}_r$ for $w \xrightarrow{/_{n\,\mathsf{f}}} r$, and $\mathtt{p}_w < \mathtt{p}_r$ for $w \xrightarrow{/_{n\,\mathsf{b}}} r$.

For an equation $r = ($`last` $w)$ `when` $(i \,\%\, n)$, with backward concomitance, the read must occur strictly after any $(i-1)$th write and before or with the subsequent write:

$$(i-1) \cdot \mathrm{period}(w) + \mathtt{p}_w \quad < \quad \mathtt{p}_r \quad \leq \quad i \cdot \mathrm{period}(w) + \mathtt{p}_w \quad \text{for } w \xrightarrow{/^{\mathsf{L}}_{n\,\mathsf{b}}} r.$$

For $r = ($`last` $w)$ `when` $(? \,\%\, n)$, the rule is *read before last write*. Only the upper bound is needed and $i = n - 1 = \frac{\mathrm{period}(r)}{\mathrm{period}(w)} - 1$, giving $\mathtt{p}_r \leq \mathrm{period}(r) - \mathrm{period}(w) + \mathtt{p}_w$.

For an equation $r = \texttt{current}(w, (i \,\%\, n))$ with forward concomitance, the write must occur with or before the ith read but strictly after the preceding one:

$$(i-1) \cdot \text{period}(r) + \texttt{p}_r \quad < \quad \texttt{p}_w \quad \leq \quad i \cdot \text{period}(r) + \texttt{p}_r \quad \text{for } w \xrightarrow{*_n \mathsf{f}} r.$$

For $r = \texttt{current}(w, (?\,\%\, n))$, the rule is *write before last read*. Only the upper bound is needed and $i = n - 1 = \frac{\text{period}(w)}{\text{period}(r)} - 1$, giving $\texttt{p}_w \leq \text{period}(w) - \text{period}(r) + \texttt{p}_r$.

3.3 Resource constraints

The resource constraint encoding is relatively standard [33, §4.3]. For each equation eq with $period(eq) > 1$, we introduce a set of *phase weight* variables: $\texttt{pw}_{eq,i} \in \{0, 1\}$ indicates whether eq is scheduled in phase $0 \leq i < \text{period}(eq)$. Two constraints relate them to the corresponding phase variable:

$$\sum_{i=0}^{\text{period}(eq)-1} \texttt{pw}_{eq,i} = 1 \qquad\qquad \sum_{i=0}^{\text{period}(eq)-1} i \cdot \texttt{pw}_{eq,i} = \texttt{p}_{eq}.$$

For example, for $\text{period}(eq) = 3$, if $\texttt{p}_{eq} = 2$ then $\texttt{pw}_{eq,0} = 0$, $\texttt{pw}_{eq,1} = 0$, and $\texttt{pw}_{eq,2} = 1$.

For a declared resource res, let $w(res, eq)$ represent the amount of res required by the equation eq. For instantiations, $w(res, x^* = f(e^*))$ refers to the constant given in the declaration of f and defaults to 0. Otherwise $w(res, x = e) = 0$.

A constraint of the form "`resource balance` *res*" is encoded by introducing an integer variable $\texttt{rmax}_{res}$, and a constraint for each phase of the hypercycle $0 \leq k < hp$:

$$0 \leq \texttt{rmax}_{res} - \sum_{\{eq \,|\, \text{period}(eq)>1\}} w(res, eq) \cdot \texttt{pw}_{eq,(k \bmod \text{period}(eq))}.$$

The objective is then to minimize each $\texttt{rmax}_{res}$ variable, which means choosing equation phases that equalize as much as possible the sums of weights across all phases of the hypercycle. The $k \bmod \text{period}(eq)$ term accounts for the repetition of an equation across the hypercycle.

A source constraint of the form "`resource` $res \sim c$" is encoded by constraining the sum of resources in each phase $0 \leq k < hp$:

$$\sum_{\{eq \,|\, \text{period}(eq)>1\}} w(res, eq) \cdot \texttt{pw}_{eq,(k \bmod \text{period}(eq))} \sim c - \sum_{\{eq \,|\, \text{period}(eq)=1\}} w(res, eq).$$

The equations with $\text{period}(eq) = 1$ are included in every phase by subtracting their weights from the constant c.

3.4 Latency constraints

Latency refers to the number of cycles from an initial read forwards to eventual related writes, or from a final write backwards to earlier related reads. Compared to the ILP encodings of dependency and resource constraints, the treatment of latency constraints is less obvious. A constraint `latency` $m \ \sim \ b \ (eq_0, eq_1, \ldots, eq_{n-1})$, where $m \in \{\texttt{exists}, \texttt{forward}, \texttt{backward}\}$ and b is a constant bound, is valid only if the chain describes a path through the source program's flow graph, $eq_0 \xrightarrow{s_0, c_0} eq_1 \xrightarrow{s_1, c_1} \cdots \xrightarrow{s_{n-2}, c_{n-2}} eq_{n-1}$. A schedule associates phases to the equations in a chain and thereby induces forward and backward end-to-end latencies.

For the ROSACE example, the latency path is $d \xrightarrow{/_2 \mathsf{f}} h \xrightarrow{/_2 \mathsf{f}} a \xrightarrow{\mathsf{D}^{\mathsf{w}}_{\mathsf{f}}} v \xrightarrow{*_4 \mathsf{b}} e$, written here with truncated equation labels and traced in red in the flow graph of Figure 3c. The left half of Figure 4 plots the elements of this chain over a hypercycle, $hp = \mathsf{lcm}(2, 4, 8, 8, 2) = 8$, for the schedule of Figure 3d. There is a row for each equation: the downward pointing arrows indicate the equation's scheduled phase repeated across the hypercycle.

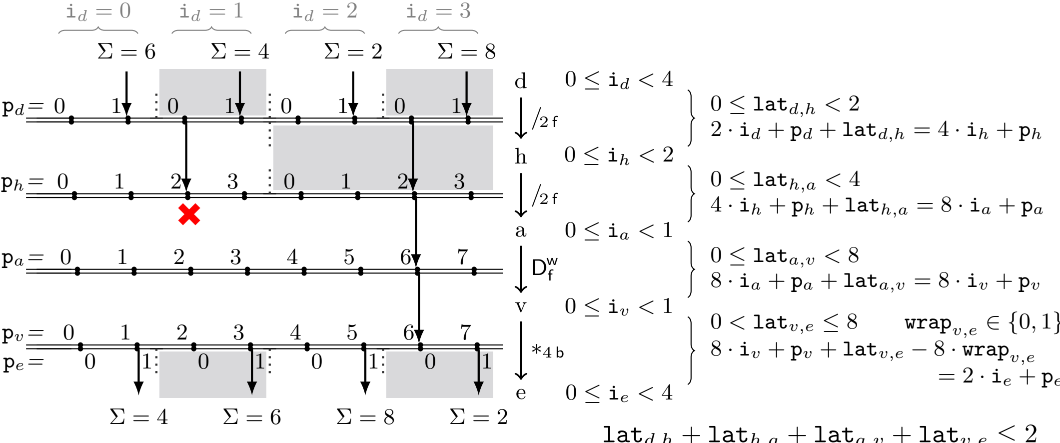


Figure 4 Unsimplified constraints for `latency exists <= 2 (d, h, a, v, e)`.

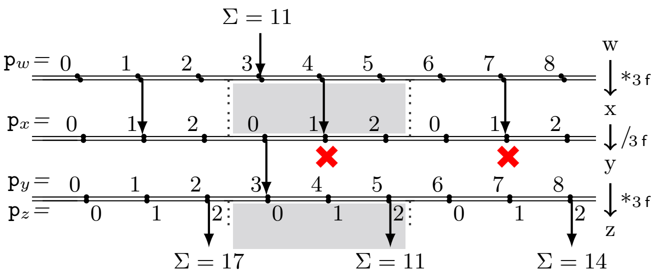


Figure 5 Latency chain: branching then selection.

The `forward` latencies, or *reaction times* (*first-to-first* [20, Figure 7]), are determined by working from the first equation in the chain through to the last. Consider the first of the four instances of d: the next closest instances are those of h after 1 cycle, the only instance of a after 4 cycles, that of v after 0 cycles, and finally the fourth instance of e after 1 cycle. The end-to-end latency is thus $1 + 4 + 0 + 1 = 6$. This is not the actual first-to-first path, which rather passes via the second instance of h, but it does not matter since, in terms of the sum of latencies, the paths commute. By always taking the next closest instance, we arrive at the correct corresponding last instance. The other forward end-to-end latencies are calculated similarly and shown at the top of the figure. Note that the fourth instance of d is sampled in a subsequent hypercycle and that the latency calculation thus "wraps around" through the first instance of h. The `backward` latencies, or *data ages* (*last-to-last* [20, Figure 7]), are determined by working from the last equation in the chain back to the first one. Consider the fourth instance of e: the closest preceding instance of v is 1 cycle before, those of a and h are immediate, and the third instance of d is 1 cycle before. The end-to-end latency is thus $1 + 0 + 0 + 1 = 2$. The backward end-to-end latencies are shown at the bottom of the figure. The `forward` and `backward` constraints require that all end-to-end latencies satisfy the bound, the `exists` constraint only requires that one backward latency does.

The goal now is to define an ILP encoding that characterizes end-to-end latencies in terms of phase variables. Bounding the resulting latency formulas constrains their phase variables and thus restricts the set of valid schedules.

3.4.1 Minimum pairwise latencies

A first idea is to calculate the end-to-end latency as the sum of minimum pairwise latencies. The minimum pairwise latency of a link with forward concomitance from w to r is

$$\mathtt{lat}_{w,r} = (\mathtt{p}_r - \mathtt{p}_w + \mathrm{period}(w)) \bmod \mathrm{period}(w) \quad \text{with } 0 \leq \mathtt{lat}_{w,r} < \mathrm{period}(w).$$

Adding period(w) avoids a negative modulo. For a link with backward concomitance, the minimum pairwise latency is period(w) when $\mathtt{p}_w = \mathtt{p}_r \bmod \mathrm{period}(w)$, giving

$$\mathtt{lat}_{w,r} = ((\mathtt{p}_r - \mathtt{p}_w + \mathrm{period}(w) - 1) \bmod \mathrm{period}(w)) + 1 \quad \text{with } 0 < \mathtt{lat}_{w,r} \leq \mathrm{period}(w).$$

Unfortunately this idea does not work for chains that contain both slow-to-fast links, which introduce branching, and fast-to-slow links, which filter branches. For example, consider the scheduled chain in Figure 5. The sequence of flow-graph arcs are shown at right. Taking the sum of minimum pairwise latencies incorrectly gives $1 + 2 + 2 = 5$. The correct forward latency, 11, is show at top, and the backward latencies, 11, 14, and 17, are shown at bottom. As explained by Feiertag et al. [20, Figure 7], the problem is that scheduling choices induce different propagation paths between a write to a chain's first variable and a corresponding read of its last one.

3.4.2 Closest instances

A better idea for encoding end-to-end latency is to characterize an arbitrary propagation path by introducing variables to identify the *instances* of each closest reader/writer relative to the hypercycle. Each such variable is constrained by the preceding write and subsequent read, and the sum of their pairwise latencies gives the end-to-end latency. We explain the idea on the ROSACE example before presenting the formal definition.

Returning to Figure 4, we introduce an instance variable for each equation in the chain. Since there are four instances of d that could participate in a path through the hypercycle, its instance variable is $0 \leq \mathtt{i}_d < 4$. Similarly, the instance variable for h is $0 \leq \mathtt{i}_h < 2$. We represent the latency between these two arbitrary instances by introducing a variable $0 \leq \mathtt{lat}_{d,h} < L$. The value of L is crucial. Since the example has an `exists` constraint, which applies to backward latencies, each reader instance must be associated to the immediately preceding writer instance by setting L to the period of the writer. Thus, here, $L = \mathrm{period}(d) = 2$. For forward latencies, each writer instance must be associated to the immediately succeeding reader instance by setting L to the period of the reader. A constraint is introduced to match writer and reader instances: $2 \cdot \mathtt{i}_d + \mathtt{p}_d + \mathtt{lat}_{d,h} = 4 \cdot \mathtt{i}_h + \mathtt{p}_h$. For each equation, we multiply the instance by the period and add the phase. The difference gives the pairwise latency. The right side of Figure 4 shows the result of applying this idea along the chain from d through to v. The instance variables $\mathtt{i}_a$ and $\mathtt{i}_v$ are not strictly necessary since they always equal zero. In the implementation, such variables are removed in a separate constant propagation pass.

For the $v \xrightarrow{*4\,\mathsf{b}} e$ link, the value of v may be read from the previous hypercycle. We permit such wraparounds by adding a variable $\mathtt{wrap}_{v,e} \in \{0, 1\}$ and subtracting $hp \cdot \mathtt{wrap}_{v,e}$ from the writer expression: $8 \cdot \mathtt{i}_v + \mathtt{p}_v + \mathtt{lat}_{v,e} - 8 \cdot \mathtt{wrap}_{v,e} = 2 \cdot \mathtt{i}_e + \mathtt{p}_e$. This encodes a modulo allowing instances to match within ($\mathtt{wrap}_{v,e} = 0$) and across ($\mathtt{wrap}_{v,e} = 1$) hypercycles. The strictnesses of the bounds $0 < \mathtt{lat}_{v,e} \leq 8$ account for the backward concomitance.

Finally, the sum of pairwise latencies is constrained: $\mathtt{lat}_{d,h} + \mathtt{lat}_{h,a} + \mathtt{lat}_{a,v} + \mathtt{lat}_{v,e} \leq 2$.

The scheme sketched above is implemented as a function from a list of flow-graph arcs to a set of ILP constraints. The function first calculates hp, the LCM of the periods in the chain's equations, and adds an instance variable for the initial writer $0 \leq \mathtt{i}_w < {}^{hp}/_{\mathrm{period}(w)}$. Then, for each arc $w \xrightarrow{s,c} r$, it adds three fresh variables, with their bounds,

$$0 \leq \mathtt{i}_r < {}^{hp}/_{\mathrm{period}(r)}$$
$$0 \leq \mathtt{lat}_{w,r} < L, \text{ for } c = \mathsf{f} \quad / \quad 0 < \mathtt{lat}_{w,r} \leq L, \text{ for } c = \mathsf{b}$$
$$\text{where } L = \mathrm{period}(r) \text{ if } \mathtt{forward} \text{ and } L = \mathrm{period}(w) \text{ if } \mathtt{backward},$$
$$0 \leq \mathtt{wrap}_{w,r} \leq 1, \text{ if } \mathtt{forward} \text{ and } s \notin \{\mathsf{D}^{\mathsf{w}}, *_n\} \text{ or if } \mathtt{backward} \text{ and } s \notin \{\mathsf{D}^{\mathsf{w}}, /_n\},$$

and an equality constraint,

$$\mathrm{period}(w) \cdot \mathtt{i}_w + \mathtt{p}_w + \mathtt{lat}_{w,r} - hp \cdot \mathtt{wrap}_{w,r} = \mathrm{period}(r) \cdot \mathtt{i}_r + \mathtt{p}_r.$$

A $\mathtt{wrap}_{w,r}$ term is not needed if, within a hypercycle, the dependency constraints guarantee that, if `forward`, a read follows the last write; or if `backward`, a write precedes the first read. The function compresses maximal sequences $w \xrightarrow{\mathsf{D}^{\mathsf{w}}_{\mathsf{f}}} \cdots \xrightarrow{\mathsf{D}^{\mathsf{w}}_{\mathsf{f}}} r$ to $w \xrightarrow{\mathsf{D}^{\mathsf{w}}_{\mathsf{f}}} r$. Finally, it adds the requested constraint on the sum of pairwise latencies: $\mathtt{lat}_{eq_0,eq_1} + \cdots + \mathtt{lat}_{eq_{n-2},eq_{n-1}} \sim b$.

For an `exists` constraint, the solver need only find a single propagation path that satisfies the constraints. For `forward` constraints, however, all propagation paths from the first equation in the chain must be considered. This is done by invoking the constraint generation function for each instance i of the first equation eq_0, and by anchoring the resulting, fresh first instance variable by an equality $\mathtt{i}_{eq_0} = i$. Similarly, for `backward` constraints, all propagation paths to the last equation in the chain must be considered. The function is invoked for each instance i of the last equation eq_{n-1}, and the resulting, fresh last instance variable is anchored by an equality $\mathtt{i}_{eq_{n-1}} = i$.

4 Code generation

The code generator takes as input (i) the original source program, (ii) a solution to the constraints from Section 3 that assigns a value to every $\mathtt{p}_{eq}$, and (iii) a parameter n_{s}, the number of step functions to generate, which must evenly divide the hyperperiod of all equations, that is, $n_{\mathsf{s}} \mid hp$. It recreates the flow graph (V, A) used to generate the constraints and then produces code comprising a static variable `c`, initialized to zero and incremented modulo hp at the end of every cycle, and n_{s} step functions, called over successive cycles in a repeating sequence. The code in Figure 3b shows the typical case where $n_{\mathsf{s}} = 1$.

A step function $step_i$ is generated for every $0 \leq i < n_{\mathsf{s}}$ and called in cycles where $\mathtt{c} \bmod n_{\mathsf{s}} = i$. The equations are filtered to determine which to include in $step_i$:

$$\mathsf{include}_i(eq) = \begin{cases} \mathtt{p}_{eq} \bmod n_{\mathsf{s}} = i & \text{if } n_{\mathsf{s}} \mid \mathrm{period}(eq) \\ i \bmod \mathrm{period}(eq) = \mathtt{p}_{eq} & \text{if } \mathrm{period}(eq) \mid n_{\mathsf{s}} \\ \mathit{true} & \text{otherwise} \end{cases}$$

Consider, for example, $n_{\mathsf{s}} = 4$: for $\mathrm{period}(a) = 8$ and $\mathtt{p}_a = 5$, the first case applies, and the equation a need only be included in $step_1$; for $\mathrm{period}(b) = 2$ and $\mathtt{p}_b = 1$, the second case applies, and b need only be included in $step_1$ and $step_3$; for $\mathrm{period}(c) = 3$, c must be included in all step functions regardless of its phase.

The equations within a $step_i$ are microscheduled according to their *instantaneous dependency graph* (V_i', A_i'), which is derived from the flow graph (V, A):

$$V_i' = \{w \mid w \in V \wedge \mathsf{include}_i(w)\} \quad A_i' = \{w \longrightarrow r \mid w \xrightarrow{s,\mathsf{f}} r \in A \wedge \mathsf{stogether}_i(w,r)\} \cup \{r \longrightarrow w \mid w \xrightarrow{s,\mathsf{b}} r \in A \wedge \mathsf{stogether}_i(w,r)\}.$$

That is, it only has vertexes for equations in $step_i$ and arcs for equations that may execute in the same instant; and arcs with backward concomitance become reversed dependencies. The predicate $\mathsf{stogether}_i(w, r)$ defines if a pair of communicating equations w and r sometimes execute together in $step_i$, recalling that either $\mathrm{period}(w) \mid \mathrm{period}(r)$ or $\mathrm{period}(r) \mid \mathrm{period}(w)$:

$$\mathsf{stogether}_i(w,r) = \mathsf{include}_i(w) \wedge \mathsf{include}_i(r) \wedge (\mathtt{p}_r \bmod \mathrm{period}(w) = \mathtt{p}_w \vee \mathtt{p}_w \bmod \mathrm{period}(r) = \mathtt{p}_r).$$

For example, if $\mathrm{period}(w) = 3$, $\mathtt{p}_w = 2$, $\mathrm{period}(r) = 6$, and $\mathtt{p}_r = 3$, then even if the two equations are in the same step function, they will never be executed together in the same cycle. However, if $\mathtt{p}_r = 5$, then r executes together with every second execution of w.

A standard algorithm [4, §5] is adapted to generate code for each $step_i$. It orders the equations according to (V_i', A_i') and translates each *eq* into a guarded assignment statement:

`switch (c %` $\mathrm{period}(eq)$`)` { `case` $\mathtt{p}_{eq}$`:` `TEq`(eq)`;`},

where `TEq` translates the equation. No guard is added if $\mathrm{period}(eq) \mid n_{\mathsf{s}}$, since it would always be true. As usual, a heuristic apposes equations with similar guards so that a subsequent *join* optimization can merge them to reduce branching. In our case, this means grouping equations with the same period where possible, and otherwise preferring equations with a greater harmonic period. The microscheduler is implemented so that equations are ordered by increasing period when the "fast-first" option is used.

5 Related work

5.1 Programming languages

Lustre [29] is a specification language with expressive activation conditions: arbitrary boolean expressions. These conditions are formalized in Lucid synchrone [7] as clock types, where a clock type is a sequence of variable names that abstract from the underlying boolean conditions. Various proposals have been made to restrict clock types to allow for specialized code generation. We present them in chronological order.

Periodic computations were specified in Signal [41] using *affine clocks* [55]. Abusing our notation, `base on` $(1 \% 4)$ is an affine clock since it is activated at instants $\{\frac{n}{4} + 1 \mid n \in \mathbb{N}_0\}$ relative to a base clock. The clock calculus is enriched with equational rules, for example, `base on` $1 \% 4$ `on` $0 \% 2 =$ `base on` $1 \% 8$. More programs can then be accepted, since two signals that do not have the same clock expression may still be composed synchronously.

The *n-synchronous model* [10] also uses clocks to specify periodic computations, this time as ultimately periodic binary words, like 00(10100). Such clocks are more expressive than affine periods but deciding equality is costly since they must be expanded to their LCM. Clock abstraction [11] solves this problem by considering envelopes (sets of clocks) characterized by a rational slope (the period) and an interval (possibly of length zero) for the initial phase. The slope of the `on` operation is the product of its argument's slopes as in our clock rules. This proposal was implemented, without code generation, in Lucy-n [43, 52].

Prelude is a multi-periodic synchronous language [21, 23, 24] whose clock types combine a rational phase and rational period. Following [7], the clock calculus is a type system that conservatively extends the ML-like clock calculus [14] of Lucid synchrone [53]. Prelude's periodic clocks correspond to clock envelopes [11], but the absence of subtyping gives a simpler, more efficient, but also more restrictive calculus. For example, the expression $f(x \texttt{ when } (1 \% 4)) + g(x \texttt{ when } (0 \% 4))$ is accepted by Lucy-n but rejected by Prelude because the clocks $1 \% 4$ and $0 \% 4$ have different phases and thus are not equal. Like Prelude, we adopt a "relaxed synchrony hypothesis" [15]: each stream may have its own notion of instant [21, §3.1.2]. The rules in Figure 2b are essentially the same as Prelude's periodic clock transformations [24, §3.3]. Prelude's semantics [21, §4.6] is defined using the tagged-signal model [42] with rational timestamps. In our setting, integer indexes suffice. While our components must execute within a cycle, those of Prelude execute as multiple tasks [50] with fixed [22] or dynamic [26] priorities. Compared to an earlier proposition [33], we discard phases in clock types, reduce the number of operators, and generate sequential code directly.

Other work treats the compilation of component-based languages into real-time tasks. Caspi et al. [8] propose a dynamic buffering protocol for multi-rate synchronous programs, and Giotto [31] introduces Logical Execution Time (LET) [37] to decouple execution and communication. Hamann et al. [30] describe AUTOSAR, where *runnables*, which correspond to our equations, are grouped into periodic tasks and communicate via *direct*, *implicit*, or LET conventions. The static ordering within a task determines which communications are "forward" or "backward" [30, §4.3]. Ignoring execution time and preemption allows us to propose a simpler model, constrain end-to-end latency, and generate sequential code.

The discrete-time subset of Simulink is a multi-periodic programming language. *Sample times* [45, §7] are period/phase pairs that are subject to rules like those for our clock types. Rate transition blocks play the same role as our `when` and `current` operators. When both data integrity and determinism are required, the writer and reader rates must be integer multiples of one another [44, §1-1482]. While Simulink models can be compiled to Lustre [56], Simulink Coder [46] is normally used. Code generation flattens a model, groups blocks by sample rate giving priority to faster ones, and produces one or more tasks for scheduling.

5.2 Real-time scheduling and end-to-end Latency

End-to-end latency continues to be studied in the context of real-time systems. Gerber et al. [27] and Davare et al. [16] apply constraint solving to assign task periods to ensure that latency bounds are met. Gerber et al. consider chains with direct and fast-to-slow links, for which sums of pairwise latencies suffice (Section 3.4.1). Davare et al. define a coarse upper bound: the sum of task periods and response times (§2.1: $l_p = \sum_{k \in p} t_k + r_k$), with response times considering preemptive scheduling for an ECU and non-preemptive scheduling for a CAN bus. The linearity of the bound permits its use in constraint-based scheduling. A tighter upper bound is possible when task priority decreases from producers to consumers [17, 39] since this reduces interference from preemption and thus worst-case response times. Task periods can be excluded from the sum if each task in a chain activates its successor. These "trigger" [49], "functional" [28], or "active" [17] chains, where activation and data dependencies coincide, are studied, for example, by Schlatow and Ernst [54] and Girault et al. [28]. In contrast, our constraint-based scheduling treats sample-based activation, assigns offsets not periods, requires slicing a source program into small tasks to avoid preemption, and uses an encoding that precisely characterizes end-to-end latencies (Section 3.4.2).

For the flight control system described in the introduction, and, doubtless, other systems, upper bounds on end-to-end latency can be too pessimistic. Doing better requires considering how chains are instantiated in a concrete schedule. Feiertag et al. [20, Figure 7] clearly describe the importance of propagation paths through communicating job instances and considering the "branching and filtering" effect of mixing slow-to-fast and fast-to-slow communications.

Which jobs actually communicate depends on the read/write discipline. It can be formalized using read and data intervals [3] or classified into *direct*, *read-execute-write*, or *LET* [30] communications. Direct access to shared variables complicates analysis and may cause inconsistencies. For read-execute-write, a task samples inputs when it begins and writes outputs when it ends. For LET, such atomic reads and writes occur at fixed times, isolating them from the effects of varying response times, and giving predictable but greater end-to-end latencies [47]. These issues do not arise in our source language and compilation scheme, but we do need to consider the timing of communications within a step function ("microscheduling"). To analyze non-harmonic communications, Hamann et al. [30, §5.2.2.3] insert virtual copy operations, much as we require programmers to insert explicit ones.

Calculating the exact worst-case end-to-end latency of a set of tasks, as opposed to an upper bound, requires exploring all data propagation paths in all possible schedules. Mohalik et al. [48] apply model-checking techniques. Lauer et al. [40] apply Mixed Integer Linear Programming (MILP) and use optimization to calculate the maximum end-to-end latency; our equality constraints (Section 3.4.2) resemble their constraints on production and consumption [40, §3.1, (2) & (5)] but we avoid real-valued time stamps, handle propagation paths across multiple hypercycles, and constrain latency during scheduling. Khatib et al. [36] propose an algorithm based on translating multi-periodic real-time task sets into Synchronous Dataflow (SDF) graphs. The algorithms of Kloda et al. [39, §IV.C] and Becker et al. [3, §V] iterate over a hypercycle through the jobs of the first task in a chain and recursively explore data propagation paths. In our approach, a solver combines (offline) scheduling with end-to-end latency calculation, allowing the latter to constrain the former. The solution of Becker et al. [3, §VI] is to prune data propagation paths that would exceed a given latency bound by adding job-level dependencies. This approach is implemented in a tool [2], which Klaus et al. [38] have incorporated into a compilation chain. They note that cyclic dependencies can be problematic [38, §6.2]. For Prelude, Wyss et al. [58] calculate worst-case latencies at the source level by defining propagation paths as *data dependency words*. Forget et al. [25] generalize this approach in a formal language for expressing end-to-end timing properties.

6 Conclusion

We present a language for expressing execution rates and rate transitions in the synchronous-reactive model. It is a special case of Lustre where programmers allow scheduling to reconcile resource requirements and end-to-end latencies. Our flow graphs facilitate the treatment of cycles and the interaction of two scheduling passes, one using an ILP solver to assign tasks to phases and the other ordering tasks within a phase. Our novel encoding of end-to-end latency constraints, allowing unconstricted rate transitions, improves on the current practice of manually scheduling critical sequences. Finally, we generalize a standard compilation scheme to produce sequential code.

Acknowledgements. We thank Matthieu Boitrel, Michel Angot, and Jean Souyris for their collaboration, and Guillaume Iooss for his earlier work. We are grateful to our RTSS 2022 reviewers for rightly rejecting an earlier, over-complicated latency encoding and for their encouraging comments.

References

1. Ardupilot website. URL: `https://ardupilot.org`.
2. Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. MECHAniSer - a timing analysis and synthesis tool for multi-rate effect chains with job-level dependencies. In *7th Int. Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2016)*, July 2016.
3. Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. Synthesizing job-level dependencies for automotive multi-rate effect chains. In *Proc. of the 22nd IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA 2016)*, pages 159–169. IEEE, August 2016.
4. Dariusz Biernacki, Jean-Louis Colaço, Gregoire Hamon, and Marc Pouzet. Clock-directed modular code generation for synchronous data-flow languages. In *Proc. of the 9th ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES 2008)*, pages 121–130. ACM Press, June 2008.
5. Pascal Brisset, Antoine Drouin, Michel Gorraz, Pierre-Selim Huard, and Jeremy Tyler. The Paparazzi solution. In *2nd US-European Competition and Workshop on Micro Air Vehicles (MAV 2006)*, October 2006.
6. Giorgio C. Buttazzo. *Predictable Scheduling Algorithms and Applications*. Real-Time Systems Series. Springer, 3ed. edition, 2011.
7. Paul Caspi and Marc Pouzet. Synchronous Kahn networks. In *Proc. of the 1996 ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'96)*, pages 226–238. ACM Press, May 1996.
8. Paul Caspi, Norman Scaife, Christos Sofronis, and Stavros Tripakis. Semantics-preserving multitask implementation of synchronous programs. *ACM Transactions on Embedded Computing Systems*, 7(2):15:1–15:40, January 2009.
9. Houssine Chetto, Maryline Silly, and Bouchentouf Toumi. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2:181–194, September 1990.
10. Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. N-synchronous Kahn networks: a relaxed model of synchrony for real-time systems. In *Proc. of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006)*, pages 180–193. ACM Press, January 2006.
11. Albert Cohen, Louis Mandel, Florence Plateau, and Marc Pouzet. Abstraction of clocks in synchronous data-flow systems. In *Proc. of the 6th Asian Symposium on Programming Languages and Systems (APLAS 2008)*, volume 5356 of *Lecture Notes in Computer Science*, pages 237–254. Springer, December 2008.
12. Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A conservative extension of synchronous data-flow with state machines. In *Proc. of the 5th ACM Int. Conf. on Embedded Software (EMSOFT 2005)*, pages 173–182. ACM Press, September 2005.
13. Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. Scade 6: A formal language for embedded critical software development. In *Proc. of the 11th Int. Symposium on Theoretical Aspects of Software Engineering (TASE 2017)*, pages 4–15. IEEE Computer Society, September 2017.
14. Jean-Louis Colaço and Marc Pouzet. Clocks as first class abstract types. In *Proc. of the 3rd Int. Workshop on Embedded Software (EMSOFT 2003)*, volume 2855 of *Lecture Notes in Computer Science*, pages 134–155. Springer, October 2003.
15. Adrian Curic. *Implementing Lustre Programs on Distributed Platforms with Real-Time Constraints*. PhD thesis, Université Joseph Fourier, September 2005.
16. Abhijit Davare, Qi Zhu, Marco Di Natale, Claudio Pinello, Sri Kanajan, and Alberto Sangiovanni-Vincentelli. Period optimization for hard real-time distributed automotive systems. In *44th Design Automation Conf.*, pages 278–283. ACM/IEEE, June 2008.
17. Marco Dürr, Georg von der Brüggen, Kuan-Hsun Chen, and Jian-Jia Chen. End-to-end timing analysis of sporadic cause-effect chains in distributed systems. *ACM Transactions on Embedded Computing Systems*, 18(5s):article no. 58, October 2019.

18 Peter Eades and Xuemin Lin. A heuristic for the feedback arc set problem. *Australasian Journal of Combinatorics*, 12:15–25, September 1995.

19 Peter Eades, Xuemin Lin, and W.F. Smyth. A fast & effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47(6):319–323, October 1993.

20 Nico Feiertag, Kai Richter, Johan Nordlander, and Jan Jonsson. A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. In *Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS 2008, co-located with RTSS 2008)*, November 2008.

21 Julien Forget. *Un Langage Synchrone pour les Systèmes Embarqués Critiques Soumis à des Contraintes Temps Réel Multiples.* PhD thesis, Université de Toulouse, November 2009.

22 Julien Forget, Frédéric Boniol, Emmanuel Grolleau, David Lesens, and Claire Pagetti. Scheduling dependent periodic tasks without synchronization mechanisms. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2010)*, pages 301–310. IEEE, April 2010.

23 Julien Forget, Frédéric Boniol, David Lesens, and Claire Pagetti. A multi-periodic synchronous data-flow language. In *Proc. of the 11th IEEE High Assurance Systems Engineering Symposium (HASE 2008)*, pages 251–260. IEEE, December 2008.

24 Julien Forget, Frédéric Boniol, David Lesens, and Claire Pagetti. A real-time architecture design language for multi-rate embedded control systems. In *Proc. of the 25th ACM Symposium on Applied Computing (SAC'10)*, pages 527–534. ACM, March 2010.

25 Julien Forget, Frédéric Boniol, and Claire Pagetti. Verifying end-to-end real-time constraints on multi-periodic models. In *Proc. of the IEEE 22nd Int. Conf. on Emerging Technologies & Factory Automation (ETFA 2017).* IEEE Press, September 2017.

26 Julien Forget, Emmanuel Grolleau, Claire Pagetti, and Pascal Richard. Dynamic priority scheduling of periodic tasks with extended precedences. In *Proc. of the IEEE 16th Int. Conf. on Emerging Technologies & Factory Automation (ETFA 2011).* IEEE Press, September 2011.

27 Richard Gerber, Seongsoo Hong, and Manas Saksena. Guaranteeing end-to-end timing constraints by calibrating intermediate processes. *IEEE Transactions on Software Engineering*, 21(7):579–592, July 1995.

28 Alain Girault, Christophe Prévot, Sophie Quinton, Rafik Henia, and Nicolas Sordon. Improving and estimating the precision of bounds on the worst-case latency of task chains. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2578–2589, November 2018.

29 Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language LUSTRE. *Proc. of the IEEE*, 79(9):1305–1320, September 1991.

30 Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. Communication centric design in complex automotive embedded systems. In *29th Euromicro Conf. on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz Int. Proc. in Informatics (LIPIcs)*, pages 10:1–10:20. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, June 2017.

31 Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proc. of the IEEE*, 91(1):84–99, January 2003.

32 IBM Corp. *IBM ILOG CPLEX 20.1 User's Manual*, 2020.

33 Guillaume Iooss, Albert Cohen, Dumitru Potop-Butucaru, Marc Pouzet, Vincent Bregeon, Jean Souyris, and Philippe Baufreton. Polyhedral scheduling and relaxation of synchronous reactive systems. In *12th Int. Workshop on Polyhedral Compilation Techniques (IMPACT 2022).* HiPEAC, June 2022.

34 Gilles Kahn. The semantics of a simple language for parallel programming. In *Proc. of the Int. Federation for Information Processing (IFIP) Congress 1974*, pages 471–475. North-Holland, August 1974.

35 Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, The IBM Research Symposia Series (IRSS), pages 85–103. Springer, 1972.

36 Jad Khatib, Alix Munier-Kordon, Enagnon Cedric Klikpo, and Kods Trabelsi-Colibet. Computing latency of a real-time system modeled by Synchronous Dataflow Graph. In *Proc. of the 24th Int. Conf. on Real-Time Networks and Systems (RTNS'16)*, pages 87–96. ACM Press, October 2016.

37 Christoph M. Kirsch and Ana Sokolova. The Logical Execution Time paradigm. In *Advances in Real-Time Systems*, pages 103–120. Springer, 2012.

38 Tobias Klaus, Florian Franzmann, Matthias Becker, and Peter Ulbrich. Data propagation delay constraints in multi-rate systems — deadlines vs. job-level dependencies. In *Proc. of the 26th Int. Conf. on Real-Time Networks and Systems (RTNS'18)*, pages 93–103. ACM Press, October 2018.

39 Tomasz Kloda, Antoine Bertout, and Yves Sorel. Latency analysis for data chains of real-time periodic tasks. In *Proc. of the IEEE 23rd Int. Conf. on Emerging Technologies & Factory Automation (ETFA 2018)*, pages 360–367. IEEE Press, September 2018.

40 Michaël Lauer, Frédéric Boniol, Claire Pagetti, and Jérôme Ermont. End-to-end latency and temporal consistency analysis in networked real-time systems. *Int. Journal of Critical Computer-Based Systems*, 5(3/4):172–196, 2014.

41 Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with SIGNAL. *Proc. of the IEEE*, 79(9):1321–1336, September 1991.

42 Edward A. Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on CAD*, 17(12):1217–1229, December 1998.

43 Louis Mandel, Florence Plateau, and Marc Pouzet. Lucy-n: a n-Synchronous extension of Lustre. In *Proc. of the 10th Int. Conf. on Mathematics of Program Construction (MPC 2010)*, volume 6120 of *Lecture Notes in Computer Science*, pages 288–309. Springer, June 2010.

44 The MathWorks. *Simulink® Reference*, March 2022. Release 2022a.

45 The MathWorks. *Simulink: User's Guide*, March 2022. Release 2022a.

46 The MathWorks. *Simulink® Coder™ User's Guide*, March 2022. Release 2022a.

47 Slobodan Matic and Thomas A. Henzinger. Trading end-to-end latency for composability. In *Proc. of the 26th IEEE Real-Time Systems Symposium (RTSS 2005)*, pages 110–119. IEEE Computer Society, December 2005.

48 Swarup Mohalik, Devesh B. Chokshi, Manoj G. Dixit, A.C. Rajeev, and S. Ramesh. Scalable model-checking for precise end-to-end latency computation. In *Proc. of the 2013 IEEE Conf. on Computer Aided Control System Design (CACSD)*, pages 19–24. IEEE, August 2013.

49 Saad Mubeen, Jukka Mäki-Turja, and Mikael Sjödin. Support for end-to-end response-time and delay analysis in the industrial tool suite: Issues, experiences and a case study. *Computer Science and Information Systems*, 10(1):453–482, January 2013.

50 Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems*, 21(3):307–338, September 2011.

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

52 Florence Plateau. *Modèle n-synchrone pour la programmation de réseaux de Kahn à mémoire borné*. PhD thesis, Université Paris XI, January 2010.

53 Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual.* Université Paris-Sud, April 2006.

54 Johannes Schlatow and Rolf Ernst. Response-time analysis for task chains in communicating threads. In *22nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2016)*. IEEE, April 2016.

55 Irina Mădălina Smarandache, Thierry Gautier, and Paul Le Guernic. Validation of mixed Signal-Alpha real-time systems through affine calculus on clock synchronisation constraints. In *Proc. of the World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, volume 1709 of *Lecture Notes in Computer Science*, pages 1364–1383. Springer, September 1999.

56 Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. Translating discrete-time Simulink to Lustre. *ACM Transactions on Embedded Computing Systems*, 4(4):779–818, November 2005.

57 Rémy Wyss, Frédéric Boniol, Julien Forget, and Claire Pagetti. A synchronous language with partial delay specification for real-time systems programming. In *Proc. of the 10th Asian Symposium on Programming Languages and Systems (APLAS 2012)*, volume 7705 of *Lecture Notes in Computer Science*, pages 223–238. Springer, December 2012.

58 Rémy Wyss, Frédéric Boniol, Julien Forget, and Claire Pagetti. End-to-end latency computation in a multi-periodic design. In *Proc. of the 28th ACM Symposium on Applied Computing (SAC'13)*, pages 1682–1687. ACM, March 2013.

Towards Efficient Explainability of Schedulability Properties in Real-Time Systems

Sanjoy Baruah
Washington University in Saint Louis, MO, USA

Pontus Ekberg
Uppsala University, Sweden

Abstract

The notion of **efficient explainability** was recently introduced in the context of hard-real-time scheduling: a claim that a real-time system is schedulable (i.e., that it will always meet all deadlines during run-time) is defined to be efficiently explainable if there is a proof of such schedulability that can be verified by a polynomial-time algorithm. We further explore this notion by (i) classifying a variety of common schedulability analysis problems according to whether they are efficiently explainable or not; and (ii) developing strategies for dealing with those determined to not be efficiently schedulable, primarily by identifying practically meaningful sub-problems that are efficiently explainable.

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

Keywords and phrases Recurrent Task Systems, Uniprocessor and Multiprocessor Schedulability, Verification, Explanation, Computational Complexity, Approximation Schemes

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.2

Funding *Sanjoy Baruah*: US National Science Foundation (Grants CNS-2141256 and CNS-2229290).
Pontus Ekberg: Swedish Research Council grant 2018-04446.

1 Introduction

A workshop titled Explainability of Real-Time Systems and their Analysis (ERSA) was held as part of the 2022 edition of the IEEE Real-Time Systems Symposium (RTSS), with a goal "to understand the role, meaning, and value of explanation in critical systems – in particular real-time systems." In a paper [8] that we had presented at this workshop, we introduced the notion of **efficient explainability** in the context of hard-real-time scheduling. In this we drew inspiration from the remarkable success that cyclic-executive (CE) [4, 5] based approaches to demonstrating timing correctness in safety-critical systems have enjoyed, particularly with regard to achieving statutory certification. In such CE based approaches, the system developer provides the certification authority (CA) with a lookup table that explicitly enumerates which task will execute at each instant; the CA checks that repeated execution of this lookup table assigns adequate computing to each task to allow all its timing constraints to be met (provided, of course, that each task respects its worst-case execution time bound). From this perspective, the lookup table may be thought of as a *certificate* that "explains" the system-developer's claim that the system is schedulable. We accordingly defined a claim of schedulability to be efficiently explainable if there is a certificate of the schedulability that can be verified in time polynomial in the representation of the system for which schedulability is being claimed. We applied this notion to the schedulability analysis of independent sporadic task systems upon preemptive uniprocessors, and made some simple observations that may nevertheless be surprising (e.g., that for uniprocessor scheduling of

© Sanjoy Baruah and Pontus Ekberg;
licensed under Creative Commons License CC-BY 4.0

constrained-deadline sporadic task systems [27, 30], fixed-priority schedulability is efficiently explainable but, under the widely-held assumption that $\mathtt{NP} \neq \mathtt{coNP}$, EDF-schedulability is not), and posed some open questions and possible directions for further research.

This Work: Motivation and Scope. We believe that the concept of efficient explainability potentially has an important role to play in the verification of future safety-critical real-time systems as such systems become ever more complex, and as they are increasingly coming to be implemented upon resource-constrained platforms:

1. CAs may be unwilling to undertake inordinately long computational procedures in order to verify the correctness of systems that are submitted for certification. This becomes more challenging for modern systems given the increased complexity of these systems and the correctness properties that they are expected to maintain in the ever-increasingly complex environments within which they operate.
2. An additional motivation for efficient explainability comes from the increasing application of the paradigm of edge computing in many safety-critical applications like autonomous navigation. Due to size, weight and power (SWaP) constraints, devices on the edge typically have limited computational capabilities; hence it is beyond their ken to undertake complex computations that are computationally demanding. They may nevertheless need to perform complex run-time operations such as admission control. One approach to this would be to perform the actual admission control computation "in the cloud" where more extensive computational capabilities are available. If such cloud computations determine that the workload remains schedulable upon admitting the additional work, then the specifications of the schedulable system including the newly-admitted work, along with an efficiently-verifiable proof (the "efficient explanation") of the schedulability, are communicated to the computationally limited edge device. The edge device can then verify the schedulability and admit the new work, *without* needing to trust the cloud computation.

These motivating considerations have prompted us to study the concept of efficient explainability further, with a view of better understanding its applicability to safety-critical system design, implementation, and verification. In this paper we report on our investigations into two **aspects** of efficient explainability:

1. We characterize several schedulability-analysis problems beyond those considered in [8] as being efficiently explainable or not.
2. If some schedulability analysis problem is unlikely to allow efficiently-verifiable certificates for all instances, there may be sub-problems of it that do. We investigate this idea in depth, proposing several avenues to identifying efficiently explainable sub-problems, as well as alternative notions of efficient explainability.

The specific **contributions** of our work include the following.

1. We extend the results of [8] to *multiprocessors* by characterizing the efficient explainability (or absence thereof) of multiprocessor partitioned schedulability of recurrent task systems.
2. We obtain a series of results that identify *efficiently explainable sub-problems of uniprocessor EDF-schedulability* of sporadic task systems (a problem that was observed in [8] to not be efficiently explainable). We also extend these results to *partitioned EDF scheduling upon multiprocessor platforms.*

3. We propose a novel concept, *Fully Polynomial-Time* ***Verification*** *Approximation Scheme* (FPTVAS), that extends the notion of FPTAS's, widely studied in the context of approximation algorithms [3, 34], from polynomial-time computation to polynomial-time verification (i.e., from `P` to `NP`).
4. We introduce, and initiate a study of, a previously unconsidered complexity class – `pseudoNP`*: non-deterministic pseudo-polynomial time* that helps generalize the concept of efficient explainability. This generalization seems particularly relevant as the schedulability problems for which efficient explanations are sought become computationally more challenging.

Organization. The remainder of this manuscript is organized as follows. In Section 2 we place this work within the larger context of verification of real-time systems, and briefly review some concepts from real-time scheduling and complexity theory that will be used in later sections of this document. Sections 3–6 contain our main technical contributions: after briefly identifying some efficiently explainable multiprocessor schedulability analysis problems in Section 3, we turn our attention, in Sections 4–6, to identifying efficiently explainable sub-problems of uni- and multi-processor schedulability analysis problems that are unlikely to be efficiently explainable. We conclude in Section 7 by reiterating the significance of the research described in this paper, and proposing some directions for further research.

2 Context, and Background & Related Work

The web page[1] for the ERSA workshop observes that "many software-intensive systems of current and future application domains require (or will require) approval from a certification authority." It highlights limitations of current approaches to obtaining such approval, particularly when applied to advanced application domains, and states, as motivation for the workshop, that "it is worth exploring [$\cdots$] whether explainability can help." We would like to emphasize that there are a myriad of aspects to explainability as it pertains to such a use-case. Let us consider, as an illustrative toy example, an effort at convincing a certification authority (CA) of the correctness of the timing behavior of a particular system by appealing to the well-known result in Liu and Layland's seminal paper [27] that *a periodic task system with utilization* $\leq \ln 2$ *will meet all its deadlines when scheduled using rate-monotonic scheduling* (or in the terminology of explainability, that *a periodic task system's utilization being* $\leq \ln 2$ *constitutes an 'explanation' of its schedulability.*)

1. The party seeking certification must demonstrate that the periodic task model proposed in [27] adequately models the salient characteristics of the workload, and that the preemptive uniprocessor model assumed in [27] adequately models the salient characteristics of the implementation platform.
2. They must provide justification for the values they have assigned to the task WCET parameters (by, for instance, showing that the values were obtained using tools [35] that have been certified for this purpose), as well as the values assigned to the task period parameters.[2]

[1] https://sites.google.com/view/ersa22
[2] Examples of WCET analysis tools are aiT (https://www.absint.com/ait/), Heptane (https://team.inria.fr/pacap/software/heptane/) and OTAWA (http://www.tracesgroup.net/otawa/).

3. The CA must accept the validity of the analysis presented in [27], that any rate-monotonic scheduled system with utilization $\leq \ln 2$ will meet all deadlines.[3]
4. Finally, the party seeking certification must demonstrate to the CA that the system utilization – i.e., the sum, over all the tasks in the system, of the ratio of their WCET parameters to their period parameters – does not exceed $\ln 2$. (Recall that this constitutes the 'explanation' of the system's schedulability, which the CA will presumably seek to verify on their own – i.e., they will not simply take the word of the party seeking certification that it is so.)

While each of these steps is crucial to achieving certification, *the focus of this paper is on the last step.* (Although this step is quite trivial for our toy example above, we will see that it is not always so.) Hence the work described in this paper should be looked upon as constituting only one part of a certification process that necessarily includes a social dimension (Step 1, accepting the mapping from an actual system to an abstract model, inherently possesses a social aspect in addition to technical ones, as do parts of Step 2); tool-development and certification (Step 2); and formal verification of schedulability results (Step 3). Our long-term vision for the non-social aspects of certification is that an automatically-verifiable proof of system correctness in some formalism such as Coq [11] or Prosa [15] will be provided to the CA. This will incorporate correctness proofs of the schedulability results (see, e.g., [13, 10] for some examples of such proofs) upon which such correctness depends, and additionally include a certificate that explains why these schedulability results imply the correctness of the specific system being considered for certification (see [29] for a seminal work on creating such certificates in Prosa). It is this last part – explaining why the formally-verified general schedulability results imply correctness for the particular system – that we address: we want to ensure that the certificate of correctness can be verified *efficiently* by the CA. In this work we outline and study theoretical underpinnings to understand which schedulability problems allow for efficient verifiability (or *explainability*, in the terminology adopted in this paper), and how to deal with schedulability problems that are unlikely to allow this for all instances.

Efficient Explainability. A schedulability problem is here said to be efficiently explainable if and only if for all schedulable task systems there is a certificate of this schedulability that can be verified by an algorithm that has running time polynomial in the size of the representation of the task system. This definition directly links efficient explainability to well-studied concepts in computational complexity theory [31, 2]; in particular, the complexity class NP: "NP is the class of languages that can be verified by a polynomial-time algorithm" [16, page 1064]. Hence, a schedulability problem is efficiently explainable if it belongs to NP, and showing that a schedulability-analysis problem is unlikely to be in NP offers strong evidence that it is not efficiently explainable. In Sections 5 and 6 we will also consider wider notions of efficient explainability than simply equating it with NP.

How does one show that a schedulability-analysis problem is unlikely to be in NP? Here one again makes use of well-established results from computational complexity theory: there are several complexity classes (see Figure 1 for some) that are widely believed to be distinct from NP in the sense that there are problems within these complexity classes that do not also belong in NP. Recall that a problem is defined to be *hard* for a complexity class if it

---

[3] We point out that this step is by no means trivial or automatic – examples abound of results that passed peer-review and were published in technical forums, only to subsequently be discovered to be erroneous. Hence it is understandable that a CA be sceptical of published results and seek further justification of their correctness than merely having passed peer review.

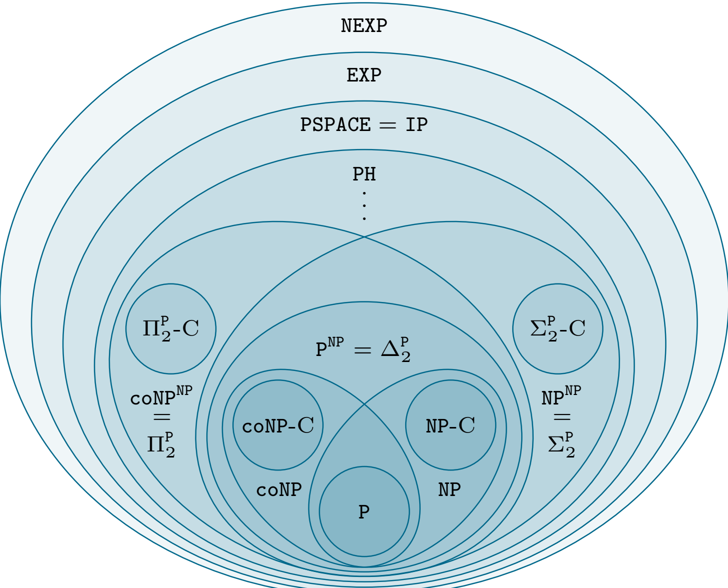


Figure 1 Complexity classes considered in this manuscript. Complexity theory researchers widely believe that no region in this diagram is empty – each is populated with problems.

is, intuitively speaking, at least as computationally difficult to solve as every other problem in that complexity class (or more precisely, every problem in the complexity class can be reduced, in polynomial time, to this hard problem). Hence showing a schedulability-analysis problem to be hard for any complexity class believed to be distinct from NP offers strong evidence that it cannot also be in NP and is therefore not efficiently explainable.

Efficient Explainability: Prior Results. The following observations and results were obtained in [8] by exploiting this equivalence between efficient explainability and membership in NP:

- It is efficiently explainable whether or not a constrained-deadline synchronous periodic or sporadic[4] task system is FP-schedulable upon a preemptive uniprocessor.
- In contrast, determining whether a constrained (or arbitrary) deadline synchronous periodic or sporadic task system is EDF-schedulable or not upon a preemptive uniprocessor platform is unlikely to be efficiently explainable.
- Suppose one were given an FP-scheduled constrained-deadline synchronous periodic task system in which each task is additionally characterized by a lower bound (a 'best-case execution time') on its execution duration, and a lower bound is specified on the response time – i.e., the duration between a job's arrival and its completion – that is acceptable for each task. Determining whether such a system will be scheduled to respect the specified response time lower bounds is not likely to be efficiently explainable.

[4] Recall that for a *periodic* task the period parameter denotes the exact duration between successive job arrivals, whereas for a *sporadic* task it denotes the minimum duration between successive job arrivals. In a *synchronous* periodic task system, the first task of each job arrives at time-instant zero.
In *constrained*-deadline task systems, the relative deadline parameter of each task is $\leq$ the task's period parameter; no such relationship need exist in *arbitrary*-deadline task systems. (*Implicit*-deadline task systems have each task's deadline parameter equal to its period parameter.)

FPTAS's. As stated in Section 1, one of our goals is to identify efficiently explainable sub-problems of schedulability-analysis problems that are determined to not be efficiently explainable. We will see that this goal essentially translates to one of obtaining sufficient (rather than exact) schedulability tests. *Speedup factors* [24, 32, 25] are a commonly used quantitative metric of the effectiveness of sufficient schedulability tests. The speedup factor of a sufficient schedulability test $\mathcal{A}$ is defined to be the smallest real number $\delta \geq 0$ such that if any task system Γ is schedulable upon a unit-speed processor, then $\mathcal{A}$ will determine that Γ is schedulable upon a speed-$(1+\delta)$ processor. Smaller speedup factors denote 'better' (i.e., closer to optimal in the worst case) sufficient tests. Thus, obtaining a good sufficient schedulability test may be thought of as obtaining a good approximation algorithm that minimizes the speedup factor. In the theory of approximation algorithms [3, 34], it is widely accepted that an FPTAS (see, e.g., [16, p. 1107] for a textbook description) is the 'best' kind of approximation algorithm: it allows for approximations that are arbitrarily close to the optimal by appropriately assigning a value to a parameter δ. In the context of sufficient schedulability tests, an FPTAS may be defined as follows:[5]

▶ **Definition 1** (FPTAS). *A fully polynomial-time approximation scheme (FPTAS) for a schedulability analysis problem is an algorithm that, given as input any problem instance* Γ *and a parameter* $\delta > 0$*, returns "unschedulable" if* Γ *is unschedulable on a speed-1 processor, and returns "schedulable" if* Γ *is schedulable on a speed-*$(1/(1+\delta))$ *processor. Its running time is bounded by a polynomial in the two parameters* $|\Gamma|$ *and* $\left(\frac{1}{\delta}\right)$.

2.1 Some relevant results from real-time scheduling theory

Fixed-Priority scheduling. *Response-time analysis (RTA)* [23, 26] is the standard technique for determining whether a constrained-deadline synchronous periodic task system is schedulable or not under fixed-priority (FP) scheduling. RTA is based on the observation [23] that if a constrained-deadline task system is schedulable under FP, then the maximum possible duration between the release of a job of τ_i and the instant this job completes execution (called the *worst-case response time* of task τ_i) is equal to the smallest positive value of R_i that satisfies the following recurrence (here $\text{hp}(\tau_i)$ denotes all jobs in the task system that have scheduling priority greater than τ_i's scheduling priority):

$$R_i = C_i + \sum_{\tau_j \in \text{hp}(\tau_i)} \left\lceil \frac{R_i}{T_j} \right\rceil \times C_j \tag{1}$$

As observed in [8], it is well known that FP-schedulability is in `NP` and must therefore be efficiently explainable: the certificate for FP-schedulability for a given task system Γ is a value for R_i for each $\tau_i \in \Gamma$ that satisfies Expression 1 and is $\leq D_i$'s. Such a certificate comprises $|\Gamma|$ numbers, and so is polynomial (in fact linear) in the representation of the task system Γ. It is straightforward to observe that each claimed R_i can be verified to be a solution to Equation 1 in linear time.

---

[5] It should be noted here that we are abusing the notion of an FPTAS slightly, to be consistent with prior work in real-time scheduling. In Definition 1 we are assuming that processor speed is the quantity to be approximated, even though the usual definition of FPTAS's allow any other approximation metric. Other metrics relevant for scheduling are, for example, makespan or maximum tardiness. We are also not quite treating the FPTAS in Definition 1 as an approximation algorithm (since the algorithm should output only "schedulable" or "unschedulable"), but without much further work we can use binary search to turn such an algorithm into an algorithm for approximating the minimum processor speed needed to schedule the task system.

Earliest-Deadline-First scheduling. In earliest-deadline-first (EDF) scheduling, at each instance the currently active (i.e., needing execution) job with the earliest deadline is executed. It is unlikely that EDF-schedulability is efficiently explainable since it has been shown [18, 20, 19] to be coNP-hard. If it was possible to create polynomial-time verifiable certificates for *all* EDF-schedulable task systems, attesting their EDF-schedulability, then by definition the EDF-schedulability problem would be in NP. This would immediately imply that NP = coNP, which goes against the expectations of most researchers in complexity theory. It is interesting to note that even though uniprocessor FP-schedulability (which is NP-complete [22]) and uniprocessor EDF-schedulability (which is coNP-complete [18]) in some sense are qualitatively equally hard to solve (they are both complete at the first level of the polynomial hierarchy), their verification problems are indeed of very different hardness.

Processor-demand analysis (PDA) [7] is the standard technique for determining whether a synchronous periodic task system is schedulable or not under EDF scheduling. PDA is centered upon the concept of the *demand bound function* (DBF): for any sporadic task τ_i and any interval-duration $t \geq 0$, $\text{DBF}_i(t)$ denotes the maximum possible cumulative execution requirement by jobs of task τ_i that both arrive in, and have deadlines within, any interval of duration t. The following formula for computing $\text{DBF}_i(t)$ was derived in [7]:

$$\text{DBF}_i(t) = \max\left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1, 0\right) \times C_i \tag{2}$$

It was also shown in [7] that a necessary and sufficient condition for task system Γ to be EDF-schedulable is that the following condition holds for all $t \geq 0$:

$$\sum_{\tau_i \in \Gamma} \text{DBF}_i(t) \leq t \tag{3}$$

Processor Demand Analysis, PDA, is essentially a means of determining whether Expression 3 holds for all t. It was proved in [7] that Condition 3 need only be checked for values of t that are of the form $t = (k \times T_i + D_i)$ for some non-negative integer k and some $\tau_i \in \Gamma$; furthermore, only such values that are no larger than the hyper-period $\text{HP}(\Gamma)$ (the least common multiple of all the T_i parameters) need be tested. The set of all such values of t for which it needs to be checked that Condition 3 is satisfied in order to verify EDF-schedulability is called the *testing set* for the sporadic task system Γ and often denoted $\mathcal{T}(\Gamma)$. It is known [7] that the cardinality $|\mathcal{T}(\Gamma)|$ of the testing set $\mathcal{T}(\Gamma)$ may in general be exponential in the representation of Γ. However, it has been shown [6, Theorem 3.1] that a smaller testing set, of pseudo-polynomial cardinality (i.e., polynomial in the size of the representation of Γ and its largest numerical parameter), can be identified for *bounded-utilization* task systems. Those are the task systems Γ satisfying the additional condition that $\sum_{\tau_i \in \Gamma} C_i/T_i \leq c$ for some pre-defined constant $c < 1$.

3 Efficiently Explainable Schedulability

We start out identifying some important efficiently explainable schedulability-analysis problems. As mentioned in Section 2.1 above, fixed-priority (FP) schedulability of constrained-deadline sporadic task systems upon preemptive uniprocessor platforms was observed to be efficiently explainable in [8]; this result is easily generalized to show that FP schedulability of constrained-deadline sporadic task systems upon *multi*processor platforms under the partitioned paradigm[6] is also efficiently explainable:

[6] Throughout this paper, when discussing partitioned scheduling, we refer to strict temporal partitioning where tasks on one partition execute independently of tasks on another partition. Semi-partitioned approaches or cases where tasks share locks across partitions may require additional care for verifiability.

▶ **Theorem 2.** *Partitioned multiprocessor fixed-priority schedulability of constrained-deadline sporadic task systems is efficiently explainable.*

Proof. In Section 2.1 we have briefly described the procedure for constructing efficiently-verifiable certificates of uniprocessor FP-schedulability of constrained deadline task systems. This procedure is easily generalized to establish the efficient explainability of partitioned FP-schedulability for constrained-deadline systems upon multiprocessors as well: an efficiently-verifiable certificate of the partitioned fixed-priority schedulability for a constrained-deadline sporadic task system Γ upon an m-processor platform comprises

1. The actual partitioning of Γ into the m sets of tasks assigned to the m processors; and
2. For each of the m partitions, a certificate of its uniprocessor fixed-priority schedulability that is constructed using the procedure that we had described in Section 2.1.

It is evident that such a certificate can be verified by a polynomial-time algorithm, and hence establishes that FP-schedulability under the partitioned paradigm of multiprocessor scheduling is efficiently explainable.

We point out that nothing in this proof requires that the processors be identical to one another; hence the result of this theorem holds for the more general heterogeneous multiprocessor platforms. ◀

As stated in Section 2.1, preemptive uniprocessor EDF-schedulability analysis for sporadic task systems is not likely to be efficiently explainable, in contrast to FP scheduling. Since partitioned multiprocessor EDF scheduling is a generalization of uniprocessor EDF scheduling, it therefore follows that partitioned multiprocessor EDF-schedulability analysis problem is also unlikely to be efficiently explainable. However, for the special case of implicit-deadline task systems (for which the corresponding uniprocessor problem is efficiently explainable – simply determine whether system utilization does not exceed the processor capacity), efficient explainability is easily established:

▶ **Theorem 3.** *Partitioned multiprocessor EDF schedulability of implicit-deadline sporadic task systems is efficiently explainable.*

Proof. An efficiently-verifiable certificate of the partitioned EDF schedulability for an implicit-deadline sporadic task system Γ upon an m-processor platform comprises the actual partitioning of Γ into the m sets of tasks assigned to the m processors. Given such a certificate, it can be verified in polynomial time that the sum of the utilizations of the tasks assigned to each processor does not exceed the capacity of that processor. ◀

Perhaps somewhat surprisingly, most other common real-time schedulability analysis problems are unlikely to have polynomial-time verifiable certificates for explaining schedulability; this motivates our efforts, reported in Sections 4–6, to investigate other avenues to dealing with such problems.

4 Identifying Efficiently Explainable Sub-Problems

In this section we consider one approach to achieving efficient explainability of problems that are *not* in `NP` (and hence not efficiently explainable): to identify relevant sub-problems that are in `NP`. We exemplify this approach with the EDF-schedulability problem. As stated in Section 2.1, the uniprocessor EDF-schedulability problem of three-parameter[7] sporadic

[7] That is, a task specified by a triple of its task parameters: $\tau_i = (C_i, D_i, T_i)$.

(or periodic) tasks is coNP-complete, from which it follows that it cannot be in NP unless NP = coNP and the polynomial hierarchy collapses to its first level. Barring the unlikely event of such a collapse, we cannot produce polynomially-sized explanations of EDF schedulability for all schedulable task systems that can be verified by a third party in polynomial time.

We set out to identify sub-problems of the uniprocessor EDF-schedulability problem for sporadic (or synchronous periodic) task systems that are in NP, and therefore efficiently explainable. We do so by identifying subsets of the language of EDF-schedulable task sets that are in NP (or indeed, even in P). Since the union of a finite set of languages in NP is itself a language in NP, we can try to cover as much as possible of the EDF-schedulable task sets with different subsets in NP, and in the end take the union of any such subsets. Any EDF-schedulable task set that is in this union of smaller languages *does* have a polynomially-sized certificate that can be verified by a third party in polynomial time.

Before proceeding we should make some important points.

- It is not possible to completely cover the coNP-complete language of EDF-schedulability by a finite set of subset languages that are each in NP (assuming NP ≠ coNP) – there will remain an infinite set of EDF-schedulable task sets that are not covered.
- In this work we do not attempt to maximize the covering, but instead focus on identifying a few rather 'natural' subsets that are in NP. The covering presented here could most certainly be expanded – this is a direction of future work.
- We are here interested in the efficient explainability of the identified subsets – that it is possible to efficiently verify a certificate of a solution – but for now are not much concerned by the efficiency of finding those solutions (and certificates) in the first place. As a result, some subsets may in practice be more difficult computational problems to *solve* than the original EDF-schedulability problem, but they will be easier to verify.

4.1 Efficiently explainable subsets of uniprocessor EDF-schedulability

Here we will list efficiently explainable subsets (forming languages in NP) to the EDF-schedulability problem. The relationships between these subsets are shown in Figure 2. The most natural such subsets are *those for which the EDF-schedulability problem itself is easy to solve*. The following two subsets of EDF-schedulability are known to be in P.

[I] – Implicit deadlines. From Liu and Layland [27] we know that a task set Γ is EDF-schedulable if it has implicit deadlines and $\sum_{\tau_i \in \Gamma} C_i/T_i \leq 1$. Both conditions are trivial to check in polynomial time, and so EDF-schedulable task sets with implicit deadlines form a language in P (and therefore in NP).

[II] – Harmonic periods and constrained deadlines. Similarly, Bonifaci et al. [12] have given a polynomial-time algorithm for determining if a task set with constrained deadlines and harmonic periods is EDF-schedulable. Hence such task sets also form a language in P (and therefore in NP).

Another approach to identifying efficiently explainable subsets is by *exploiting the optimality of EDF upon preemptive uniprocessors.* From this optimality we trivially get that for any uniprocessor scheduling algorithm $\mathcal{A}$, the set of all of $\mathcal{A}$-schedulable task systems is a subset of the set of EDF-schedulable task systems. For which $\mathcal{A}$ is $\mathcal{A}$-schedulability in NP? An obvious candidate is Fixed-Priority (FP).

[III] – FP-schedulable with constrained deadlines. As explained in Section 2.1, the task systems that are FP-schedulable form a language in NP, and this language must be a subset of EDF-schedulability. We should use the optimal deadline-monotonic (DM) priority

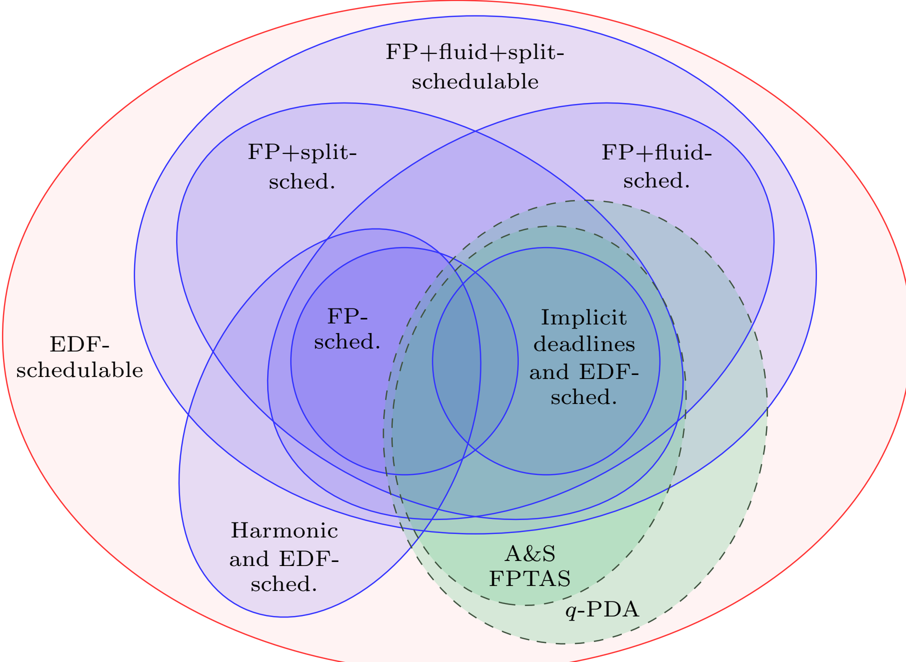


Figure 2 The relationships between the different languages of uniprocessor schedulability considered in this work. While the outer language of EDF-schedulability is coNP-complete, the marked subsets are all languages in NP. The union of any collection of these subsets is also in NP and therefore admits polynomial-time verifiable certificates. The two languages demarcated by dashed lines – the *Albers & Slomka FPTAS* and *EDF-schedulability by q-PDA* – are of a different sort than the others in that they are parameterized approximations; they will be described later in Section 5.

ordering to make the subset as large as possible. It is not known if FP-schedulability for task sets with *arbitrary* deadlines is in NP, but we can certainly truncate any $D_i > T_i$ to $D_i = T_i$ as preprocessing if we wish.

We note that Davis et al. [17] have shown that the considered subset of FP-schedulable task sets with constrained deadlines (and DM priorities) *itself contains as a strict subset* the set of all task sets with constrained deadlines that are EDF-schedulable on a speed-Ω processor, where $\Omega \approx 0.56714$ is the unique constant such that $\Omega e^{\Omega} = 1$. This gives us a bound on how much processor capacity we can lose in the worst case by proving EDF-schedulability by means of FP-schedulability.[8]

Are there more uniprocessor scheduling algorithms $\mathcal{A}$ for which $\mathcal{A}$-schedulability is in NP? There are very many, and the key to seeing this is to recognize that $\mathcal{A}$ *does not need to be any kind of "practically reasonable" scheduling algorithm*, since we do not intend to actually execute $\mathcal{A}$ during runtime – we are simply exploiting the fact that, since EDF is optimal,

[8] As a curiosity and an aside, this also means we have the marvelous situation that a coNP-complete language (EDF-schedulability on a speed-1 processor) contains an NP-complete language (FP-schedulability on a speed-1 processor) that in turn contains a coNP-complete language (EDF-schedulability on a speed-Ω processor). Indeed, we can make the infinite chain of strict subsets

$$(\text{EDF, speed-1}) \supset (\text{FP, speed-1}) \supset (\text{EDF, speed-}\Omega) \supset (\text{FP, speed-}\Omega) \supset (\text{EDF, speed-}\Omega^2) \supset \cdots$$

which alternate forever between being coNP- and NP-complete!

$\mathcal{A}$-schedulability necessarily implies EDF-schedulability. Thus we just want to efficiently verify EDF-schedulability by means of verifying $\mathcal{A}$-schedulability instead, but EDF, which has very efficient implementations, will be the run-time scheduling algorithm that is used. *Fluid schedulers* are among the scheduling algorithm that are easy to reason about, although they may be difficult to implement. Next, we consider a scheduler that schedules some tasks fluidly.

[IV] – FP+fluid-schedulability. A fluid scheduler assigns a constant fraction f_i of the processor to task τ_i, such that τ_i is served continuously at this rate. Clearly, τ_i will meet all of its deadlines if it scheduled fluidly with a rate $f_i = \delta_i$, where $\delta_i = C_i/\min(D_i, T_i)$ is the density of the task.

However, we do not need to schedule all tasks in the task set fluidly. We define the FP+fluid scheduler to be the scheduler that (optimally) partitions the task set Γ into two disjoint subsets Γ_{fluid} and Γ_{fp}, and then schedules each task $\tau_i \in \Gamma_{\text{fluid}}$ fluidly with rate δ_i, and schedules the tasks $\tau_i \in \Gamma_{\text{fp}}$ with an FP-scheduler (and DM priorities). The tasks in Γ_{fluid} run on a reserved processor fraction of speed Δ, where $\Delta = \sum_{\tau_i \in \Gamma_{\text{fluid}}} \delta_i$, and the tasks in Γ_{fp} run on the "remaining" processor fraction of speed $1 - \Delta$.

To see that FP+fluid-schedulability is in NP, consider that all tasks will meet their deadlines if both $\Delta \le 1$ and the tasks in Γ_{fp} are FP-schedulable on a processor of speed $1 - \Delta$. A certificate of FP+fluid-schedulability can then simply consist of (1) the partitioning of Γ into Γ_{fluid} and Γ_{fp}, and (2) fixed-points to the response-time equation of the tasks in Γ_{fp}, where each C_i has been multiplied by $1/(1 - \Delta)$. Given such a certificate we could easily verify in polynomial time that indeed $\Gamma = \Gamma_{\text{fluid}} \cup \Gamma_{\text{fp}}$, that $\Delta \le 1$, and that the given fixed-points are actual fixed-points ($\le D_i$) to the response-time equation for each task $\tau_i \in \Gamma_{\text{fp}}$ on a speed-$(1 - \Delta)$ processor.

FP+fluid-schedulability is of course a superset of plain FP-schedulability (since we can set $\Gamma_{\text{fluid}} = \emptyset$ and $\Gamma_{\text{fp}} = \Gamma$) and a superset of plain fluid scheduling (since we can set $\Gamma_{\text{fluid}} = \Gamma$ and $\Gamma_{\text{fp}} = \emptyset$). To see that FP+fluid-schedulability is in fact a strict superset of the union of both, we can consider the following simple task set:

$$\Gamma = \{\tau_1 = (2, 4, 4), \tau_2 = (3, 6, 8), \tau_3 = (1, 9, 10)\}$$

It can be readily checked that this task set is not fluid-schedulable (the total density > 1) and is also not FP-schedulable (τ_2 will miss a deadline under DM-priority ordering). It is however FP+fluid-schedulable with $\Gamma_{\text{fluid}} = \{\tau_1\}$ and $\Gamma_{\text{fp}} = \{\tau_2, \tau_3\}$.

FP+fluid-schedulability is an example of a subset of EDF-schedulability that seems potentially harder to *solve* in practice than just solving EDF-schedulability. For example, it is well-known that EDF-schedulability can be solved in pseudo-polynomial time for bounded-utilization task sets [7], but it is not obvious that FP+fluid-schedulability can be so solved if we want to find the best possible partitioning of Γ into Γ_{fluid} and Γ_{fp}. However, this is not our main concern, and being in NP it is qualitatively easier to verify the solutions of FP+fluid-schedulability.

Task splitting is another common scheduling technique that we can exploit to come up with suitable scheduling algorithms $\mathcal{A}$. Task splitting is known to often improve schedulability (see for example [14], where this technique is referred to as *period transformation*), but it normally comes with the drawback of some extra runtime overheads, especially if tasks are

split into many much smaller pieces. This is not a concern in this context as we again do not intend to ever run the resulting scheduling algorithm $\mathcal{A}$, we merely want to find $\mathcal{A}$ such that $\mathcal{A}$-schedulability is in NP.

[V] – FP+split-schedulability. We consider here the simple splitting technique where a constrained-deadline task $\tau_i = (C_i, D_i, T_i)$ can be split into the smaller task

$$\tau'_i = \left(\frac{C_i}{k_i}, \frac{T_i}{k_i} - (T_i - D_i), \frac{T_i}{k_i}\right),$$

for $k_i \in \mathbb{N}_+$, with which every job of τ_i is served as k_i jobs of τ'_i. It can be readily confirmed that if the k_i jobs of τ'_i all meet their deadlines, then so does the original job that they serve. Note that splitting a task τ_i with $D_i < T_i$ may result in it receiving a negative relative deadline, in which case the split task is clearly unschedulable.

FP+split-schedulability is in NP since we can provide as a certificate the k_i's and fixed-points to the response-time equation for the split tasks τ'_i. The solution is then verified in polynomial time by reproducing the split tasks using the k_i's (some tasks may have $k_i = 1$, and remain unsplit) and verifying that the provided fixed-points are indeed valid fixed-points to the response-time equation that are each $\leq \frac{T_i}{k_i} - (T_i - D_i)$.

Now that we have FP+fluid and FP+split, nothing is stopping us from combining the power of both, if indeed verification time and not solution time is our main concern, because combining them will not take us out of NP.

[VI] – FP+fluid+split-schedulability. The FP+fluid+split scheduler simply partitions the task set Γ into Γ_{fluid} and Γ_{fp}, and then schedules the tasks in Γ_{fluid} fluidly and then allows the tasks in Γ_{fp} to be split into smaller tasks before they are scheduled by an FP scheduler on the remaining processor fraction. Polynomial-time verifiable certificates are easily constructed similarly to how they are constructed for FP+fluid and FP+split.

We note that FP+fluid-schedulability [IV] and FP+split-schedulability [V] both contain as subsets the plain FP-schedulability for constrained deadlines [III] as well as EDF-schedulability with implicit deadlines [I]. As we will see below, neither contain the other though, they each cover different parts of the original EDF-schedulability problem. The following is a simple task set that can be readily checked to be unschedulable by FP+fluid, but schedulable by FP+split (by splitting τ_1 into $\tau'_1 = (1, 1, 2)$):

$$\Gamma = \{\tau_1 = (2, 3, 4), \tau_2 = (3, 6, 6)\}$$

In the other direction, the following example can be checked to be unschedulable by FP+split (no task can be split and keep a non-negative relative deadline), but to be schedulable by FP+fluid (by setting $\Gamma_{\text{fluid}} = \{\tau_3\}$ and $\Gamma_{\text{fp}} = \{\tau_1, \tau_2\}$):

$$\Gamma = \{\tau_1 = (1, 2, 9), \tau_2 = (7, 9, 100), \tau_3 = (1 + \epsilon, 10, 100)\}$$

Now that we have seen that FP+fluid-schedulability and FP+split-schedulability cover different parts of the EDF-schedulability language, we note that FP+fluid+split-schedulable is in fact more than just the union of these two. This is demonstrated by the following example, which is unschedulable by both FP+fluid and FP+split, but is schedulable by FP+fluid+split:

$$\Gamma = \{\tau_1 = (3, 6, 8), \tau_2 = (7, 12, 100), \tau_3 = (1/2 + \epsilon, 13, 100)\}$$

It is readily confirmed that none of the eight possible choices for which subset of tasks to schedule fluidly would cause Γ to be FP+fluid-schedulable, and none of the possible task splittings (only τ_1 can be split and keep a positive deadline) would cause Γ to be FP+split-schedulable. However, by splitting τ_1 into $\tau_1' = (3/2, 2, 4)$ and by setting $\Gamma_{\text{fluid}} = \{\tau_3\}$ and $\Gamma_{\text{fp}} = \{\tau_1', \tau_2\}$, the task system is indeed FP+fluid+split-schedulable (for small enough ϵ).

The above list of sub-problems in NP to uniprocessor EDF-schedulability is by no means comprehensive. It is always possible to come up with additional artificial sub-problems in NP (for example by hard-coding schedulable task systems), and quite likely there are several more "natural" sub-problems other than the ones presented here as well.

We note that while the above is presented for three-parameter sporadic or synchronous periodic tasks, we can trivially use exactly the same sub-problems for asynchronous periodic tasks as well, by simply ignoring the release offsets of all tasks. This is because the synchronous arrival sequence is the worst-case for uniprocessor EDF [7], so if we can show EDF-schedulability for the corresponding *synchronous* task system (for example using one of the explainable sub-problems above), then we immediately show it also for the asynchronous task system.

4.2 Efficiently explainable subsets of partitioned EDF-schedulability

Similarly to the FP-schedulability case in Section 3, as soon as we have efficient explainability of a uniprocessor schedulability problem, then we automatically get efficient explainability of the corresponding partitioned multiprocessor schedulability problem. This is the case even though the partitioned multiprocessor variant may be much harder to *solve* than the uniprocessor variant. For instance, while the EDF-schedulability problem is "only" coNP-complete on uniprocessors, it is both NP- and coNP-hard on partitioned multiprocessors, thus unlikely to be even in coNP. Indeed, the partitioned EDF-schedulability problem for asynchronous task systems is Σ_2^{P}-complete [21], and is a much harder problem than for uniprocessors. Even so, partitioned EDF-schedulability is no less explainable than the uniprocessor variant. A certificate of partitioned EDF-schedulability can simply consist of the partitioning $\Gamma_1, \ldots, \Gamma_m$ of Γ upon the m processors, together with a certificate of uniprocessor EDF-schedulability for each of the m partitions. This certificate is verified by checking that indeed $\Gamma = \Gamma_1 \cup \cdots \cup \Gamma_m$, and by verifying the per-partition certificates. Note that the per-partition certificates do not have to be of the same type, they could for example be certificates from different sub-problems listed earlier in this section.

5 A Scheme for Efficient Explainability

In Section 4 we explored two approaches to identifying efficiently explainable sub-problems of the EDF-schedulability analysis problem: (i) restricting the problem instances (in [I] and [II]); and (ii) instead testing schedulability for some other scheduler that is dominated by EDF (in [III]–[VI]). Here we propose a third approach: directly designing a sufficient schedulability test for EDF-schedulability analysis. For this the sufficient test should define a language in NP, in the sense that for all task systems that pass the test, we can create polynomial-time verifiable certificates of this fact.

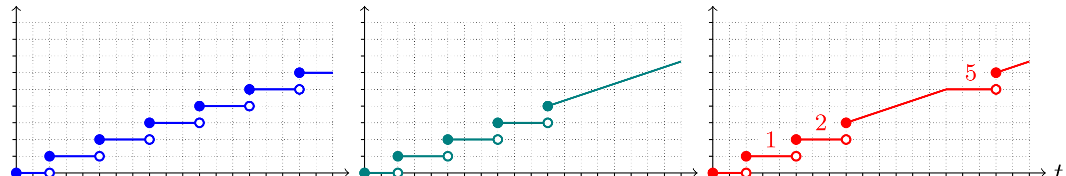


Figure 3 The left plot depicts $\text{DBF}_i(t)$ for a task τ_i with $C_i = 1, D_i = 2$, and $T_i = 3$. The center plot depicts $\overline{\text{DBF}}_i(t)$ with $k = 3$, and the right plot, $\overline{\text{DBF}}_i(t, \mathcal{S}_i)$ for $\mathcal{S}_i = \{1, 2, 5\}$. Note that $\overline{\text{DBF}}_i(t)$ ($\overline{\text{DBF}}_i(t, \mathcal{S}_i)$, respectively) is discontinuous only at the first k steps (only around the steps in $\mathcal{S}_i$, resp.), and is piecewise linear with slope $\leq C_i/T_i$ everywhere else.

As stated in Section 2.1, processor-demand analysis (PDA) is the standard test for EDF-schedulability. PDA checks that Eq. 3 (reproduced below):

$$\sum_{\tau_i \in \Gamma} \text{DBF}_i(t) \leq t$$

holds for all values of $t \in \mathcal{T}(\Gamma)$, where $\mathcal{T}(\Gamma)$ is the potentially exponentially-sized testing set. A sufficient EDF-schedulability test is easily obtained by replacing each task's demand bound function (the $\text{DBF}_i(t)$ terms in Eq. 3) with an approximation $\overline{\text{DBF}}_i(t)$, such that $\text{DBF}_i(t) \leq \overline{\text{DBF}}_i(t)$ for all t. It is evident (see Eq. 2) that $\text{DBF}_i(t)$ is a step function with a zeroth step over interval $[0, D_i)$, a first step over interval $[D_i, T_i + D_i)$ and so on – see the left plot in Fig. 3. Albers and Slomka [1] proposed the following approximation to $\text{DBF}_i(t)$ (depicted in the center plot in Fig. 3): letting k denote any positive integer constant, their approximation retains steps zero through k while the remainder is over-approximated by a straight line with slope equal to the task's utilization:

$$\overline{\text{DBF}}_i(t) = \begin{cases} \text{DBF}_i(t), & \text{for } t < D_i + kT_i \\ (T_i - D_i + t) \times \frac{C_i}{T_i}, & \text{for } t \geq D_i + kT_i \end{cases} \tag{4}$$

We note that $\overline{\text{DBF}}_i(t)$ is discontinuous only at points $t = \ell T_i + D_i$ for some $\tau_i \in \Gamma$ and $\ell \in \{0, 1, \ldots, k\}$. It therefore follows that the left-hand side of the approximated version of Eq. 3, i.e., $\sum_{\tau_i \in \Gamma} \overline{\text{DBF}}_i(t)$, is discontinuous at no more than $((k+1) \times |\Gamma|)$ points, and is piecewise linear with slope at most $\sum_{\tau_i \in \Gamma} C_i/T_i$ elsewhere. Assuming $\sum_{\tau_i \in \Gamma} C_i/T_i \leq 1$, we therefore only need to evaluate the approximated version of Eq. 3 at the $\leq ((k+1) \times |\Gamma|)$ points of discontinuity; for constant k, this yields a polynomial-time sufficient EDF-schedulability test. It was also shown in [1] that any Γ that is deemed to not be EDF-schedulable by this sufficient test is guaranteed to actually not be EDF-schedulable upon a speed-$(k/(k+1))$ processor. Since k may take on any value, the Albers and Slomka polynomial-time sufficient test [1] is therefore an FPTAS (see Definition 1) for EDF schedulability analysis; as mentioned in Section 2, FPTAS's are considered to be the 'best' kind of approximation algorithm (that runs in polynomial time).

A scheme for efficient explainability. We can directly use the Albers and Slomka FPTAS [1] to design a scheme for efficient explainability, in the following manner. Suppose that we wish to explain, with a speedup factor[9] equal to $(1 + \delta)$, that some task system Γ is EDF-schedulable – we can do so by simply using the Albers and Slomka [1] over-approximation

[9] I.e., if we fail to explain the EDF-schedulability of Γ then Γ is in fact guaranteed to not be EDF-schedulable upon a speed-$(1/(1 + \delta))$ processor.

of the demand bound function with $k \leftarrow \lceil 1/\delta \rceil$. Since the running time of the Albers and Slomka [1] test is polynomial in $|\Gamma|$ and k, i.e., polynomial in $|\Gamma|$ and $(1/\delta)$, it follows that this does indeed constitute a scheme for efficient explainability of EDF schedulability.

An improved scheme for efficient explainability. The scheme described in the previous paragraph is obtained by direct application of the FPTAS from [1]. But we can in fact improve on this for the purposes of efficient explainability: There is no particular reason why it is the *first* k steps of $\overline{\text{DBF}}_i$ that must be exact (not over-approximated), nor why the $\overline{\text{DBF}}_i$ of each task τ_i must be exact for the *same* number of steps (the same value of k).

Let us examine these ideas a bit further. Observe that in the function DBF_i the interval of the ℓth step, $\ell \geq 1$, is given by

$$\text{step}_i(\ell) = \big[(\ell - 1)T_i + D_i,\ \ell T_i + D_i\big). \tag{5}$$

For any $\mathcal{S}_i \subset \mathbb{N}_+$, let us define $\overline{\text{DBF}}_i(t, \mathcal{S}_i)$ to be the approximation to $\text{DBF}_i(t)$ that agrees with $\text{DBF}_i(t)$ over the intervals of the steps in $\mathcal{S}_i$, and is a linear over-approximation elsewhere:

$$\overline{\text{DBF}}_i(t, \mathcal{S}_i) = \begin{cases} 0, & \text{if } t < D_i \\ \text{DBF}_i(t), & \text{if } t \in \text{step}_i(\ell) \text{ for some } \ell \in \mathcal{S}_i \\ (T_i - D_i + t)\frac{C_i}{T_i}, & \text{otherwise} \end{cases} \tag{6}$$

The approximation $\overline{\text{DBF}}_i(t, \mathcal{S}_i)$ is illustrated in the right-most plot of Fig. 3. For any $\mathcal{S}_i$, we have $\text{DBF}_i(t) \leq \overline{\text{DBF}}_i(t, \mathcal{S}_i)$ for all t. By picking some set of steps $\mathcal{S}_i$ for each task τ_i and then replacing $\text{DBF}_i(t)$ by $\overline{\text{DBF}}_i(t, \mathcal{S}_i)$ in Eq. 3 we therefore get a sufficient EDF-schedulability test.

As with the Albers and Slomka [1] approximation, we note that $\overline{\text{DBF}}_i(t, \mathcal{S}_i)$ is discontinuous only at points $t = \ell T_i + D_i$ where $\ell \in \mathcal{S}_i \cup \{0\}$, and is piecewise linear with slope at most C_i/T_i elsewhere. It follows that $\sum_{\tau_i \in \Gamma} \overline{\text{DBF}}_i(t, \mathcal{S}_i)$ is discontinuous at most at $\sum_{\tau_i \in \Gamma}(|\mathcal{S}_i| + 1)$ points, and is piecewise linear with slope at most $\sum_{\tau_i \in \Gamma} C_i/T_i$ elsewhere. Assuming $\sum_{\tau_i \in \Gamma} C_i/T_i \leq 1$, we therefore only need to evaluate the approximated version of Eq. 3 at the points of discontinuity (that are less $\text{HP}(\Gamma)$). If $\sum_{\tau_i \in \Gamma}(|\mathcal{S}_i| + 1)$ is bounded by a polynomial in the size of the task system, then we can check all points in the testing set in polynomial time.

For a fixed polynomial q, we let q-PDA be the subset of all task systems Γ for which there exists sets $\mathcal{S}_i \subset \mathbb{N}_+$ for each $\tau_i \in \Gamma$, such that $\sum_{\tau_i \in \Gamma} \overline{\text{DBF}}_i(t, \mathcal{S}_i) \leq t$ for all $0 \leq t \leq \text{HP}(\Gamma)$ and $\sum_{\tau_i \in \Gamma}(|\mathcal{S}_i| + 1) \leq q(|\Gamma|)$. By the reasoning above, q-PDA is in $\mathtt{NP}$ since a task set Γ can be verified to be in q-PDA in polynomial time if given the sets $\mathcal{S}_i$ as a certificate.

While it may require significant effort to find the sets $\mathcal{S}_i$, the q-PDA has the potential to allow much more efficient verification of solutions than the Albers and Slomka FPTAS. In fact, using q-PDA we may exponentially decrease the discontinous points that need to be checked in Eq. 3. The following simple example demonstrates this.

▶ **Example 4.** Consider the task set $\Gamma = \{\tau_1 = (1, 1, 2), \tau_2 = (\alpha/2, \alpha, 2\alpha)\}$, where α is some large even number. It can be readily confirmed that Γ is EDF-schedulable. However, using the approximation in Eq. 4 with $k < \alpha/2$ we get $\sum_{\tau_i \in \Gamma} \overline{\text{DBF}}_i(\alpha) = \alpha + 1/2$, and therefore we need $k \geq \alpha/2$ to establish that Γ is schedulable with the Albers and Slomka FPTAS, for a total of at least α discontinuous points to check. However, using q-PDA and the approximation in Eq. 6 with $\mathcal{S}_1 = \{\alpha/2\}$ and $\mathcal{S}_2 = \{1\}$, we can establish that Γ is schedulable by only checking 4 discontinuous points. In other words, Γ is in q-PDA with $q(n) = 2n$. ◀

How much of EDF-schedulability that is covered by q-PDA depends very much on 'q, which makes it a type of parameterized approximation. This motivates the following definition, which extends the concept of FPTAS's for schedulability tests that was introduced (Definition 1) in Section 2.

▶ **Definition 5** (FPTVAS). *A fully polynomial-time* ***verification*** *approximation scheme (FPTVAS) for a schedulability analysis problem is an algorithm that, given as input an instance* Γ*, a parameter* $\delta > 0$*,* ***and a certificate****, returns "unschedulable" if* Γ *is unschedulable on a speed-1 processor, and returns "schedulable" if* Γ *is schedulable on a speed-*$(1/(1+\delta))$ *processor. Its running time is bounded by a polynomial in the two parameters* $|\Gamma|$ *and* $\left(\frac{1}{\delta}\right)$*.*

While our definition of FPTAS's for schedulability analysis problems (Definition 1) is essentially an instantiation of the preëxisting concept of FPTAS's from approximation theory [3, 34], the notion of FPT**V**AS's in Definition 5 above is, to our knowledge, novel – we are not aware of prior work in complexity theory that lifts the concept of FPTAS's from polynomial-time computation (i.e., the complexity class P) to polynomial-time verification (the class NP).

5.1 Extension to Multiprocessors

In contrast to uniprocessor EDF schedulability where an FPTAS is known to exist [1], no FPTAS is known for multiprocessor partitioned EDF schedulability – indeed, a lower bound of 1.5026 was recently obtained [28] on the speedup factor of the state-of-the-art partitioned EDF scheduling heuristic [9]. We cannot therefore simply use a preëxisting FPTAS to obtain an approximation scheme for efficient explainability of partitioned multiprocessor EDF schedulability. We can, however, extend the FPTVAS for uniprocessor EDF schedulability obtained above in the following manner to obtain an FPTVAS for partitioned EDF-schedulability of sporadic / synchronous periodic task systems. Suppose task system Γ is EDF-schedulable upon being partitioned upon an m-processor platform. For any fixed polynomial function q, the certificate of its schedulability would consist of

1. The partitioning $\Gamma_1, \Gamma_2, \ldots, \Gamma_m$ of Γ amongst the m processors; and
2. For each partition Γ_j, $1 \leq j \leq m$, a certificate of its uniprocessor EDF schedulability. Such a certificate would comprise the sets $\mathcal{S}_i$ for each $\tau_i \in \Gamma_j$, together satisfying the constraints that $\sum_{\tau_i \in \Gamma_j} \overline{\text{DBF}}_i(t, \mathcal{S}_i) \leq t$ for all $0 \leq t \leq \text{HP}(\Gamma_j)$ and $\sum_{\tau_i \in \Gamma_j}(|\mathcal{S}_i| + 1) \leq q(|\Gamma_j|)$.

As in the uniprocessor case, this FPTVAS immediately implies an approximation scheme for efficient explainability of partitioned multiprocessor EDF schedulability.

6 Explainability Beyond Polynomial-time

There are many examples in real-time scheduling theory where not only polynomial-time algorithms are considered to be efficient, but also *pseudo-polynomial* time algorithms. (E.g., both response-time analysis [23] for FP-schedulability and the processor-demand approach [7] for EDF-schedulability of bounded-utilization systems are widely used schedulability analysis algorithms that have pseudo-polynomial running times.) The fundamental reason for why pseudo-polynomial time algorithms are often considered as being efficient in real-time scheduling is simply that the numerical parameters that appear in scheduling problems tend to have some direct physical meaning. For example, the parameters C_i, D_i and T_i of a sporadic task are supposed to represent physical time in some given unit, and we would therefore not expect to be given input instances with numerical parameters that are too large to be meaningful on a human timescale. Whether a pseudo-polynomial time algorithm is to be considered efficient certainly depends on what we expect the inputs to look like.

If we do accept pseudo-polynomial time as acceptably efficient in the context of solving problems, should we also accept pseudo-polynomial time as acceptably efficient for *verifying* solutions? We see no particular reason to not do so, if the problem is such that numerical values tend to be reasonably small. This motivates the following definition.

▶ **Definition 6** (pseudoNP). *We can consider* pseudoP *as the complexity class of problems that can be solved by a pseudo-polynomial time algorithm. Equivalently,* pseudoP *contains the problems that would be in* P *if numbers were written in unary. Analogously, we define* pseudoNP *as the class of problems that would be in* NP *if numbers were written in unary. The class* pseudoNP *then contains the problems for which there exist pseudo-polynomially sized certificates that can be verified in pseudo-polynomial time.*

The classes pseudoP and pseudoNP do not fit so neatly into the hierarchy of complexity classes shown in Figure 1. We can see that pseudoP of course contains P, but must also be contained in EXP since the value of a (naturally represented) numerical parameter is at most exponential in the total length of the input, and therefore any polynomial in the value of the largest number cannot be larger than exponential in the input length. Further, any problem in EXP can be (artificially) transformed to allow a pseudo-polynomial time algorithm by padding input instances with large numbers. Hence, pseudoP intersects with all the classes up to EXP in Figure 1, but only completely contains P. By similar arguments, pseudoNP contains NP, is contained in NEXP and intersects with all other classes in Figure 1, including NEXP. The class pseudoNP captures an interesting property of problems, which to the authors' knowledge is not well-studied.

What problems can be found in pseudoNP that are neither in pseudoP nor in NP? Partitioned EDF-schedulability is again a good example. We know that uniprocessor EDF-schedulability of three-parameter sporadic (or synchronous periodic) tasks can be solved in pseudo-polynomial time if the utilization of task systems is bounded by some constant $c < 1$ (say, $c = 0.99$). [7] On the other hand, we know that partitioned EDF-schedulability is both NP- and coNP-hard, even with utilization bounded by any $c > 0$ [21], and thus is not in NP (unless NP = coNP). It is not difficult to see that this is the case even if we enforce a per-partition utilization bound of c. Also, the partitioned problem is NP-hard in the strong sense (as it generalizes Bin-Packing) and is therefore not in pseudoP (unless P = NP). However, the partitioned EDF-schedulability problem with a per-partition utilization bound of $c < 1$ *is* in pseudoNP. A certificate for this problem can simply consist of the partitioning, and the verifier can check that each partition has a utilization of at most c, and then directly verify the schedulability of each partition in pseudo-polynomial time using the standard PDA test [7].

An example of a problem that is *not* in pseudoNP is the uniprocessor EDF-schedulability problem for unbounded-utilization task systems. Since this problem is coNP-complete in the strong sense [20], it is (by definition) coNP-complete also if numbers are written in unary. Since the unary version is coNP-complete, it cannot be in NP (unless NP = coNP), and therefore the uniprocessor EDF-schedulability problem for unbounded-utilization task systems is not in pseudoNP.

7 Discussion

1. We have classified several multiprocessor schedulability analysis problems as efficiently explainable or not.
2. Using uniprocessor EDF schedulability analysis as a concrete example problem, we have developed multiple distinct methods for identifying efficient explainable sub-problems of problems that are not efficiently explainable. (We have also applied these methods to partitioned multiprocessor EDF schedulability analysis.)

3. We have extended the notion of FPTAS's, which are widely studied for approximation algorithms, from approximately solving a problem to verifying an approximate solution – this yields the novel concept of FPTVAS's (Definition 5), and extends the idea of approximation schemes to schemes for efficient explainablity.
4. We have extended the concept of pseudo-polynomial time algorithms, which are increasingly coming to be accepted as being 'efficient enough' for pre-run-time analysis (such as schedulability analysis), from efficient determination of schedulability to efficient explanation of schedulability, by defining the novel complexity class `pseudoNP` (Definition 6).

Our contributions in all these aspects are by no means complete or comprehensive – a large amount of work remains to be done in both classifying the explainability or non-explainability of other important schedulability-analysis problems, and in identifying efficiently-explainable sub-problems for those determined to not be efficiently explainable. Additionally, our explorations of the concepts of FPTVAS's (Definition 5) and `pseudoNP` (Definition 6) are quite basic – we believe both these concepts are potentially very meaningful and so merit considerable additional investigation.

We reiterate that we believe the use of formal, machine-verifiable proofs (such as those in Maida et al. [29]) is a promising way forward if we wish to use advanced techniques and recent developments in real-time systems research to explain schedulability to a certification authority (CA). Rather than trying to convince the CA that, say, an analysis for FP+fluid+split is sound and can be used to indirectly prove EDF-schedulability, such details can all be contained in the formal proof. The CA needs only to trust the proof assistant itself (e.g., Coq) and agree with the model of the system and basic definitions. The stated goal of the certification step (e.g., that the system is EDF-schedulable) is then guaranteed by the proof produced by the proof assistant, and can be trusted without even knowing the particular proof strategy employed. With this work we want to put forward the idea that techniques for enabling efficient explainability, as outlined in brief in this paper, could guide the creation of such machine-verifiable proofs that may indeed be *verified efficiently*, even as systems grow in size and complexity.

Finally, a discussion on verifiability of solutions to computational problems would not be complete without mentioning interactive proof systems. `IP` is the complexity class of problems where a *verifier* with only polynomial computational resources can be convinced (to an arbitrary degree of certainty) by an all-powerful *prover* that a valid solution exists to a given problem instance. In contrast to the static certificates –the explanations– considered in this paper, interactive proof systems work by letting the verifier and prover interactively exchange messages with each other, where the verifier challenges the explanations of the prover by asking specially-crafted (and randomized) questions. `IP` was shown to equal `PSPACE` (see Figure 1) in a landmark result [33], meaning that very many practical problems have such interactive proof systems. While interactive proof systems come with their own set of significant challenges, requiring interactive communication and accepting a small probability of incorrectly verified solutions, we believe that they have a place in explainability of real-time systems as well, and represent an interesting direction for future work.

References

1 K. Albers and F. Slomka. An event stream driven approximation for the analysis of real-time systems. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, pages 187–195, Catania, Sicily, July 2004. IEEE Computer Society Press.

2 Sanjeev Arora and Boaz Barak. *Computational Complexity – A Modern Approach*. Cambridge University Press, 2009.

3 Giorgio Ausiello, Alberto Marchetti-Spaccamela, Pierluigi Crescenzi, Giorgio Gambosi, Marco Protasi, and Viggo Kann. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer Verlag, 1999.

4 T. P. Baker and A. Shaw. The cyclic executive model and Ada. In *Proceedings of the 9th Real-Time Systems Symposium (RTSS)*, pages 120–129, 1988.

5 T. P. Baker and A. Shaw. The cyclic executive model and Ada. *Real-Time Systems*, 1(1):7–25, 1989.

6 S. Baruah, R. Howell, and L. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems: The International Journal of Time-Critical Computing*, 2:301–324, 1990.

7 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 (RTSS)*, pages 182–190, Orlando, Florida, 1990. IEEE Computer Society Press.

8 Sanjoy Baruah and Pontus Ekberg. Certificates of real-time schedulability. In *International Workshop on Explainability of Real-time Systems and their Analysis (ERSA)*, 2022.

9 Sanjoy Baruah and Nathan Fisher. The partitioned multiprocessor scheduling of deadline-constrained sporadic task systems. *IEEE Transactions on Computers*, 55(7):918–923, July 2006.

10 Kimaya Bedarkar, Mariam Vardishvili, Sergey Bozhko, Marco Maida, and Björn B. Brandenburg. From intuition to Coq: A case study in verified response-time analysis of FIFO scheduling. In *IEEE Real-Time Systems Symposium, RTSS 2022, Houston, TX, USA, December 5-8, 2022*, pages 197–210. IEEE, 2022. `doi:10.1109/RTSS55097.2022.00026`.

11 Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013. Project website: `https://coq.inria.fr`.

12 V. Bonifaci, A. Marchetti-Spaccamela, N. Megow, and A. Wiese. Polynomial-time exact schedulability tests for harmonic real-time tasks. In *Proceedings of the 34th Real-Time Systems Symposium (RTSS)*, pages 236–245, December 2013.

13 Sergey Bozhko and Björn B. Brandenburg. Abstract Response-Time Analysis: A Formal Foundation for the Busy-Window Principle. In Marcus Völp, editor, *Proceedings of the 32nd Euromicro Conference on Real-Time Systems (ECRTS)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:24, Dagstuhl, Germany, 2020. `doi:10.4230/LIPIcs.ECRTS.2020.22`.

14 Björn B. Brandenburg and Mahircan 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)*, pages 99–110, 2016. `doi:10.1109/RTSS.2016.019`.

15 Felipe Cerqueira, Felix Stutz, and Björn B. Brandenburg. PROSA: A case for readable mechanized schedulability analysis. In *Proceedings of the 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 273–284, 2016. `doi:10.1109/ECRTS.2016.28`.

16 T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.

17 Robert Davis, Thomas Rothvoss, Sanjoy Baruah, and Alan Burns. Exact quantification of the sub-optimality of uniprocessor fixed priority pre-emptive scheduling. *Real-Time Systems: The International Journal of Time-Critical Computing*, 43(3):211–258, 2009.

18 Friedrich Eisenbrand and Thomas Rothvoß. EDF-schedulability of synchronous periodic task systems is coNP-hard. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, January 2010.

19 P. Ekberg and W. Yi. Uniprocessor feasibility of sporadic tasks remains coNP-complete under bounded utilization. In *Proceedings of the 36th Real-Time Systems Symposium (RTSS)*, pages 87–95, 2015.

20 P. Ekberg and W. Yi. Uniprocessor feasibility of sporadic tasks with constrained deadlines is strongly coNP-complete. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 281–286, 2015.

21 Pontus Ekberg and Sanjoy Baruah. Partitioned scheduling of recurrent real-time tasks. In *Proceedings of the 42nd Real-Time Systems Symposium (RTSS)*, pages 356–367, 2021. `doi:10.1109/RTSS52674.2021.00040`.

22 Pontus Ekberg and Wang Yi. Fixed-priority schedulability of sporadic tasks on uniprocessors is NP-hard. In *Proceedings of the 38th Real-Time Systems Symposium (RTSS)*, pages 139–146. IEEE Computer Society, 2017. `doi:10.1109/RTSS.2017.00020`.

23 M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, October 1986.

24 B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. In *36th Annual Symposium on Foundations of Computer Science (FOCS'95)*, pages 214–223, Los Alamitos, October 1995. IEEE Computer Society Press.

25 B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM*, 37(4):617–643, 2000.

26 J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the 10th Real-Time Systems Symposium (RTSS)*, pages 166–171. IEEE Computer Society Press, December 1989.

27 C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

28 Xingwu Liu, Zizhao Chen, Xin Han, Zhenyu Sun, and Zhishan Guo. Tighter bounds of speedup factor of partitioned EDF for constrained-deadline sporadic tasks. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 431–440, 2021. `doi:10.1109/RTSS52674.2021.00046`.

29 Marco Maida, Sergey Bozhko, and Björn B. Brandenburg. Foundational Response-Time Analysis as Explainable Evidence of Timeliness. In *Proceedings of the 34th Euromicro Conference on Real-Time Systems (ECRTS)*, volume 231 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:25, Dagstuhl, Germany, 2022. `doi:10.4230/LIPIcs.ECRTS.2022.19`.

30 Aloysius 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.

31 Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

32 Cynthia A. Phillips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 140–149, El Paso, Texas, 4–6 May 1997.

33 Adi Shamir. IP = PSPACE. *J. ACM*, 39(4):869–877, October 1992. `doi:10.1145/146585.146609`.

34 Vijay V. Vazirani. *Approximation Algorithms*. Springer-Verlag, Berlin-Heidelberg-New York-Barcelona-Hong Kong-London-Milan-Paris-Singapur-Tokyo, 2001.

35 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53, May 2008.

The Safe and Effective Use of Low-Assurance Predictions in Safety-Critical Systems

Kunal Agrawal
Washington University in Saint Louis, MO, USA

Sanjoy Baruah
Washington University in Saint Louis, MO, USA

Michael A. Bender
Stony Brook University, NY, USA

Alberto Marchetti-Spaccamela
Sapienza Università di Roma, Italy

Abstract

The algorithm-design paradigm of *algorithms using predictions* is explored as a means of incorporating the computations of lower-assurance components (such as machine-learning based ones) into safety-critical systems that must have their correctness validated to very high levels of assurance. The paradigm is applied to two simple example applications that are relevant to the real-time systems community: energy-aware scheduling, and classification using ML-based classifiers in conjunction with more reliable but slower deterministic classifiers. It is shown how algorithms using predictions achieve much-improved performance when the low-assurance computations are correct, at a cost of no more than a slight performance degradation even when they turn out to be completely wrong.

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

Keywords and phrases Algorithms using predictions, robust scheduling, energy minimization, classification, on-line scheduling

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.3

Funding *Sanjoy Baruah*: US National Science Foundation (Grants CPS- 1932530, CNS-2141256, and CNS-2229290)
Michael A. Bender: US National Science Foundation (Grants CCF-2247577 and CCF-2106827)
Alberto Marchetti-Spaccamela: ERC 788893 AMDROMA, Sapienza Res. Proj. RM11816436B2F886

1 Introduction

The increasing use of Learning-Enabled Components (LECs) in safety-critical applications such as autonomous driving present unique challenges to the discipline of safety-critical systems engineering. On the one hand, the availability of such LECs dramatically enhances the capabilities of autonomous cyber-physical systems – it is hard to conceive of, e.g., self-driving cars that do not make extensive use of LECs such as Deep Neural Networks for perception. But on the other hand, most widely used LECs cannot provide performance guarantees at high enough levels of assurance that they may be used in the verification and validation processes that are so fundamental to safety-critical systems engineering.

Context. A recent line of work in the algorithms community, *algorithms using predictions*, offers a promising approach for incorporating low-assurance predictions of the kind that are typically provided by LECs into algorithms designed for use in safety-critical systems. Algorithms using predictions are intended to perform very well when the predictor is accurate

© Kunal Agrawal, Sanjoy Baruah, Michael A. Bender, and Alberto Marchetti-Spaccamela;
licensed under Creative Commons License CC-BY 4.0

(this property is called *consistency* – see Sec. 3), while simultaneously guaranteeing to provide acceptable performance regardless of the quality of the predictions (called *robustness* – also formally defined in Sec. 3).

In this paper, we investigate the applicability of this framework to safety-critical systems. We have found that the framework needs to be adapted for this purpose due, amongst other reasons discussed in Sec. 3, to the presence of hard constraints (e.g., deadlines that must be met) in many safety-critical systems. Most prior work on algorithms using predictions has focused upon pure optimization problems that do not include such hard constraints[1] – indeed, amongst the 28 papers tagged with the keyword "scheduling" in the comprehensive list of publications on the topic that is maintained on the Algorithms With Predictions web-site,[2] we found only two [2, 3] that consider workload models with hard deadlines. (We will discuss these two papers further in Sec. 2.)

Our Contributions. In this paper, we *adapt* the algorithms using predictions framework for use in safety-critical systems and *provide an introductory explanation* of the adapted framework, specifically written to be comprehensible with moderate effort by members of the real-time systems community who may have had no prior exposure to algorithms using predictions. This explanation emphasizes the main features of the framework, and makes the case that this framework can contribute to the ongoing efforts of our community at incorporating LECs into safety-critical CPS's. In addition, we *illustrate the applicability* of this framework to real-time systems by considering the following two problems that are of interest to the real-time systems community:

1. As a first example illustration, we examine a problem that has been studied extensively in our community: energy-efficient hard-real-time scheduling. One of the main challenges encountered in this problem is what is commonly referred to as "the WCET problem" [16] – the non-determinism and variation in the execution duration of individual pieces of code upon modern computing platforms. In hard-real-time systems, the WCET problem requires that computing capacity be provisioned to accommodate a conservative worst-case upper bound on the needed amount of computation; we will see that this can lead to excessive energy being consumed during run time. If a reasonably good (although not guaranteed to be correct) prediction on the actual execution duration of the code is also available, then we will see that the algorithms using predictions framework can realise considerable energy savings.
2. Our second example illustration considers a form of LECs called IDK classifiers that has recently garnered some attention in the real-time systems community (see, e.g., [1, 5, 6]). Classifiers are used for classifying sensor readings into one of a set of a classes (e.g., a camera input to "pedestrian", "stop sign", etc.); LEC-based classifiers are generally not able to guarantee to do so at a high enough level of assurance and so should be backed up by deterministic – i.e., not learning-enabled – classifiers (including, perhaps human intervention). We will examine how the framework of algorithms using predictions can nevertheless use LEC-based classifiers to provide improved performance in the form of reduced expected response times.

[1] E.g., a paper at RTSS last year [18] proposed an algorithm using predictions for soft real-time scheduling to minimize the average response time – it is not at all obvious how one would adapt this algorithm to handle hard deadlines.

[2] `https://algorithms-with-predictions.github.io` (Accessed on 25/02/2023.)

Organization. The remainder of this manuscript is arranged as follows. In Section 2 we step through the process of designing an algorithm using predictions for a very simple energy-aware scheduling application, emphasizing, at each point in the presentation, the role that the predictions play in the algorithm design and the consequent implications on algorithm performance. This example application will give us an intuitive understanding of the algorithm-design paradigm of algorithms using predictions; we formalize this intuition in Section 3 and explicitly articulate the elements of the paradigm. We provide a second, somewhat more sophisticated, illustration of the application of this paradigm in Section 4, where we design an algorithm using predictions for minimizing the expected duration needed to successfully complete a classification operation. We close in Section 5 by placing this work within a larger context of safety-critical systems design, and some suggested directions of follow-up research.

2 Illustrative Example: Energy-Efficient Scheduling

In this section we apply the algorithms using predictions framework, suitably adapted as needed, to a problem that has been widely studied in the real-time systems community: scheduling to minimize energy consumption (see, e.g., the surveys [4, 9] and the references cited therein). We will show how predictions of run-time behavior, when available, can be used to significantly reduce energy consumption when the predictions are somewhat accurate while not requiring too much additional energy when the predictions are completely wrong.

This section is organized as follows. We first briefly describe (Sec. 2.1) the system model we will use, in which a job is characterized by its worst-case execution time, which is guaranteed to be a safe upper bound on its actual execution duration, as well as a prediction (which may be inaccurate) of the actual duration for which it will execute. We then propose (Sec. 2.2) a baseline prediction-oblivious algorithm that is fairly straightforward – what a real-time systems developer would probably come up with if no prediction of the actual execution duration had been provided. Next (in Sec. 2.3) we explain how the prediction can be incorporated to obtain an algorithm that is more energy-efficient than the prediction-oblivious one when the prediction is more-or-less correct, whilst consuming not much more energy than the prediction-oblivious algorithm even when the prediction is completely off the mark. We then (Sec. 2.4) address the issue of learning better predictions during run-time, and finally (Sec. 2.5) apply these concepts upon a simple example system. We close out the section by discussing related work and listing some open questions (Sec. 2.6).

2.1 System model

Since our primary purpose is to illustrate the algorithms using predictions framework, we will make a number of simplifying assumptions in both our workload model and our energy model for the purposes of ease of presentation (and so that we can focus on emphasizing the use of prediction in the algorithms we develop). We will point out, where relevant, how our results are easily modified to account for the factors that we abstract away in our simplifications.

We assume that we have a single job that is released at time zero, has a hard deadline at time D, and is characterized by a worst-case execution time parameter (WCET) W, that is to be executed upon a uniprocessor platform. The work done by the processor over any time-interval $[t_1, t_2]$ is equal to $\left(\int_{t_1}^{t_2} s(t)\, dt\right)$, where $s(t)$ denotes the processor speed at time-instant t. The speed may be changed by the run-time scheduler at any instant to take on any non-negative real value; there is no cost or penalty associated with changing the

Table 1 Summarizing the system model: $\langle W, D, P, \alpha, \gamma_{\text{ub}} \rangle$.

symbol	interpretation
W	worst-case execution time
D	deadline
P	predicted execution time
α	exponent of power function (power = speed$^\alpha$)
γ_{ub}	robustness bound (maximum ratio of energy consumed)

speed. The *power* required to run the processor at some speed s is equal to s^α, where α is a positive real value that is larger than 1.[3] Hence, the *energy* consumed by the processor over any time-interval $[t_1, t_2]$ is given by $\left(\int_{t_1}^{t_2} \left(s(t)\right)^\alpha dt\right)$.

A predicted execution duration. As is widely known in the real-time systems community, in safety-critical systems the values that are assigned to WCET parameters tend to be conservative over-approximations; hence it is unlikely that our example job will actually need to execute for a duration W under the overwhelming majority of run-time circumstances that arise in practice. Let us assume that we additionally have a prediction, P, on the *actual execution time* (AET) of the job, where P is a non-negative real-valued number that is no larger than the WCET W.

Recall that our goal is to develop an algorithm that performs very well when the prediction is correct, while simultaneously guaranteeing to not perform too poorly even when the prediction is incorrect. We assume that we are provided with an additional parameter that is a real number > 1, γ_{ub}, specifying the maximum multiplicative factor by which the energy consumed by an algorithm using predictions may exceed the energy consumed by a reasonable prediction-oblivious algorithm that ignores the prediction (which we will describe in Sec. 2.2), regardless of how poor the prediction turns out to be.

Generalizing the model. As stated earlier, we are assuming a very simple system model in order to keep the focus on the algorithms using predictions framework. Here, we briefly discuss possible extensions to our model that render it more realistic.

- We are considering the problem of scheduling a single job within a specified time-interval $[0, D]$. Such problems could arise in application systems where a more complex workload is being scheduled non-preemptively, with the non-preemptive schedule generated using heuristic tools that seek to maximize the *slack* for each job. Energy-efficient implementation of the schedule so generated would require us to schedule each job within a specified time-interval, yielding (for each job) the problem addressed in our model.
- Our energy model (that the power needed to execute the processor at speed s is equal to s^α) is idealized in that it ignores, e.g., leakage energy. But as we will see later in this section, the methodology we develop allows for "black-box" models to be substituted for this simple energy model – the overall methodology remains unchanged, the only difference being that numerical computation (rather than the solving of closed-form expressions) may be needed.
- In a similar vein, adding the restriction that the processor speed may only take on one of a few pre-specified permissible values does not change the essential structure of our methodology (although the combinatorial explosion resulting from having to choose from amongst a set of discrete speeds may slow down the running times).

[3] Observe that energy-efficient scheduling is trivial if $\alpha \leq 1$: it is optimal to simply run the processor at maximum speed until the job completes.

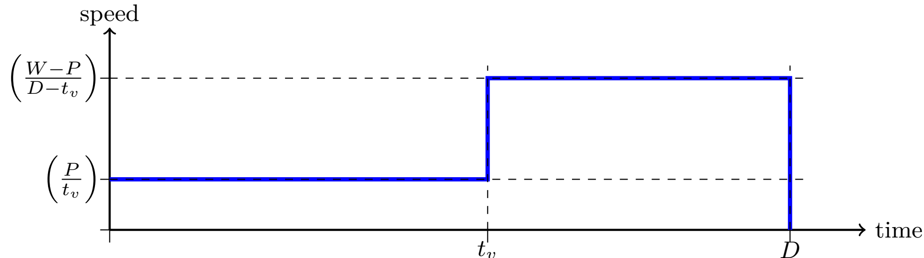


Figure 1 Speed profile for Algorithm ALG (Sec. 2.3): A job is executed according to this profile until completion (upon which the speed is set to zero).

2.2 A Prediction-Oblivious Algorithm

If we were to choose to not make use of the prediction, a reasonable run-time algorithm would be to set the processor speed to (W/D) and then execute the job until completion. (This follows from the well-understood phenomenon – see, e.g., [17] – that for $\alpha > 1$, the energy required to complete a known amount of work is minimized by having the CPU run at a constant speed for the duration of execution of the work. Thus, setting the processor speed equal to W/D would result in the minimum energy consumption in the worst case – i.e., when the job requires maximum execution.) Since this algorithm has the least worst-case energy consumption, let us refer to it as WC. Note that the duration for which WC would execute a job whose realized (i.e., actual) execution time turns out to be equal to A is $\frac{A}{\text{speed}} = \frac{A}{W/D} = \frac{AD}{W}$ time units. Hence, the energy consumed in executing a job with actual execution time A, denoted $E_{\text{WC}}(A)$, is given by the following expression.

$$E_{\text{WC}}(A) \stackrel{\text{def}}{=} \left[\left(\frac{W}{D}\right)^{\alpha} \times \left(\frac{AD}{W}\right)\right] = \left[\left(\frac{W}{D}\right)^{\alpha} \times A \cdot \left(\frac{D}{W}\right)\right] = \left(\frac{W}{D}\right)^{\alpha-1} \times A \qquad (1)$$

2.3 Incorporating The Prediction

We now develop an algorithm, ALG, that makes use of the predicted value P of the actual execution time of the job. Recall that the goal in so doing is to have ALG consume less energy than WC if the prediction is correct, i.e., if the actual execution time happens to be equal (or close) to P, while consuming no more than γ_{ub} times the amount of energy consumed by WC even if the prediction is completely off.

Our prediction-oblivious algorithm of Section 2.2 had set the processor speed to equal W/D. Observe that it is unsafe for ALG to analogously replace the WCET W by the predicted execution duration P and simply set the speed equal to P/D, since it would then have completed exactly P units of execution by the deadline D. Hence if the prediction turns out to be an underestimation and the actual execution time exceeds P, ALG would cause a deadline miss to occur. Therefore ALG instead sets itself a *virtual deadline* t_v that is strictly before the actual deadline at D (i.e., $t_v < D$ – see Figure 1), and seeks to ensure that the job will complete at this virtual deadline if the prediction is correct (i.e., the job needs exactly P units of execution). Analogously to Algorithm WC in Section 2.2, this goal is achieved by setting the initial processor speed equal to exactly P/t_v. If during some execution the job completes prior to time-instant t_v, then it follows, by a reasoning analogous to that used in deriving Expression 1, that the energy consumed in ALG is equal to $\left((P/t_v)^{\alpha-1} \times A\right)$, where A again denotes the actual execution time (the AET) of the job on this particular execution. If however the job has not completed execution by its virtual deadline, ALG

concludes that the prediction is incorrect and adjusts processor speed to ensure that the maximum amount of execution that could remain, $(W-P)$, will complete within the duration remaining prior to the actual deadline, $(D - t_v)$; it does this by increasing the processor speed to equal $(W - P)/(D - t_v)$. In this case the energy consumed over the interval $[0, t_v]$ equals $\Big((P/t_v)^{\alpha-1} \times P\Big)$ (since P units of work was done during this interval), while $(A - P)$ units of work is done for the remainder of the execution and so the energy consumed between time-instant t_v and the job's completion is equal to $\Big(((W - P)/(D - t_v))^{\alpha-1} \times (A - P)\Big)$. Putting the pieces together, we conclude that the energy consumed by ALG on executing an invocation of the job that has AET A is given by

$$E_{\text{ALG}}(A) = \begin{cases} \left(\frac{P}{t_v}\right)^{\alpha-1} \times A, & \text{if } A \leq P \\ \left(\frac{P}{t_v}\right)^{\alpha-1} \times P + \left(\frac{W-P}{D-t_v}\right)^{\alpha-1} \times (A-P), & \text{otherwise} \end{cases} \tag{2}$$

It remains to specify how the value of t_v is to be determined. Informally speaking, the slower the processor speed the less the energy consumed. Since our goal in designing algorithms that use predictions is to have excellent performance (in this example, minimal energy consumption) in the event that the prediction is correct, we would like the initial processor speed to be as small as possible – this is achieved by setting t_v to be as large as we possibly can. (Note from Expression 2 that $E_{\text{ALG}}(P)$ is smaller for larger t_v.) However in the event of the prediction turning out to be *in*correct, there should be adequate time remaining between the virtual deadline and the actual one to allow the job to be completed by increasing the processor speed. We have seen that increasing the processor speed requires an exponential increase in the power needed (and hence the energy consumed) – we must ensure that the total amount of energy consumed when the prediction is incorrect does not break the specified robustness bound (i.e., does not exceed γ_{ub} times the amount of energy that would be consumed by Algorithm WC to execute the same job). It is evident that since ALG executes at a greater speed (and hence less energy-efficiently) than WC for $A > P$, the ratio of $E_{\text{ALG}}(A)/E_{\text{WC}}(A)$ *increases* with increasing A, taking its maximal value when the job actually executes for a duration equal to its WCET (i.e., A takes on its largest possible value of W). The virtual deadline is therefore assigned the largest value of t for which

$$\begin{aligned} & E_{\text{ALG}}(W) \leq \gamma_{\text{ub}} \times E_{\text{WC}}(W) \\ \equiv \quad & \left(\frac{P}{t}\right)^{\alpha-1} \times P + \left(\frac{W-P}{D-t}\right)^{\alpha-1} \times (W-P) \leq \gamma_{\text{ub}} \times \left(\frac{W}{D}\right)^{\alpha-1} \times W \end{aligned} \tag{3}$$

Computing the virtual deadline t_v. We now discuss how the largest value of t satisfying Expression 3 is determined. For the special case where α, the exponent defining the relationship between the processor's speed and its power consumption, is equal to two (i.e., power equals speed-squared), Expression 3 may be algebraically rewritten as the quadratic inequality

$$\gamma_{\text{ub}} \times t^2 - \overbrace{\left((\gamma_{\text{ub}} - 1) \times D + \frac{2PD}{W}\right)}^{-B} \times t + \overbrace{\left(\frac{PD}{W}\right)^2}^{C} \leq 0 \tag{4}$$

and hence, using the well-known formula for finding the roots of a quadratic equation, we get the following closed-form expression for t_v (with B and C denoting the expressions indicated in Eqn 4 above):

$$t_v = \frac{-B + \sqrt{B^2 - 4\gamma_{\text{ub}} C}}{2\gamma_{\text{ub}}} \tag{5}$$

For values of α other than two, we do not have a closed-form expression for t_v Instead we can obtain, via numerical methods, an arbitrarily close approximation of the true value of t_v in the following manner.

1. Since ALG should have its initial speed be no larger than the worst-case algorithm WC's initial speed, we should have $(P/t_v) \leq (W/D)$ or $t_v \geq (P/W) \times D$. This defines a lower bound on the value of t_v; D is obviously an upper bound.
2. Within this range, observe the monotonicity property of t satisfying Expression 3: if t_o satisfies Expression 3 than so do all $t < t_o$; if t_o does not satisfy Expression 3 then neither does any $t > t_o$.
3. We can therefore do binary search between $\big((P/W) \times D\big)$ and D to determine an arbitrarily close approximation of the largest value of t satisfying Expression 3.

We have assumed here the simplified power model: (power = speed$^\alpha$) to derive the value to be assigned to t_v. We point out that more complex models may be used – simply replace the one occurrence in the derivation of Eqn. 1 and the two occurrences in the derivation of Eqn. 2 of (speed$^\alpha$) by calls to a black-box implementation of the appropriate power function.

2.4 Learning an Improved Prediction

Thus far, we have assumed that our goal is to schedule the execution of a single job in an energy-efficient manner. It may be the case for certain application scenarios that this single job is invoked repeatedly in a periodic or sporadic manner. In such applications it may make sense to update the predicted execution duration (the parameter P) based on the observed actual execution times of previously-executed invocations. (Standard techniques, such as a PID controller or some appropriate Reinforcement Learning [15] method, may be used for this purpose.) But observe that P appears in Expression 3 which is used in computing the virtual deadline; hence, updating the value of P will require that the value assigned to the virtual deadline parameter t_v also be updated. That is, if the value of P is not a priori fixed then the value assigned to the virtual deadline can be looked upon as a function of P – this dependence of the virtual deadline on the value of P may be made explicit by referring to the virtual deadline as a function of P: $t_v(P)$. Computing $t_v(P)$ every time the value of P is updated may be computationally too expensive to do during run-time; hence we propose to precompute the values of $t_v(P)$ within a reasonable range of values for P, and store these values (alongwith the corresponding values of the speeds at which the processor should be run before and after t_v) in a lookup table for use during run-time.

2.5 An Example

We will now illustrate the concepts introduced above upon an example problem instance $\langle W, D, P, \alpha, \gamma_{\text{ub}}\rangle = \langle 8, 10, 5, 2, 1.1\rangle$. That is, we have a job with WCET $W = 8$, relative deadline $D = 10$, and a predicted execution duration $P = 5$, that is to be executed upon a processor with the power exponent $\alpha = 2$ (recall that this means that executing the processor at speed s requires power s^α). A robustness bound $\gamma_{\text{ub}} = 1.1$ is also specified: in using the prediction we may not exceed 1.1 times the energy consumed by the prediction-oblivious algorithm WC regardless of how inaccurate the prediction turns out to be.

Computing the virtual deadline. Since α happens to have value 2 in our example instance, we may solve for t_v exactly using Equation 5 to compute that $t_v = \mathbf{7.6}$.

Comparing the two algorithms. We have plotted the ratio of the energy that is consumed by the algorithm using predictions to that consumed by the prediction-oblivious algorithm in Figure 2. This plot reveals that ALG consumes approximately 82.2% of the energy that WC

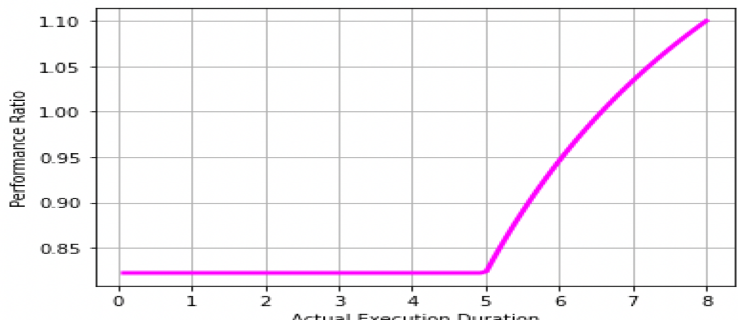


Figure 2 Ratio of energy consumed by ALG to the energy consumed by WC as a function of the actual execution time (AET), as AET ranges from 0 to $W = 8$.

consumes when the AET of the job is at (or below) its predicted execution duration. Thus, in this example *a perfect prediction results in an approximately* 17.8% *savings in consumed energy.* This same level of savings is also obtained with predictions that over-estimate the AET (i.e., when AET $< P$). And while the relative savings decreases as the AET increases beyond the predicted value, notice that the energy consumed by ALG remains below that consumed by WC as long as the AET is no larger than ≈ 6.575. Algorithm WC is more energy-efficient for values of AET that fall in the range $(6.575, 8]$; note, however, that the ratio never exceeds the specified bound of $\gamma_{\text{ub}} = 1.1$.

Updating the prediction. For application scenarios where the same job is invoked repeatedly at run-time, we had discussed in Section 2.4 how one could use on-line learning to *update* the value of the prediction P. We had stated that the value of the virtual deadline parameter t_v changes as P changes – Figure 3 depicts t_v as a function of P for our example. Prior to run-time a table of values for t_v as a function of P should be pre-computed and stored for use during run-time.

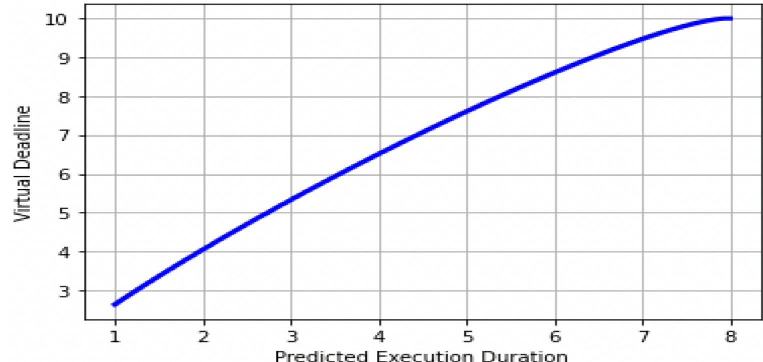


Figure 3 The virtual deadline $t_v(P)$ as a function of the predicted execution duration P.

2.6 Related Work and Future Directions

Although (as mentioned earlier) there is an extensive body of work dealing with energy-efficient scheduling, most of it does not consider predictions. We found only three papers [2, 3, 10] dealing with energy from amongst the 28 papers tagged with the keyword "scheduling" in the list of publications maintained on the Algorithms With Predictions web-site[4] ("energy" is not a searchable keyword there). In contrast to our model where we have just a single job, these papers all allow for multiple jobs; however they make simplifying assumptions regarding AETs that appear unrealizable in practice. Antoniadis et al. [2] assume that the exact execution duration of a job is known beforehand (and it is the arrival times and deadlines that are unknown and the subject of prediction), while Bamas et al. [3] assume that a job's AET becomes known at the instant of job arrival. These assumptions do not model the reality of the vast majority of embedded systems where the AET only becomes known by actually executing a job to completion. Lee et al. [10] consider a very different problem (that does not involve deadlines): minimize energy cost in data centres where the cost of energy depends upon both the volume and the peak usage, and where energy can be either locally generated or drawn from the grid.

Future work. In our opinion, the most obvious generalization of our model would be to allow for the specification of multiple jobs (eventually, even recurrent tasks). However exploiting predictions for energy-efficient scheduling becomes considerably more challenging when multiple jobs are to be considered – the interested reader is encouraged to glance through [3] to get a flavor for the complexities that arise even under the unrealistic assumption that predictions of a job's AET are revealed to hold or not upon job arrival (despite the other simplifying assumptions there, such as that all jobs have the same relative deadline, processor sharing is permitted, etc.).

3 Algorithms Using Predictions: The Foundations

The idea of "better than worst case" analysis has recently enjoyed much attention in theoretical computer science [14]. The motivation is that worst case instances are not common, and so we should design algorithms to also perform well in the common case while simultaneously maintaining worst case guarantees. One approach is to use predictions to guide an algorithm – these predictions may be generated via machine learning, human intuition, or other low assurance methods. Such predictions may be incorrect, therefore algorithms should not trust them entirely. The goal is to get the best of both worlds by improving performance when the prediction holds, without degrading performance by too much when the prediction fails. More specifically, these algorithms are characterized by three properties (formally defined a bit later in this section):

1. *Consistency* (Definition 4): When the predictions are correct, the algorithm provides very good performance for the instance.
2. *Robustness* (Definition 3): When the predictions are incorrect, the algorithm is not much worse than a good algorithm that does not use predictions.
3. *Smoothness* (Definition 5): The algorithm's performance degrades smoothly with the error in the prediction.

[4] `https://algorithms-with-predictions.github.io` (Accessed on 25/02/2023.)

In other words, the *consistency* of an algorithm using predictions characterizes its performance when the predictor is perfectly accurate, and its *robustness* its performance guarantee regardless of the quality of the predictions.

▶ **Example 1.** Let us revisit the example application of Sec. 2 – Fig 2 denotes the performance ratio of the algorithm using predictions, ALG, to that of the prediction-oblivious algorithm WC. Note that the performance ratio is ≈ 0.822 when the prediction is exact – this is a quantitative measure of ALG's consistency. ALG's robustness follows from the observation that the performance ratio never exceeds the specified bound of $\gamma_{\text{ub}} = 1.1$. ALG's smoothness property is visually evident from Fig 2: the performance ratio remains ≈ 0.822 for AETs $\leq$ prediction, and degrades smoothly with increasing error for AETs $>$ the prediction. ◀

The standard manner of characterizing the performance of any algorithm (not just those using predictions) designed to operate in an online setting is via *competitive analysis.* Consider some relevant performance objective that we want to minimize [5] – in the example in Section 2, this metric is energy.

▶ **Definition 2** (Competitive Ratio). *For any input X, let* OPT(X) *denote the cost achieved by an optimal clairvoyant algorithm and let* ALG(X) *denote the cost of the online algorithm* ALG *on this input. The competitive ratio of* ALG *is the ratio* (ALG(X)/OPT(X)) *maximized over all possible inputs X.*

One can think of the algorithms using predictions framework as a generalization of competitive analysis. In this framework, the online algorithm gets a prediction, say P. This prediction should be something that allows the algorithm to make better decisions than it would be able to in the absence of this prediction – for instance, in the example of Sec. 2, the prediction is the actual execution time (AET) of the job. We want the algorithm to perform well even with incorrect predictions; therefore, robustness is simply the competitive ratio of an algorithm which uses predictions.

▶ **Definition 3** (Robustness). *Given an online input X and prediction P for X, let* ALG(X, P) *denote the cost of the online algorithm using this prediction and* OPT(X) *the cost of the optimal clairvoyant algorithm.* ALG*'s robustness is defined as the ratio* (ALG(X, P)/OPT(X)) *maximized over all values of X and P.*

The algorithms using predictions framework seeks to ensure that robustness of an algorithm is as close as possible to the best competitive ratio that can be achieved by any online algorithm for the problem.

Informally, *consistency* is the competitive ratio of the algorithm when the prediction is accurate – it measures how well the algorithm uses predictions to improve performance. In particular,

▶ **Definition 4** (Consistency). *Given an online input X and an exact prediction P for X, let* ALG(X, P) *denote the cost of the online algorithm using this prediction and* OPT(X) *the cost of the optimal clairvoyant algorithm. The consistency for the algorithm is defined as the ratio* ALG(X, P)/OPT(X) *maximized over all values of X assuming P exactly matches the predicted value.*

[5] We can define this for maximization objectives in a similar manner.

There typically is a trade-off between consistency and robustness since one can think of robustness as hedging against an inaccurate prediction. In the energy example of Section 2, one can adjust this trade-off by making different decisions about the virtual deadline – a later virtual deadline may improve consistency while potentially being less robust and vice versa.

In addition to consistency and robustness, we want the algorithms to be able to gracefully handle small errors in predictions – this property is called *smoothness*. Smoothness is harder to define precisely and generally. Given an input X, say the perfect prediction for X is $\mathcal{P}_X$ – this is the prediction which one would generate if we knew X precisely. Say the actual prediction given the algorithm $\textsc{alg}$ is P. The error $\textsc{error}$ is some function of $\mathcal{P}_X$ and P – this function can be problem and algorithm specific, but informally is some difference function between the two parameters which is 0 when $P = \mathcal{P}_X$ and increases as the difference between the two increases. Ideally, we want algorithms whose competitive ratio relative to the optimal increases slowly as the error increases.

▶ **Definition 5** (Smoothness). *Given an online input X and a prediction P for X, let $\textsc{alg}(X, P)$ denote the cost of the online algorithm using this prediction and $\textsc{opt}(X)$ the cost of the optimal clairvoyant algorithm Let $\textsc{error}$ denote the error between the prediction P and the perfect prediction $\mathcal{P}_x$ for X. We say that the algorithm is $f(\textsc{error})$-smooth if $\textsc{alg}(X, P)/\textsc{opt}(X) \leq \beta + f(\textsc{error})$ for all $\textsc{error}$ and all X.*

We want the smoothness function $f(\textsc{error})$ to grow smoothly as $\textsc{error}$ moves away from 0. Therefore, smoothness is simply a generalization of consistency – most definitions of consistency are also in terms of the competitive ratio when the error is 0.

Adapting the framework. Above we have described the algorithms using predictions framework as it is currently defined (and used) by the optimization algorithms community. We now briefly discuss two ways in which this framework needs adaptation in order to render it more suitable for safety-critical systems.

1. Whereas the framework focuses on the trade-off between robustness and consistency, *we look upon robustness as a constraint* that must be met, with consistency (and smoothness) being metrics that we can optimize for/ trade off amongst. Although this is in one sense merely a restriction of the algorithms using predictions framework, both our examples (in Sec. 2 and the upcoming one in Sec. 4) show that this restriction fundamentally changes the process of applying the framework to specific problems.
2. The framework, as currently defined, primarily compares the performance of $\textsc{alg}$ with that of an optimal clairvoyant algorithm. But such comparison may not always be appropriate: consider, for instance, our energy-efficient scheduling application of Sec. 2. It is evident that no on-line algorithm can have a competitive ratio (Definition 2) smaller than infinity[6], and hence the framework as currently defined is not applicable to this application. But we saw in Sec. 2 that applying our adaptation, which *compares the performance of* $\textsc{alg}$ *with that of a reasonable prediction-oblivious algorithm*, yields significant energy savings.

[6] To see this, let s_o denote the initial speed at which $\textsc{alg}$ runs the processor; for very (small) AET A, it consumes $\left(s_o^{\alpha-1} \times A\right)$ units of energy. Meanwhile the clairvoyant $\textsc{opt}$, knowing the value of A, would execute at speed (A/D) and so consume $\left((A/D)^{\alpha-1} \times A\right)$ units of energy. The competitive ratio is thus $\left(\frac{s_o D}{A}\right)^{\alpha-1}$, which, for any fixed $s_o > 0$ (i.e., any initial speed chosen by $\textsc{alg}$), approaches ∞ as $A \to 0$.

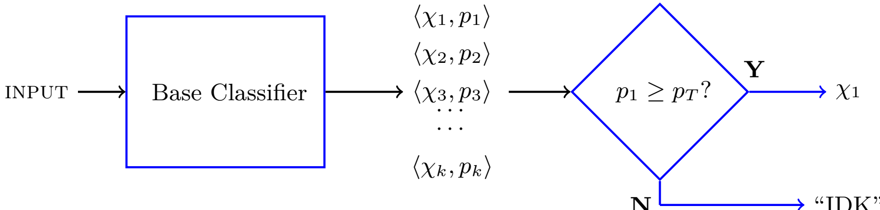


Figure 4 Obtaining an IDK classifier from a base classifier. For a given input, the base classifier outputs up to k ordered pairs $\langle \chi_i, p_i \rangle$, indicating that it believes that the input belongs to the class χ_i with confidence score p_i. (It is assumed that $p_1 \geq p_2 \geq \cdots \geq p_k$, i.e., $\chi_1, \chi_2, \ldots, \chi_k$ are the k most likely classes, in order.) The threshold confidence parameter for the IDK classifier is set at p_T.

We point out that while the first adaption above can be looked upon as a restriction of the algorithms using predictions framework, the second adaptation *expands* the typical use-case for this framework and thereby enables its use in situations where the original framework is not directly applicable. This expanded applicability was in evidence in Sec. 2; next in Sec. 4, we will however not make use of this second adaptation, instead going back to the original framework and comparing with an optimal clairvoyant algorithm.

4 Illustrative Example: IDK Classifiers

A classifier is a software component that categorizes each input provided to it into one of a fixed set of classes. IDK classifiers, also known as classifiers that *defer* [7, 8, 12], are obtained from base classifiers that use deep learning and related AI technologies in the following manner (see Figure 4): if the base classifier is unable to arrive at a classification decision at an adequately high level of confidence, then a dummy class, IDK (for "I Don't Know") is output instead. In scenarios where classification is safety-critical, IDK classifiers should be used in conjunction with more traditional ("deterministic") classifiers for the same classification problem: if an IDK classifier outputs IDK upon some input, then the deterministic classifier is called upon that input to provide an authoritative classification. (Of course, it is only meaningful to use IDK classifiers in this manner if doing so is more efficient, in some sense or the other, than directly calling the deterministic classifier in the first place.)

As part of ongoing efforts to provide a scheduling-theoretic framework that allows for the use of LECs in hard-real-time safety-critical systems, the real-time scheduling theory community has recently (see, e.g., [1, 5, 6]) begun studying IDK classifiers. The three cited papers [1, 5, 6] consider variants of this problem: *given one or more IDK classifiers and a deterministic classifier for the same classification problem, how does one determine which of the classifiers to execute, and in what order, to minimize the expected duration needed to obtain a successful classification (perhaps within a specified hard deadline)?*

Abdelzaher et al. [1] describe how the training phase of IDK classifiers should be modified in order to estimate the probabilities that they will succeed in classifying an arbitrary input.[7] In this paper, we look upon the probability values so obtained as predictions of the true (unknown – perhaps unknowable) underlying probabilities, and apply the algorithms using predictions paradigm to deal with the possibility that these predictions are incorrect.

[7] Dependencies amongst classifiers can also be estimated by the method in [1]; e.g., what is the conditional probability that IDK classifier K_i returns "IDK" upon inputs for which classifier K_j has returned "IDK"?

The specific problem considered. Analogously to our approach in Section 2, we consider here a simplified version of the problem studied in [1, 5, 6] upon which to apply our adaptation of the algorithms using predictions framework. Specifically, we assume that we have available to us a single IDK classifier and a single deterministic classifier for the same classification problem, and must decide whether to (i) directly execute the deterministic classifier; or (ii) first call the IDK classifier, subsequently calling the deterministic classifier in the event that the IDK classifier returns "IDK." Our objective is to minimize the expected execution duration – i.e., the duration needed to successfully classify the input. If the probability of the IDK classifier returning a successful classification were a priori known, this problem would be trivial to solve: call the IDK classifier first if and only if its execution duration, plus the product of the probability it returns "IDK" times the execution duration of the deterministic classifier, is strictly smaller than the execution duration of the deterministic classifier. As stated above we assume, however, that we only have a predicted, rather than the actual, value of this probability. Specifically, let us suppose that the available pair of classifiers is characterized as a three-tuple $I \stackrel{\text{def}}{=} \langle C, \Pi, D\rangle$ where

- C is the execution duration[8] of the IDK classifier, and D is the execution duration of the deterministic classifier. (It is a reasonable assumption that $C < D$, since if $C \geq D$ there is no benefit, from the perspective of minimizing the expected execution duration, to executing the IDK classifier.)
- Π is the *predicted* probability that the IDK classifier will succeed (i.e., $(1 - \Pi)$ is the predicted probability that it will output "IDK").
  We point out that *the "real" probability that the IDK classifier will succeed, which we will denote as P, is unknown* to a non-clairvoyant algorithm. That is, Π is a prediction of the unknown (perhaps unknowable) value of P.

Given such a three-tuple $I \stackrel{\text{def}}{=} \langle C, \Pi, D\rangle$, an algorithm must decide whether to call the IDK classifier first (and call the deterministic classifier only if this returns "IDK"), or to directly call the deterministic classifier. The **performance metric** of an algorithm that makes this decision is the expected duration of its classification decision:

$$\begin{cases} C + (1 - P) \times D, & \text{if the algorithm calls the IDK classifier first} \\ D, & \text{if the algorithm directly calls the deterministic classifier} \end{cases}$$

The objective is to obtain a classification with minimum expected duration. We define the **prediction error** (see Sec. 3, the paragraphs between Definitions 4 and 5) to be the difference between the actual and the predicted success probabilities:

$$\text{ERROR}(I) \stackrel{\text{def}}{=} |\Pi - P| \tag{6}$$

A problem instance. As we have previously pointed out, in safety-critical systems it makes sense to place a bound on the "poorness" of acceptable solutions – in the formalism of algorithms using predictions as laid out in Sec. 3, to specify a bound on the robustness of the algorithm. Hence we specify a problem instance as (I, γ_{ub}) where I denotes the pair of classifiers: $I = \langle C, \Pi, D\rangle$, and γ_{ub} denotes the maximum acceptable robustness, defined here as the ratio of the expected execution duration for the algorithm using predictions

[8] For this application (in contrast to the one in Sec. 2), we make the simplifying assumption that the given parameters C and D represent the actual execution times (AETs) of the classifiers – this allows us to focus on LEC-specific aspects of prediction without getting bogged down revisiting the kinds of issues explored in Sec 2. The model is easily generalized to incorporate uncertainty in (and predictions of) the values of these AET's.

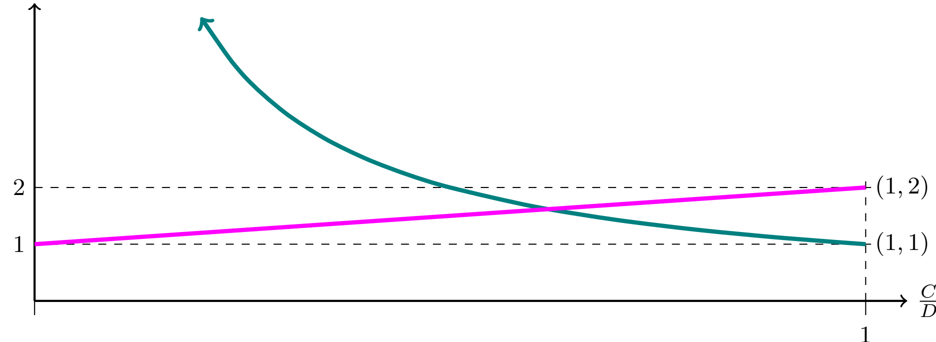


Figure 5 Bound on ratio of expected execution durations as a function of C/D. The straight line: use the IDK classifier. The curve: do not use the IDK classifier.

to the expected execution duration for an optimal algorithm (denoted OPT) that knows the value of P. Note that in contrast to the energy-efficient scheduling example of Sec. 2 where we had compared its performance of an algorithm designed to use predictions against the "reasonable" prediction-oblivious algorithm that we specified in Sec. 2.2, here we are comparing against an idealized clairvoyant optimal algorithm in OPT.

4.1 An Algorithm Using the Prediction

We will now design an algorithm, ALG (see Algorithm 1), that uses the predicted value Π to decide whether to directly execute the deterministic classifier, or to execute the IDK classifier first and the deterministic one only if the IDK classifier fails. We will first explore the constraints placed on ALG due to the robustness requirement: that its performance (expected duration) never exceed that of OPT by more than a factor γ_{ub}. Once we have identified the space of robust designs, we will next see how consistency considerations direct us to a choice within this design space; finally, we will characterize the smoothness characteristics of this choice.

Meeting the robustness constraint. ALG must make one of the two available choices: execute the IDK classifier first, or directly execute the deterministic classifier. Let us study the implications upon robustness of each choice.

1. If ALG executes the IDK classifier first, its expected duration is $C + (1 - P) \times D$. In the worst-case scenario for ALG, the value of P turns out to be equal to zero and OPT, knowing this, would directly execute the deterministic classifier for a duration of D. The performance ratio is therefore equal to $\left(1 + \frac{C}{D}\right)$ – this is plotted as a function of (C/D) in **magenta** as the increasing straight line in Fig. 5.
2. If ALG directly executes the deterministic classifier, then its duration is D. In the worst-case scenario for ALG, $P = 1$ and hence OPT would execute the IDK classifier for a duration of C. The performance ratio is therefore equal to $\left(\frac{D}{C}\right)$ – this is plotted as a function of (C/D) in **teal** as the decreasing curve in Fig. 5.

The plots of Figure 5 help us identify the space of acceptable designs for any given problem instance $(I = \langle C, \Pi, D \rangle, \gamma_{\text{ub}})$. For this instance, locate the position of the point with x-coordinate (C/D) and y-coordinate γ_{ub} in Fig. 5.

1. If this point lies below both the curves, then neither choice can guarantee to meet the specified robustness bound and hence we must **declare failure**.
2. If this point lies between the two curves, the choice of whether to directly call the deterministic classifier or to call the IDK classifier first is forced on ALG by the robustness bound. If the point lies to the left of the intersection of the two curves, then only calling the IDK classifier first can guarantee the robustness bound whereas if it lies to the right of the intersection, then only directly calling the deterministic classifier can guarantee the robustness bound.
3. However if this point lies above both curves, then either choice meets the robustness bound. *This is the only situation where an algorithm is free to use the prediction*; below we will discuss how it does so and makes a choice to achieve consistency.

Let $(\hat{x}, \hat{y})$ denote the point where the two curves in Fig. 5 intersect. We have $1+\hat{x} = 1/\hat{x}$ and $\hat{y} = 1/\hat{x}$, from which it follows that $(\hat{x}, \hat{y}) = (1/\varphi, \varphi)$ where φ denotes the famous constant commonly called the Golden Ratio: $\varphi = (1+\sqrt{5})/2 \approx 1.618$. Theorem 6 follows:

▶ **Theorem 6.** *(i)* ALG *never returns failure for problem instances with $\gamma_{\text{ub}} \geq \varphi$; and (ii) the prediction is never useful for instances with $\gamma_{\text{ub}} \leq \varphi$, where φ denotes the Golden Ratio: $\varphi = (1+\sqrt{5})/2$.* ◂

Achieving Consistency. As we have observed above, the prediction is meaningful only for systems $(I = \langle C, \Pi, D \rangle, \gamma_{\text{ub}})$ for which the point $(C/D, \gamma_{\text{ub}})$ lies above both curves in Fig. 5. Recall (Definition 4) that consistency characterizes the performance of an algorithm using predictions when the prediction is exactly correct; for our example, that translates to the case when $\Pi = P$. If $\Pi = P$, it is evident that ALG should use the IDK classifier iff

$$\Big(C + (1-\Pi)\cdot D < D\Big) \Leftrightarrow \Big(C + D - \Pi\, D < D\Big) \Leftrightarrow \Big(\boldsymbol{\Pi} > \frac{\boldsymbol{C}}{\boldsymbol{D}}\Big)$$

An ALG that executes the IDK classifier iff $\big(\Pi > (C/D)\big)$ clearly has consistency equal to one: when the prediction is exact, it has exactly the same expected duration to successful classification as OPT does.

Smoothness Analysis. We now analyze the smooth properties (Definition 5) of the algorithm obtained above (for convenience, also listed in pseudo-code form in Algorithm 1). Since the predictions are only usable when $\gamma_{\text{ub}} > \max(D/C, (1 + C/D))$, we restrict our analysis of smoothness to instances satisfying this property. In Figure 6, we plot the expected duration of both ALG (in blue) and OPT (in red) as a function of the true probability of successful classification by the IDK classifier, P, for both the cases when ALG chooses to first execute the IDK classifier (top left), and when it directly executes the deterministic classifier (bottom left). Let us separately consider the two cases.

1. For ALG to choose to use the IDK classifier (top left in Fig. 6), it must be the case that $\Pi > C/D$. Now when P is also $\geq C/D$, ALG's performance is the same as OPT's. On the other hand when $P < C/D$, the performance ratio is given by

   $$\frac{C + (1-P)\times D}{D} = \frac{C}{D} + (1-P) = 1 + \left(\frac{C}{D} - P\right)$$

   Meanwhile, ERROR $= \Pi - P > \left(\frac{C}{D} - P\right)$; plugging into the above expression, we have performance ratio

   $$= 1 + \left(\frac{C}{D} - P\right) < 1 + \text{ERROR}$$

Algorithm 1 The algorithm using predictions ALG. (The prediction finds use in Lines 10-13.)

Input: (I, γ_{ub})

1 /*$I \stackrel{\text{def}}{=} \langle C, D, \Pi \rangle$; γ_{ub} is the maximum permissible robustness
2 **if** $\left(\gamma_{ub} < \min\left(1 + \frac{C}{D}, \frac{D}{C}\right)\right)$ **then** /*Below both lines in Figure 5
3   **return** *failure*
4 **if** $\left(\min(1 + \frac{C}{D}, \frac{D}{C}) \le \gamma_{ub} \le \max\left(1 + \frac{C}{D}, \frac{D}{C}\right)\right)$ **then** /*Between the lines in Figure 5
5   **if** $\left(1 + \frac{C}{D}\right) \le \frac{D}{C}$ **then**
6   **return** *execute the IDK classifier first*
7   **else**
8   **return** *execute the deterministic classifier*
9 **if** $\left(\gamma_{ub} > \max\left(1 + \frac{C}{D}, \frac{D}{C}\right)\right)$ **then** /*Above both lines in Figure 5
10   **if** $\left(\Pi > \frac{D}{C}\right)$ **then**
11   **return** *execute the IDK classifier first*
12   **else**
13   **return** *execute the deterministic classifier*

Observe that ALG's performance loss when compared to OPT is *one-sided* – there is no performance loss unless $P < \Pi$ (i.e., Π is an over-estimation of the true probability P). The top right plot of Fig 6 plots the worst-case relative performance as a function of error, as the error ranges over $[0, C/D]$.

2. We can similarly consider the situation where ALG decides against using the IDK classifier (bottom left in Fig. 6). It must then be the case that $\Pi \le C/D$. If P is also $\le C/D$, then the performance ratio is one. If $P > C/D$, then the performance ratio is given by

$$\frac{D}{C + (1-P) \times D} = \frac{1}{\left(\frac{C}{D}\right) + (1-P)} = \frac{1}{1 - \left(P - \frac{C}{D}\right)}$$

Meanwhile, $\text{ERROR} = P - \Pi > P - \left(\frac{C}{D}\right)$; plugging into the above expression, we have performance ratio

$$= \frac{1}{1 - \left(P - \frac{C}{D}\right)} < \left(\frac{1}{1 - \text{ERROR}}\right)$$

Once again, ALG's performance loss when compared to OPT is one-sided – there is no loss unless $P > \Pi$ (i.e., Π is an under-estimation of the true probability P). The bottom right plot of Fig 6 plots the worst-case relative performance as a function of error, as the error ranges over $[0, 1 - (C/D)]$.

By visual inspection of the right-hand plots in Fig. 6, it is clear that the relative performance of ALG does indeed degrade smoothly with error. (When ALG chooses to use the IDK classifier – top right in Fig. 6 – the performance degrades linearly with error, but that is not the case when ALG directly uses the deterministic classifier – bottom right in Fig. 6.)

4.2 Related Work and Future Directions (on IDK Classifiers)

While several papers [1, 5, 6] dealing with IDK classifiers have recently appeared in the real-time literature, we are not aware of any prior work that integrates predictions with IDK classifiers. We nevertheless consider the prior papers [1, 5, 6] to be very relevant to our work

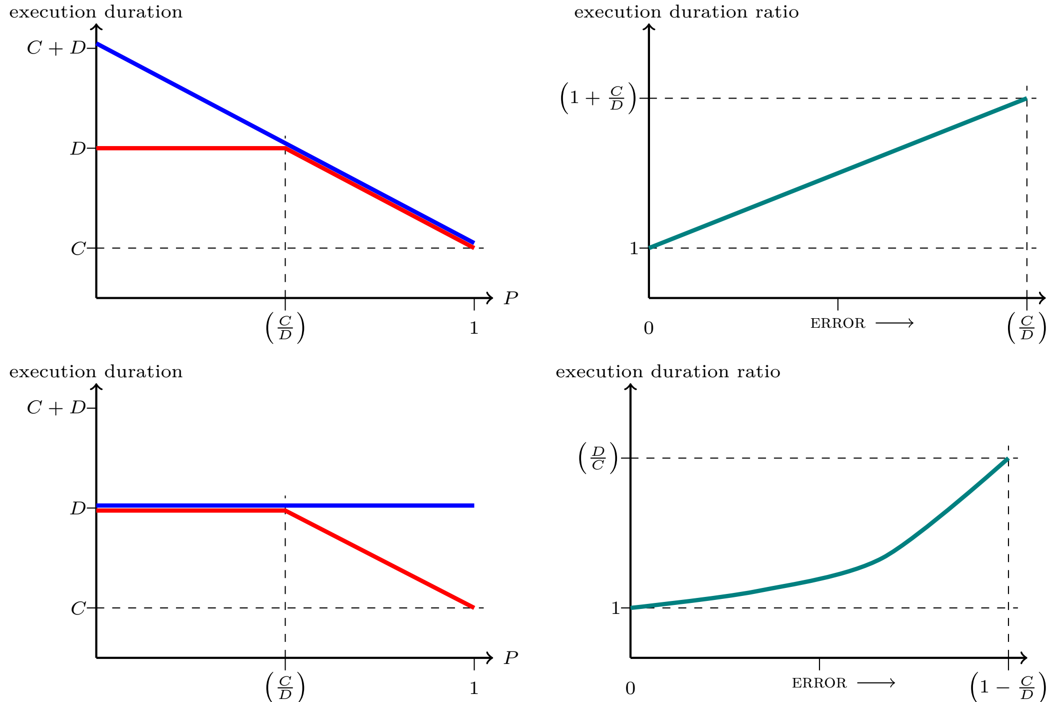


Figure 6 Smoothness. Blue is ALG; red is OPT. **Top**: ALG uses the IDK classifier (and so $\Pi > C/D$). **Bottom**: ALG does not use the IDK classifier ($\Pi \leq C/D$).

(and closely related to it) since they, Abdelzaher et al. [1] in particular, directly address the issue of obtaining a probabilistic characterization of run-time behavior of the classifiers by detailing procedures for determining the probability values – these are the "predictions" of our model. These papers also consider cascades of multiple classifiers (rather than just two, one IDK and one deterministic, as we have done), and show that one can in general obtain smaller expected duration to classification via such longer cascades. As future work, we plan to extend the methodology we have proposed in this manuscript to cascades of > 2 classifiers.

Another direction of future research involves incorporating learning. In our energy-efficient scheduling example of Sec. 2, we saw that if the job under consideration is repeatedly invoked and predictions can therefore be improved via a process of on-line learning, how doing so may allow for further efficiency gains (Sec. 2.4). But incorporating learning is not straightforward for IDK's even if the classification problem is repeatedly invoked: if $\Pi \leq C/D$, then ALG recommends against the use of the IDK classifier and we will never have a chance to update the prediction (the value of Π). Therefore some more sophisticated learning framework must be applied. We may, for example, consider using RL-based learning [15] to occasionally force ALG to call the IDK classifier, but for this we can only guarantee robustness over a sequence of classification decisions rather than on each individual classification.

5 Context and Conclusions

Sources of uncertainty abound in safety-critical real-time CPS's that operate within complex environments – these include traditional sources such as variation in run-time execution duration (the "WCET problem" [16]), and newer ones arising from the use of learning-enabled components in such systems. We believe that the recently proposed algorithm-design

paradigm of *Algorithms using Predictions* may be a good fit for dealing with such uncertainty, and should therefore be considered for use in safety-critical real-time systems – this manuscript reports on our efforts at doing so. Most prior work on algorithms using predictions dealt with pure optimization problems, and studied trade-offs between consistency (making good use of accurate predictions) and robustness (not suffering too much performance loss when predictions turn out to be incorrect). Since safety-critical systems are characterized by hard constraints, we had to adapt the framework and accord primacy to robustness: given an acceptable degree of robustness, identify a design space of acceptable algorithms and determine the algorithm offering maximum consistency within this space (and then characterize the smoothness of this algorithm). To demonstrate the usefulness of this framework, we have used it to design algorithms for a pair of applications that are of current interest to the real-time computing community; for both, we have shown that algorithms using predictions achieve improved performance when accurate predictions are available, without being penalized beyond pre-specified robustness bounds if the predictions go wrong.

As stated earlier, there are multiple sources of uncertainty in modern complex safety-critical real-time CPS's in addition to the two we have considered here. For each such source, the availability of good predictions could potentially be used to improve performance. For instance, in many event-triggered systems where triggering events may occur repeatedly, the *period* parameter [11, 13] specifies the minimum duration between successive triggerings. Current scheduling techniques usually allocate computational resources assuming that successive triggerings will occur at the maximum rate (i.e., separated by exactly the period parameter); if predictions are available of a larger inter-triggering duration, the algorithms using predictions framework could perhaps be used to design algorithms that make efficient use of computational resources when such predictions are correct, without catastrophic consequences if they turn out wrong.

References

1 Tarek Abdelzaher, Kunal Agrawal, Sanjoy Baruah, Alan Burns, Robert I. Davis, Zhishan Guo, and Yigong Hu. Scheduling IDK classifiers with arbitrary dependences to minimize the expected time to successful classification. *Real-Time Systems (to appear)*, 2023. URL: https://www-users.york.ac.uk/~rd17/papers/IDKarbitrary.pdf.

2 Antonios Antoniadis, Peyman Jabbarzade, and Golnoosh Shahkarami. A Novel Prediction Setup for Online Speed-Scaling. In Artur Czumaj and Qin Xin, editors, *18th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2022)*, volume 227 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:20, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SWAT.2022.9.

3 Etienne Bamas, Andreas Maggiori, Lars Rohwedder, and Ola Svensson. Learning augmented energy minimization via speed scaling. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 15350–15359. Curran Associates, Inc., 2020. URL: https://proceedings.neurips.cc/paper/2020/file/af94ed0d6f5acc95f97170e3685f16c0-Paper.pdf.

4 Mario Bambagini, Mauro Marinoni, Hakan Aydin, and Giorgio Buttazzo. Energy-aware scheduling for real-time systems: A survey. *ACM Trans. Embed. Comput. Syst.*, 15(1), January 2016. doi:10.1145/2808231.

5 Sanjoy Baruah, Alan Burns, Robert Davis, and Yue Wu. Optimally ordering IDK classifiers subject to deadlines. *Real Time Syst.*, 2022. doi:10.1007/s11241-022-09383-w.

6 Sanjoy Baruah, Alan Burns, and Yue Wu. Optimal synthesis of IDK-cascades. In *Proceedings of the Twenty-Ninth International Conference on Real-Time and Network Systems*, RTNS '21, New York, NY, USA, 2021. ACM.

7 C. K. Chow. An optimum character recognition system using decision functions. *IRE Transactions on Electronic Computers*, EC-6(4):247–254, 1957. doi:10.1109/TEC.1957.5222035.

8 Corinna Cortes, Giulia DeSalvo, and Mehryar Mohri. Learning with rejection. In Ronald Ortner, Hans Ulrich Simon, and Sandra Zilles, editors, *Algorithmic Learning Theory*, pages 67–82. Springer International Publishing, 2016.

9 Bartloomiej Kocot, Pawel Czarnul, and Jerzy Proficz. Energy-aware scheduling for high-performance computing systems: A survey. *Energies*, 16(2), 2023. URL: https://www.mdpi.com/1996-1073/16/2/890.

10 Russell Lee, Jessica Maghakian, Mohammad Hajiesmaili, Jian Li, Ramesh Sitaraman, and Zhenhua Liu. Online peak-aware energy scheduling with untrusted advice. In *Proceedings of the Twelfth ACM International Conference on Future Energy Systems*, e-Energy '21, pages 107–123, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3447555.3464860.

11 C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

12 David Madras, Toniann Pitassi, and Richard Zemel. Predict responsibly: Improving fairness and accuracy by learning to defer. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, pages 6150–6160, Red Hook, NY, USA, 2018. Curran Associates Inc.

13 Aloysius 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.

14 Tim Roughgarden, editor. *Beyond the Worst-Case Analysis of Algorithms*. Cambridge University Press, 2020. doi:10.1017/9781108637435.

15 Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT press, 2018.

16 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53, May 2008.

17 F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In IEEE, editor, *36th Annual Symposium on Foundations of Computer Science: October 23–25, 1995, Milwaukee, Wisconsin*, pages 374–382, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995. IEEE Computer Society Press.

18 Tianming Zhao, Wei Li, and Albert Y. Zomaya. Real-time scheduling with predictions. In *IEEE Real-Time Systems Symposium, RTSS 2022, Houston, TX, USA, December 5-8, 2022*, pages 331–343. IEEE, 2022. doi:10.1109/RTSS55097.2022.00036.

Memory Latency Distribution-Driven Regulation for Temporal Isolation in MPSoCs

Ahsan Saeed
Robert Bosch GmbH, Stuttgart, Germany

Denis Hoornaert
Technische Universität München, Germany

Dakshina Dasari
Robert Bosch GmbH, Stuttgart, Germany

Dirk Ziegenbein
Robert Bosch GmbH, Stuttgart, Germany

Daniel Mueller-Gritschneder
Technische Universität München, Germany

Ulf Schlichtmann
Technische Universität München, Germany

Andreas Gerstlauer
The University of Texas at Austin, TX, USA

Renato Mancuso
Boston University, MA, USA

Abstract

Temporal isolation is one of the most significant challenges that must be addressed before Multi-Processor Systems-on-Chip (MPSoCs) can be widely adopted in mixed-criticality systems with both time-sensitive real-time (RT) applications and performance-oriented non-real-time (NRT) applications. Specifically, the main memory subsystem is one of the most prevalent causes of interference, performance degradation and loss of isolation. Existing memory bandwidth regulation mechanisms use static, dynamic, or predictive DRAM bandwidth management techniques to restore the execution time of an application under contention as close as possible to the execution time in isolation.

In this paper, we propose a novel distribution-driven regulation whose goal is to achieve a *timeliness objective* formulated as a constraint on the probability of meeting a certain target execution time for the RT applications. Using existing interconnect-level Performance Monitoring Units (PMU), we can observe the Cumulative Distribution Function (CDF) of the per-request memory latency. Regulation is then triggered to enforce first-order stochastical dominance with respect to a desired reference. Consequently, it is possible to enforce that the overall observed execution time random variable is dominated by the reference execution time. The mechanism requires no prior information of the contending application and treats the DRAM subsystem as a black box. We provide a full-stack implementation of our mechanism on a Commercial Off-The-Shelf (COTS) platform (Xilinx Ultrascale+ MPSoC), evaluate it using real and synthetic benchmarks, experimentally validate that the *timeliness objectives* are met for the RT applications, and demonstrate that it is able to provide 2.2x more overall throughput for NRT applications compared to DRAM bandwidth management-based regulation approaches.

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

Keywords and phrases temporal isolation, memory latency, real-time system, multi-core

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.4

Funding *Ahsan Saeed*: This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871669.
Denis Hoornaert: Denis Hoornaert was supported by the Chair for Cyber-Physical Systems in Production Engineering at TUM and the Alexander von Humboldt Foundation.
Renato Mancuso: The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grants number CCF-2008799 and CNS-2238476.

© Ahsan Saeed, Denis Hoornaert, Dakshina Dasari, Dirk Ziegenbein, Daniel Mueller-Gritschneder, Ulf Schlichtmann, Andreas Gerstlauer, and Renato Mancuso;
licensed under Creative Commons License CC-BY 4.0
35th Euromicro Conference on Real-Time Systems (ECRTS 2023).
Editor: Alessandro V. Papadopoulos; Article No. 4; pp. 4:1–4:23
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

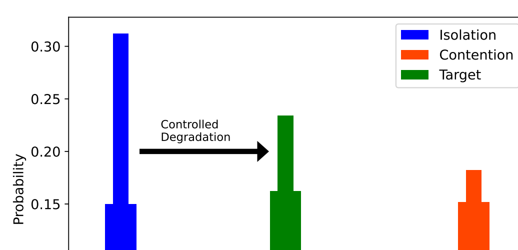


Figure 1 Execution time distributions in isolation (blue) and contention (red). The controlled degradation target can be expressed by reasoning in terms of controlled distribution shift (green).

1 Introduction

An important trend across industrial, automotive and avionics domains is the adoption of MPSoCs. However, a key barrier in designing mixed-criticality systems is the presence of shared resources like the main memory, the cache and the interconnect, which makes it non-trivial to bound the execution time of RT applications running on these MPSoCs. This is because when two or more applications are executed in parallel on different cores, which we refer to as the contention scenario, the interaction between them on shared hardware resources can lead to unforeseen and unpredictable delays [8, 34, 36]. It is well known that memory contention is a key source for performance degradation [7], and practitioners across the industry and academia are looking for solutions that facilitate temporal isolation between applications while using COTS platforms.

Existing hardware-oriented mechanisms for memory interference control require dedicated hardware [2, 11, 13] that is not feasible in COTS multi-core platforms. In contrast, software-oriented memory bandwidth management-based regulation mechanisms are promising grassroots techniques to approach the problem of controlling memory interference by periodically monitoring the memory bandwidth originating from each core and stalling cores when the egress memory bandwidth exceeds a pre-defined threshold. This threshold can be (1) fixed and computed offline for a given combination of applications [5, 41], (2) predicted on the fly [5, 41, 42] or (3) computed dynamically by instrumenting the current memory utilization at the memory controller [23]. A common denominator across the above approaches is that (1) the system parameters for regulation are based on experimental evaluation and not on a formal analysis (2) they focus on restoring the execution time of an application under contention as close as possible to the execution time in isolation.

Ideally, however, the aggressiveness of regulation should directly depend on the target execution time. Indeed, if the RT applications have sufficient slack, less aggressive regulation is desirable as it enables better progress for the NRT applications. Consider the qualitative situation depicted in Figure 1. On the left (resp., right) side of the figure, we depict the distribution of execution time of an application executing in isolation, blue area (resp., contention, red area). Controlled degradation (green area) is achieved if a *bounded shift* is allowed from the solo case and in the direction of the contention case. With this intuition, a *timeliness objective* can be non-ambiguously expressed as a (1) target execution time and (2) a condition on the mass of the execution time distribution that can cross said target.

In this paper, we propose a distribution-driven regulation approach, whose goal is to achieve a *timeliness objective* formulated as a constraint on the probability of meeting a certain target execution time. This definition allows us to unite WCET-like constraints and high-percentile latency constraints typical of real-time cloud systems (tail latency). The basic premise of our approach stems from the observation that the latency distribution of memory transactions of an application under contention gets skewed compared to the execution in isolation. Therefore, it is possible to precisely influence the overall application execution time so long as we can (1) characterize this distribution and (2) affect its shape via regulation. With this basic principle, we first theoretically compute the reference CDF from the distribution of the per-request memory latency for a given target execution time. Then, we enforce first-order stochastical dominance by periodically checking that the CDF of the observed memory latency distribution of the RT application (obtained by sampling at the PMU) stays above the reference CDF of the per-request memory latency. In case this condition is violated, the NRT cores are suspended till the condition of first-order stochastical dominance holds again. If the reference per-request memory latency first-order stochastically dominates the observed latency, then it follows that the overall execution time random variable is dominated by the reference execution time random variable. Consequently, the observed execution time achieves the *timeliness objective*.

The proposed distribution-driven regulation truly considers the impact of memory contention on the latency and execution time of an application, as opposed to memory bandwidth-based [5, 41, 42] or memory utilization-based approaches [23]. Furthermore, we can also control the level of degradation while guaranteeing timeliness by varying the reference CDF of the per-request memory latency.

With this work, we make the following contributions:

1. To the best of our knowledge, our work is the first that demonstrates the use of an interconnect-level PMU to capture the latency distribution of memory transactions and to leverage it for precise control over an application's execution time under contention.
2. We mathematically characterize the distribution of memory latency for an application and demonstrate its effect when the application is executed in isolation and contention.
3. We provide a formal mathematical proof supporting how our proposed approach meets the imposed *timeliness objective* for the RT applications, ultimately enabling controlled degradation.
4. Finally, we perform an evaluation on a COTS platform (Xilinx Ultrascale+ MPSoC) using an extensive set of realistic and synthetic benchmarks from the San Diego Vision Benchmarks [35], DAPHNE [30], and IsolBench [33] suites. We demonstrate its effectiveness in (1) allowing controlled degradation, (2) providing probabilistic guarantees for RT application, and (3) reducing the execution time of NRT applications by up to 2.2x compared to DRAM bandwidth management-based regulation approaches.

The rest of the paper is organized as follows: Section 2 provides the survey of related work. Section 3 describes the system model and the main assumptions of our approach. After presenting the main theory behind our approach and its mathematical formalization in Section 4, Section 5 describes the overall architecture and the main algorithm of our approach. Section 6 describes the implementation, and Section 7 discusses the experimental setup and presents the results. Finally, Section 8 concludes with a summary and outlook on future work.

2 Related Work

There has been a significant amount of work [18] to tackle the issue of memory interference.

The first category includes techniques that essentially employ *memory bandwidth management-based regulation*. In this category of approaches, the effects of memory contention are statically regulated by controlling the outgoing memory bandwidth from each core as in MemGuard [5,41,42], or by directly measuring the utilization at the memory controller [23] and then based on the observed utilization, dynamically regulating the outgoing memory bandwidth from each of the cores. In these approaches, the designer has to experimentally derive the correct system parameters, and furthermore, there are no formal techniques to guarantee the impact of such a regulation on the execution time of the application.

The second category includes *profile-driven approaches* like E-WarP [27, 29] and the work in [1], where an application's behavior is profiled to sufficiently characterize it. Then, together with insights into the underlying regulation mechanism – E-WarP uses Memguard under the hood – it is possible to accurately predict the worst-case execution time. In contrast, the proposed approach in this paper is not about predicting the WCET but rather about setting a target execution-time distribution and adjusting the regulation scheme accordingly.

The third category of approaches falls broadly into the category of *WCET estimation approaches* [14, 18, 20]. These approaches perform WCET estimation by leveraging detailed models of the memory subsystem and do not assume any specific regulation approach. They only consider worst-case memory access latencies considering a certain arbitrary memory placement (bank arrangement) and the underlying workload.

Next, there are the hardware-based regulation mechanisms, which include using a dedicated memory controller [2] or additional hardware like FPGAs [11, 13], which is orthogonal to our approach. In addition, embedded high-performance platforms are increasingly offering QoS modules [25,31,45] on the interconnect between masters (CPUs, GPUs, DMAs) and main memory to regulate and prioritize memory requests. However, the existing QoS modules account for the traffic generated by the core cluster connected to the interconnect as a single master, which does not alleviate cross-core contention [21]. Secondly, a static QoS configuration may lead to inefficiencies in the utilization of the underlying DRAM subsystem for dynamic workloads.

Other hardware-based techniques for COTS platforms, such as RDT [9,28] and MPAM [44], essentially enforce a desired memory bandwidth limit at the hardware-level. This reduces the regulation overhead and significantly improves the granularity of bandwidth regulation. The recently proposed MemPol [46] loosely belongs to this category because it leverages debug interfaces to halt/resume CPUs with the goal of enforcing a target bandwidth. Despite said benefits, the aforementioned shortcomings of memory bandwidth management-based regulation are still present. Nonetheless, a promising direction for future work entails combining the techniques proposed in this paper with hardware-based bandwidth enforcement.

We approach the problem from a different perspective by not relying on the notion of DRAM bandwidth. Instead, we directly reason on the properties of the observed distribution of latencies for the memory transactions performed by the application under analysis. Our approach starts by considering design-time timeliness constraints and uses one such specification to construct a target cumulative distribution (CDF). The latter is then used to enact regulation. The proposed approach also makes no assumptions on the memory transactions generated by the contending applications.

3 System Model and Assumptions

We hereby review the key assumptions and the system model required for the results presented in Section 4 to hold. These assumptions are also experimentally validated in Section 7.2 and Section 7.3.

A1: Multicore Platform Topology. We assume a system comprised of m application CPUs $\Pi_1, \ldots, \Pi_m$. For simplicity, we assume that the high-criticality workload is only deployed on CPU Π_1, which can be considered the *real-time core*. The memory hierarchy comprises zero or more levels of cache. Cache misses caused by `load` or `store` instructions at the last-level cache (LLC) cause read/write memory requests to be initiated towards a single shared main memory subsystem via a single shared bus. Note that we distinguish between memory instructions (`load`/`store`) and the resulting traffic that they might cause in terms of read (and possibly write) requests to the underlying main memory subsystem.

A2: Cache Model. We assume that (1) either all the cache levels are private per-core caches, or (2) if shared cache levels exist, they can be partitioned among the cores to prevent inter-core cache interference. All the cache levels adopt a write-back, write-allocate policy. By write-allocate, `store` instructions that cause a cache miss to trigger a read memory request downstream to fill the cacheline to be modified. A cacheline that has been modified is marked as *dirty*. By write-back, cache refills might trigger a write memory request downstream if the cache replacement policy has selected a dirty cacheline for eviction. We make no assumption about the specific cache replacement policy adopted by the cache controllers at the different levels. We make no assumption about the inclusiveness of adjacent cache levels.

A3: In-order CPUs. We assume that the considered CPUs are unable to reorder instructions. Thus, the latency incurred by pending `load` instructions is additive with respect to the time spent executing instructions that do not perform memory operations. The same is true for `store` instructions. This assumption is pessimistic yet safe if out-of-order CPUs are considered instead.

Timing anomalies arising due to microarchitectural effects can violate this assumption. In this work, we followed a measurement-based evaluation approach. Therefore, timing anomalies are accounted for in the measured runtime. If these anomalies are to be estimated using static analysis, the work in [12] demonstrates that timing anomalies can be statically bounded and accounted for at design time without introducing an intractable amount of pessimism.

A4: Blocking reads, non-blocking writes. As per A2, both `load` and `store` instructions cause an LLC cache miss to trigger a read request to the main memory. As per A3, the latency incurred by such read requests is additive with respect to the time spent by the rest of the instructions that do not generate main memory requests. Conversely, if a memory instruction triggers a write-back to the main memory, the resulting write memory transaction is carried out non-blockingly with respect to the instruction stream under analysis. Therefore, the latency of read requests in main memory is *on the critical path* from the standpoint of total execution time, while the latency of write requests is not. This is not to say that the contention generated by write requests is not considered, but rather that what matters is their impact on the latency of read transactions.

Note that, in typical DRAM subsystems, batched write requests could be prioritized over reads, causing read requests to temporarily stall. However, by controlling the latency distribution of read requests, one can control how this reflects into the total execution time, essentially factoring in the overall impact of write requests.

A5: Measurable Read Latency Distribution. We assume that the platform provides a performance monitoring unit (PMU) capable of collecting measurements on the latency of read memory requests. The PMU shall be located at the interface of the shared bus and main memory subsystem. The latency is measured as the difference between the timestamp at which a read request is forwarded to the main memory and the timestamp at which the response for the said request is returned (request turnaround time). We assume that, when queried, the PMU can return (an approximation of) the distribution of the observed latencies of read requests issued by a core Π_k under analysis. We will discuss the ability to do so in commercial platforms in Section 6.

A6: Computation and Read-latency Additivity. By A4 and A5, we can decompose the worst-case execution time E as a sum of two contributions $E = C + L$, where L is the total latency of read memory transactions. Let N denote the worst-case number of read requests and let us indicate the per-request latency as l_i, then $L = \sum_{i=1}^{N} l_i$. C denotes the time spent for anything other than waiting for read responses, and is a constant, regardless of whether the workload executes in isolation vs. contention. Conversely, l_i and thus L and E are random variables that are affected by the level of congestion of the main memory subsystem. In practice, we observe a small deviation (less than 1.8%) in the value of C when measured in isolation vs. under contention, as evaluated in detail in Section 7.3. One such deviation might arise from contention over Miss Status Holding Registers (MSHR) [33] or LLC tag/data banks [6]. For the sake of simplicity, C is assumed to be constant in our theoretical formulation. In practical instantiations of our framework, this value should be experimentally derived and a safe upper-bound on the compute-only time shall be used.

A7: Profiled Critical Workload. We assume that the high-criticality workload deployed on Π_1 can be profiled offline to derive the worst-case execution time E_{isol} and total read latency L_{isol} in isolation. This can be done using traditional measurement-based approaches and allows us to upper-bound the value of $C = E_{isol} - L_{isol}$, which is the time spent by the CPU to carry out any other operation except waiting for read requests to be fulfilled. As per A2, C is computed with statically partitioned shared caches (if any). As per A5, L_{isol} measurement is enabled by the PMU.

A8: I.I.D. Read Transaction Latencies. We assume that l_i are independent samples from the same (unknown) distribution. Intuitively, the independence arises from the fact that between any two subsequent read transactions, a random amount of time can elapse, and a random amount of congestion can be caused by interfering CPUs. Thus, l_i's are independent and identically-distributed (i.i.d.) random variables.

4 Distribution-Driven Regulation

In this section, we introduce the theoretical results that represent the foundation of the proposed distribution-driven regulation. We introduce the notations in Table 1.

Table 1 Summary of notation used.

Symbols	Descriptions	Symbols	Descriptions
E_{isol}	Total execution time in isolation	$\bar{l}_{\sigma^2}$	Variance of read memory transactions reference
E_{reg}	Total execution time under regulation	L_{isol}	Total latency of read memory trans. in isolation
$\bar{E}$	Total execution time target	l_{min}	Min read latency
C	Non-memory compute time	l_{max}	Max read latency
L	Total latency of read memory transactions	l_i	Latency of an individual read memory transaction i
l_μ	Mean latency of read memory transactions	N	Worst-case number of read requests
l_{σ^2}	Variance of read memory transactions	α	Acceptable tolerance for execution time to exceed $\bar{E}$
$\bar{l}_\mu$	Mean latency of read memory trans. reference		

Regulation Goal. Unlike the related literature surveyed in Section 2, our goal is to achieve a *timeliness objective* formulated as a constraint on the probability of meeting a certain execution time target $\bar{E}$. Formally, given an execution time target $\bar{E}$ and an acceptable error $\alpha \in [0, 1]$, the goal of regulation can we written as

$$P(E_{reg} \leq \bar{E}) \geq 1 - \alpha, \tag{1}$$

where E_{reg} is the actual execution time observed under regulation and (possibly) in the presence of main memory contention for the application under analysis. When α is such that $\alpha \to 0$, then $\bar{E}$ represents a worst-case execution time (WCET) constraint. Note however that the timeliness constraint formulation in Eq. 1 is more generic. For instance, setting $\alpha = 0.01$ expresses a 99th-percentile tail latency requirement on E_{reg}.

Goal-driven Regulation Strategy. We hereby describe how the regulation strategy can be built from the goal formulated in Eq. 1 given a value of $\bar{E}$ and α. Following the notation and assumptions in A6 (Section 3), we can rewrite Eq. 1 as follows:

$$P(C + L \leq \bar{E}) = P\left(\sum_{i=1}^{N} l_i \leq \bar{E} - C\right) \geq 1 - \alpha. \tag{2}$$

The key insight into our approach is that, by controlling the distribution of per-request latency l_i via regulation, we can directly control the distribution of the total memory latency L and thus impact the distribution of E_{reg} to satisfy Eq. 1.

As we previously mentioned, l_i's are independent and identically-distributed random variables (as per A8) following an unknown distribution. Call l_μ and l_{σ^2}, respectively, the (unknown) mean and variance of the l_i random variables. From the Central Limit Theorem (CLT) [10], it holds that the random variable Z constructed as

$$Z = \frac{\sum_{i=1}^{N} l_i - N l_\mu}{\sqrt{N l_{\sigma^2}}} = \frac{L - N l_\mu}{\sqrt{N l_{\sigma^2}}} \sim \mathcal{N}(0, 1), \tag{3}$$

follows a standard normal distribution, i.e. a normal distribution with mean $\mu = 0$ and variance $\sigma^2 = 1$. The latter property is captured by the notation $Z \sim \mathcal{N}(0, 1)$. Note that Eq. 3 only holds for large values of N. Since our goal is to analyze and regulate memory-intensive applications, this condition holds. In fact, our experiments described in Section 7 highlight that for the considered applications, the order of magnitude of N is somewhere between 10^6 and 10^7.

From Eq. 3 we can derive that $L \sim \mathcal{N}(N l_\mu, N l_{\sigma^2})$. Let us indicate with $\Phi(x)$ the Cumulative Distribution Function (CDF) of the standard normal distribution. We can then rewrite Eq. 2 as follows:

$$P(L \le \bar{E} - C) = \Phi\left(\frac{(\bar{E} - C) - Nl_\mu}{\sqrt{Nl_{\sigma^2}}}\right) \ge 1 - \alpha. \tag{4}$$

So far, we have treated l_μ and l_{σ^2} as unknown values. The insight at this point is that, when regulation is performed (by pausing/resuming the activity of interfering cores), we can exert direct control over the underlying distribution of $L = \sum_{i=1}^{N} l_i$ and thus over the value of Nl_μ and Nl_{σ^2}. In fact, our goal is not to enforce a specific value of l_μ and l_{σ^2}. Instead, it is enough to identify two values $\bar{l}_\mu$ and $\bar{l}_{\sigma^2}$ such that the following inequality holds for every value of $\bar{E} \in \mathbb{R}^+$:

$$\Phi\left(\frac{(\bar{E} - C) - Nl_\mu}{\sqrt{Nl_{\sigma^2}}}\right) \ge \Phi\left(\frac{(\bar{E} - C) - N\bar{l}_\mu}{\sqrt{N\bar{l}_{\sigma^2}}}\right) \ge 1 - \alpha. \tag{5}$$

Regulation Condition. Recall from A5 in Section 3 that we are able to periodically *snapshot* the distribution of read latencies. By enacting start/stop control over the interfering cores, we can impact such distribution. We are now ready to derive the condition according to which, given a snapshot, we should pause or resume the activity of the interfering cores.

More specifically, we can *observe* the CDF of the random variable l_i while the application under analysis is running. Call this observed CDF function $F_l(t) = P(l_i \le t)$. If regulation is applied such that

$$\forall t \in \mathbb{R}^+, F_l(t) \ge \Phi\left(\frac{(\bar{E} - C) - \bar{l}_\mu}{\sqrt{\bar{l}_{\sigma^2}}}\right) = \bar{F}_l(t), \tag{6}$$

then we have two properties. The first, is that $\bar{F}_l(t)$ is the CDF of a random variable $l_i^{norm} \sim \mathcal{N}(\bar{l}_\mu, \bar{l}_{\sigma^2})$. The second is that l_i^{norm} is said to first-order stochastically dominate l_i [26]. Indeed, Eq. 6 is one possible definition of first-order stochastic dominance, also indicated with the notation $l_i^{norm} \ge_1 l_i$.

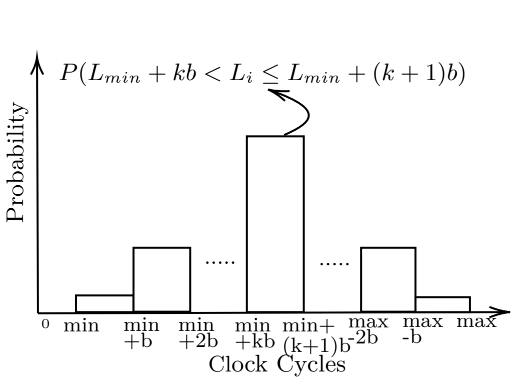
It is a known result [26, Theorem 1.A.3] [19, Lemma 6] that stochastical dominance between random variables implies stochastical dominance in the aggregate. Formally, given two random variables X and Y and a positive integer k, if Y is k-th order stochastically dominated by X (i.e., $X \ge_k Y$), then $\forall n \in \mathbb{N}^+$ and i.i.d. replicas $X_1, \ldots, X_n$ of X and $Y_1, \ldots, Y_n$ of Y it holds that

$$\sum_{i=1}^{n} X_i \ge_k \sum_{i=1}^{n} Y_i \implies \sum_{i=1}^{n} X_i \ge_1 \sum_{i=1}^{n} Y_i. \tag{7}$$

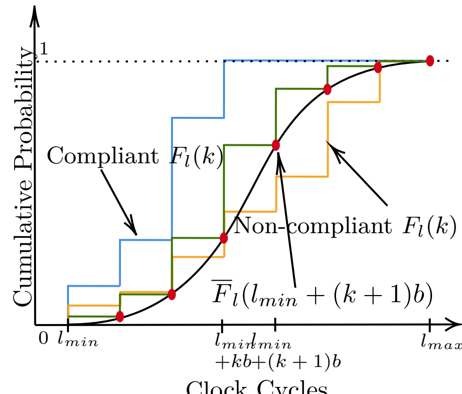
Next, we note that from Eq. 7 and 6 it immediately follows that $\sum_{i=1}^{N} l_i^{norm} \ge_1 \sum_{i=1}^{N} l_i$. Moreover, by leveraging the properties of the normal distribution [17], we know that $\sum_{i=1}^{N} l_i^{norm} \sim \mathcal{N}(N\bar{l}_\mu, N\bar{l}_{\sigma^2})$. This brings us to the final step. That is, the random variable L under regulation is first-order stochastically dominated by a normal distribution of mean $N\bar{l}_\mu$ and variance $N\bar{l}_{\sigma^2}$. This means that, as long as Eq. 6 is ensured via regulation, Eq. 5 holds.

Final Formulation. Putting everything together, we have the following workflow. First, given the target $\bar{E}$ and α, numerically compute $\bar{l}_\mu$ and $\bar{l}_{\sigma^2}$ such that

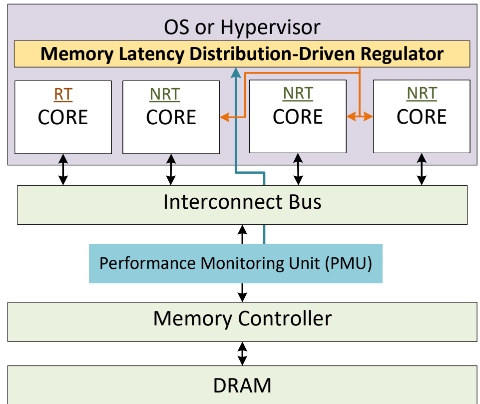
$$\Phi\left(\frac{(\bar{E} - C) - N\bar{l}_\mu}{\sqrt{N\bar{l}_{\sigma^2}}}\right) \ge 1 - \alpha \tag{8}$$



■ **Figure 2** Visual representation of the read-latency PMF.



■ **Figure 3** Discretized, compliant and non-compliant latency CDF.



■ **Figure 4** Overview of our system architecture consisting of MPSoC.

holds. Second, use the same values of $\bar{l}_\mu$ and $\bar{l}_{\sigma^2}$ to construct the target per-request latency CDF $\bar{F}_l$ as described in Eq. 6. Next, at runtime, observe the CDF of l_i, namely F_l, and pause/resume (regulate) the activity of the non-real-time CPUs to ensure that $\forall t \in \mathbb{R}^+, F_l(t) \geq \bar{F}_l(t)$. So long as this inequality holds, it also holds that $P(C + L \leq \bar{E}) = P(L \leq \bar{E} - C) \geq 1 - \alpha$ because Eq. 5 holds.

4.1 Discrete-domain Formulation

The results derived so far in Section 4 assume that we are able to snapshot online a *continuous* distribution of read latency accesses. This is practically impossible with realistic hardware. In this subsection, we relax precisely this requirement.

Let l_{min} and l_{max} be the minimum and maximum possible read latency. Consider a realistic PMU that defines K latency observation bins with configurable size b. If a transaction i was counted in the first bin, then its latency l_i was somewhere in the range $[l_{min}, l_{min} + b)$; more in general, if it was counted in the kth bin with $k \in \{0, \ldots, K-1\}$, then its latency was somewhere in the range $[l_{min} + kb, l_{min} + (k+1)b)$.

When queried, the PMU reports the number of read transactions completed by Π_1 whose latency fell in each of the K bins. Assume that this number is cumulative since the time at which the application was launched – if it is reset after a snapshot, e.g. to prevent overflows

Algorithm 1 Memory Latency Distribution-Driven Regulator.

```
   input : number of latency bins K, reference CDF F̄_k ∀ k ∈ {0...K−1}
 1 foreach regulation interval r do
 2     foreach latency bin k ∈ {0...K−1} do
 3         Sample the height of latency bin l_{k,r}
 4         γ_{k,r} = γ_{k,r−1} + l_{k,r}
 5     end
 6     foreach latency bin k ∈ {0...K−1} do
 7         f_{k,r} = γ_{k,r} / Σ_{k=0}^{K−1} γ_{k,r}     ▷ Normalize bins to obtain PMF
 8         F_{k,r} = Σ_{m=0}^{k} f_{m,r}                ▷ Construct observed CDF
 9     end
10     if F_{0,r} < F̄_0 ∨ ··· ∨ F_{K−1,r} < F̄_{K−1} then
11         suspend all NRT cores
12     else
13         resume all NRT cores
14     end
15     r = r + 1
16 end
```

in the counters, then it can be accumulated in software at each snapshot. In software, divide the number of transactions in each bin (i.e. the height of the bin) by the total number of transactions in the entire snapshot. The result is a valid observed Probability Mass Function (PMF) $f_l(k)$ for the read request latency l_i for the generic request i. Figure 2 provides a visual representation of the PMF. In other words, the height of each bin provides the value of $f_l(k) = P(l_{min} + kb \leq l_i < l_{min} + (k+1)b)$. From the acquired PMF, it is easy to compute the corresponding observed CDF as

$$F_l(k) = \sum_{j=0}^{k} f_l(j) = P(l_i < l_{min} + (k+1)b). \tag{9}$$

Recall that (Eq. 6) we can construct a normal distribution $\bar{F}_l(t)$ of reference with appropriate values of $\bar{l}_\mu$ and $\bar{l}_{\sigma^2}$ such that Eq. 8 is satisfied. At runtime, whenever a new read latency distribution snapshot is acquired, it is enough to check the following condition:

$$\forall k \in \{0, \ldots, K-1\}, F_l(k) \geq \bar{F}_l(l_{min} + (k+1)b). \tag{10}$$

This condition is visually depicted in Figure 3. Indeed, if the condition expressed in Eq. 10 holds, then our reference $l_i^{norm} \sim \mathcal{N}(\bar{l}_\mu, \bar{l}_{\sigma^2})$ first-order stochastically dominates l_i. This is the case for the blue curve in Figure 3. Conversely, if for some k Eq. 10 does not hold, the non-real-time CPUs must be paused – regulation must be triggered. This is the case for the orange line in Figure 3. The implicit assumption, which we validate in Section 7.3, is that pausing the interfering CPUs allows to *shift* the observed $F_l(k)$ in subsequent snapshots.

Finally, note that numerically computing the value of $\bar{F}_l(t)$ online can lead to excessive overhead in the regulator. Instead, the K values of $\bar{F}_l(k)$ necessary to check the validity of Eq. 10 can be pre-computed offline and stored in a lookup table for efficient online retrieval. These values are depicted as red dots in Figure 3.

5 System Overview

An overview of our system architecture is depicted in Figure 4. We consider an MPSoC in which a core designated as RT core is dedicated to host time-sensitive RT applications, while the others are designated as NRT cores that host performance-oriented NRT applications.

The purpose of the memory latency distribution-driven regulator introduced in Section 4 is to achieve the *timeliness objective* (Equation (1)) on the execution time of applications running on the RT core. The regulator is activated periodically on each NRT core using a timer interrupt. The timer interrupt triggers the sampling of memory latency distribution using the Performance Monitoring Unit (PMU) (shown in blue in Figure 4) for the memory transactions originating from RT core. This memory latency distribution is normalized to obtain the probability mass function (PMF), as described in Section 4.1 and then is used to derive the cumulative distribution function (CDF). From the CDF, we enforce the rule of first-order stochastic dominance (Equation (6)), which states that if any bin violates the reference CDF for the target distribution of execution time, the regulation is triggered, and all the NRT cores are suspended, as highlighted with red lines in Figure 4.

In principle, the regulator could reside either in software, such as the Operating System (OS) or hypervisor, or in hardware, such as a Field Programmable Gate Array (FPGA). For analysis and evaluation of the mechanism, the regulator optionally stores the PMF and key characteristics in the DRAM memory.

The proposed mechanism can be implemented on any platform on which we are able to measure (1) memory latency distribution and (2) filter the memory transaction on a per core basis.

5.1 Memory Latency Distribution-Driven Regulator Algorithm

Algorithm 1 sketches our proposed distribution-driven regulation. Let the total number of bins in the memory latency distribution be denoted by K. Furthermore, we denote by $\bar{F}_k$ the reference CDF assigned to each bin.

At the beginning of each regulation interval $r > 1$, the regulator first samples the number of transactions (since the last interval) with latency that falls in bin $l_{k,r}$. This is repeated for each bin (Line 3). The samples are accumulated into the variable $\gamma_{k,r}$ (Line 4). We then apply height normalization to derive the PMF f_k (Line 7). The PMF is converted into a CDF F_k by summing up the probabilities associated with the variable up to each bin (Line 8). This CDF F_k is then compared against the reference CDF $\bar{F}_k$ for each bin (Line 10). If the condition in Eq. 10 does not hold, all the NRT cores are suspended (Line 11). They will be resumed only when Eq. 10 holds again (Line 15).

The theoretical formulation provided in Section 4 assumes that the PMF (or CDF) of the per-request latency can be observed infinitely fast. Clearly, this is not possible in realistic hardware, hence a non-zero regulation interval T_r must be picked. Because of that, what could happen is that during T_r, the distribution of memory latencies shifts so drastically that it cannot be recovered. Although this can happen, its effect can be easily bounded. In the worst-case, right after a snapshot that satisfied Eq. 10 (otherwise, the NRT cores would be stopped) with exact equalities between left- and right-hand sides, a back-to-back sequence of memory transactions with latency l_{max} occurs. These can be at most $\lceil T_r/l_{max} \rceil$ because Π_1 is an in-order CPU (A3 in Section 3). Thus, the extra time cost $H = (l_{max} - l_{min})\lceil T_r/l_{max} \rceil$ can be accounted for by computing a new, more restrictive $\bar{E}' = \bar{E} - H$. Interestingly, since we can observe the typical latency distribution under unrestricted contention, it is also possible to compute the probability that such a case can occur.

6 Implementation

We have performed a full-system implementation that includes a partitioning hypervisor augmented to support the proposed memory latency distribution-driven regulator. The implementation is carried out on the Xilinx Ultrascale+ Multi-Processor System-on-Chip (MPSoC) ZCU102 [40]. The SoC features 4 ARM Cortex A53 [4] cores clocked at 1.2 GHz. Each core has its own private L1 data and instruction cache, whereas the 4 cores share a unified L2 cache. The SoC also features a tightly-coupled FPGA, which is not needed to implement the proposed approach. We only use the FPGA for the validation experiments on the nature of DRAM read transaction latencies conducted in Section 7.2.

We use the Jailhouse-RT partitioning hypervisor [15, 27] to partition resources in our system, which is an ideal choice for this type of implementation because it is lightweight, easy to port/modify, includes support for cache coloring [16, 43] and bandwidth regulation, and is open-source.

6.1 AXI Performance Monitor (APM)

We sample the memory latency in the Xilinx Ultrascale+ MPSoC [40] using the AXI Performance Monitor (APM) hardware module. The APM measures the key performance metrics like the amount of read/write memory transactions, min/max/total latency, and other performance metrics for the AMBA AXI [3] in a system. The APMs implemented on Xilinx Ultrascale+ MPSoC [40] are based on the Xilinx AXI Performance Monitor available as a LogiCORE IP [37].

The APM has 10 hardware counters that can be configured to simultaneously monitor up to 10 performance metrics for any interface points called slots on the AXI interconnect. There is also a global-clock counter in addition to these 10 hardware counters that run at the APM clock frequency of 533.5 MHz.

The APM can be configured to monitor the performance metrics for a particular slot using the *Metric Selector* register. Furthermore, the APM contains a Range Incrementer module that compares the performance metric count with the low and high ranges from the *Range* register and increments the count of the given performance metric by one if the value falls within the limits. The Range Incrementer is useful in obtaining the read/write latency ranges that we leverage in this work to sample the memory latency distribution.

We configured 8 *Metric Selector* registers in conjunction with 8 *Range* registers to monitor read memory latency (as defined in Section 3 *A6: Measurable Read Latency Distribution*) with respectively low and high ranges of 0-40, 41-80, 81-120, 121-160, 161-200, 201-240, 241-280, and 281-2000 clock cycles. The rationale behind the selection of these ranges is discussed in Section 7.4. These 8 performance metrics provide the number of read memory transactions that fall within the given read memory latency limits, referred to as bins. Furthermore, 2 *Metric Selector* registers are configured to report the total number of read transactions and total read latency. The total number of read transactions is N, as used throughout the mathematical formalization in Section 3. Additionally, we verify that the total number of read transactions and the sum of all bins are always the same. This ensures that no memory transaction *escapes* the bins. The global-clock counter is used as the reference for all the calculations in this paper. The included hardware counters can be set and read via a memory-mapped interface.

The APM slot is configured to monitor the AXI communication between the cores and the memory controller. In addition, we employ the AXI ID filtering to monitor the transactions emanating from a core with a certain AXI ID. The AXI IDs for the cores are evaluated experimentally. Once the AXI IDs for each core have been determined, we utilize the *Filter* and *Mask* registers to set up AXI ID filtering.

Currently, the APMs are adopted in Xilinx Ultrascale boards. However, since these APM IPs are part of the AXI bus, they are deployable on other SoCs. They can also be deployed in programmable logic (FPGA) to gather statistics on the traffic observed over AXI bus segments generated, for instance, by in-FPGA accelerators.

7 Validation and Evaluation

In this section, we first experimentally validate the key assumptions presented in Section 3. Then we discuss the key design parameters of our system. Finally, we present a full system evaluation where we validate the effectiveness of our approach to ensure the timeliness of different sets of applications.

7.1 Experimental Setup

We evaluate our approach on the Xilinx Ultrascale+ Multi-Processor System-on-Chip (MPSoC) ZCU102 [40] as introduced in Section 6. A combination of real-world [35], [30], and synthetic [33] benchmarks are used to evaluate the proposed approach. For our real-world benchmarks, we use a subset of the benchmarks in the San Diego Vision Benchmark Suite (SD-VBS) [35]. The input dataset for the benchmark applications comes in 9 different sizes. Since we are interested in DRAM-bounded applications, we use the ones with the largest input data size (named *FullHD*). The other benchmark suite is the Darmstadt Automotive Parallel Heterogeneous Benchmark Suite (DAPHNE) [30], which represents parallelizable workloads from the automotive domain. For our evaluation, we used the applications that run exclusively on the CPU. We also use a synthetic 'Bandwidth' benchmark from the IsolBench suite [33] that is engineered to continuously perform memory write operations. In the rest of the paper, we refer to this benchmark as the *MemBomb* application.

Unless otherwise stated, all experiments refer to the *isolation scenario* or simply *isolation* in which the disparity application is running on the designated RT core with no other applications running in parallel. In contrast, a *contention scenario* or simply *contention* happens when the same *disparity* application is running on the designated RT core while synthetic *MemBomb* applications are running on the three NRT cores. The *disparity* application is selected as it has the lowest average IPC and the highest average memory utilization [23] in the benchmark suite, making it an ideal candidate for demonstrating memory interference-related effects.

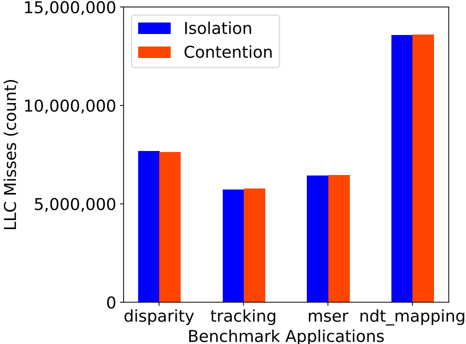
For consistency, we always activate the hypervisor. The regulator is activated on each NRT core to facilitate comparison with a memory bandwidth management-based regulation (MemGuard [5]). However, the current implementation can be extended to sample the PMU values from only one NRT core responsible for suspending the other NRT cores. All the obtained results are calculated on 100 runs for each configuration to remain statistically significant.

7.2 Validation of I.I.D. Assumption A8

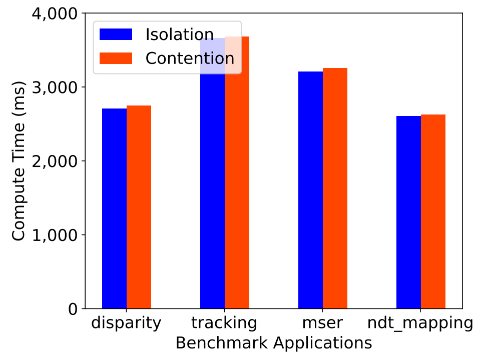
In order to validate hypothesis A8 in Section 3, i.e., that the latencies of read memory transactions emitted by the cores are i.i.d., we perform 10 different statistical tests called Permutation Tests [32]. These tests are designed to find evidence that empirical samples are i.i.d.. The rationale is that if i.i.d. holds in all cases, the regulation system is guaranteed to be operated correctly. Conversely, if the i.i.d. property is validated only in some cases, a full-system implementation and evaluation are necessary to assess the correct end-to-end behavior of a system that employs the proposed distribution-driven regulation.

Table 2 Summary of permutation testing results for Synthetic (table upper half) and Real-world (table lower half) memory traffic. Test pass noted with ✓and fail with ×.

Test no.	1	2	3	4	5	6	7	8	9	10	Pass (%)
Synthetic Benchmarks: AXI Traffic Generator											
Rand. Pattern + Rand. ITG	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	100
Rand. Pattern + Fix ITG	✓	✓	✓	×	✓	✓	×	✓	✓	×	70
Seq. Pattern + Rand. ITG	✓	✓	×	×	✓	✓	✓	✓	✓	✓	80
Seq. Pattern + Fix ITG	✓	✓	×	×	✓	✓	×	✓	✓	×	60
Real-world Benchmarks: SD-VBS											
Best-case	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	100
Worst-case	✓	×	✓	×	✓	✓	×	✓	×	×	50
Mode-case	✓	✓	✓	×	✓	✓	✓	✓	✓	✓	90



(a) Similar LLC misses for applications on RT core in isolation and contention.



(b) Similar Compute Time for applications on RT core in isolation and contention.

Figure 5 Experimental results to validate the key assumptions, as stated in detail in Section 3, hold for our system.

Performing permutation testing requires measuring the memory latency of individual memory transactions at the finest granularity. The aforementioned APMs can only measure aggregated latency values and are thus not suitable for the purpose. Instead, and only for these experiments, we leverage the tightly-coupled FPGA of the evaluation platform.

The experiment is divided into four successive steps: (1) generate memory traffic, (2) capture the activity at the AXI level, (3) measure and compile each transaction's response time, and (4) perform a set of permutation tests.

To evaluate the memory latency of both synthetic and real-world benchmarks, we implement two distinct FPGA designs. The first FPGA design is composed of an *AXI Traffic Generator (ATG)* [38], which generates heavy synthetic memory traffic toward the memory controller. We configure the ATG to generate four types of access patterns that combine *random* and *sequential* accesses with *random* and *fixed* inter-transaction gaps (ITG). The traffic activity created by the ATG is captured and stored for post-processing by an *Integrated Logic Analyzer* [39] (ILA), which is also instantiated in the FPGA.

The second FPGA design is implemented to evaluate the real-world memory traffic by observing the activity originating from the main CPUs running SD-VBS benchmarks in isolation. The design is a simplified version of the approach introduced in [22] and consists

of only a loopback IP linking the core cluster with the memory controller through the FPGA (i.e., no transformations are performed on the transactions' address). Similarly, the Jailhouse-RT hypervisor [15] is instrumented to target the FPGA memory range instead of the memory controller, making the hypervisor and benchmark memory traffic observable via an ILA. We run different SD-VBS benchmarks with different inputs in a sequence and randomly acquire fragments of memory traces. Thus, while we know that the captured activity belongs to *some* SD-VBS benchmark, we cannot determine which trace corresponds to which specific benchmark.

Table 2 shows the results of the first 10 permutation tests performed on the two FPGA designs, on the top and bottom, respectively. For synthetic benchmarks, the number of passed tests increases as randomness in the pattern, and ITG is introduced. Therefore, for ATG with random memory access pattern and random ITG has the highest tests pass of 100%, whereas sequential memory access pattern with fixed ITG has the lowest test pass of 60%. Hence, the percentage of tests pass increases as access pattern and ITG randomness grow.

For real-world benchmarks, 30 snapshots of memory traffic are captured. Since applications have different phases, the ILA buffer is small, and memory transactions are captured asynchronously, we observed variation in the results of permutation tests. In the best-case scenario, all tests are passed, although pass percentages as low as 50% have been seen on rare occasions. The mode (value that appears most often) indicates a 90% pass.

In summary, the permutation testing indicates that not all tests are passed under all scenarios, albeit an indication that A8 holds in most of the cases has emerged. Nonetheless, we conduct a full-stack implementation to verify that the *timeliness objective* (Equation (1)) we impose is, in fact, met with real-world applications.

7.3 Validation of Other Key System Assumptions

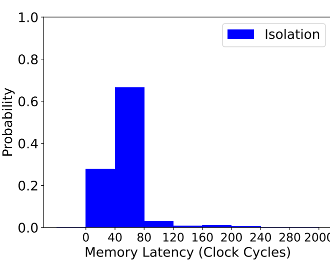
In this subsection, we experimentally validate that the key assumptions, as stated in detail in Section 3, hold for our system.

Validation of A2: Cache Model. First, we show that the total numbers of LLC misses for an application executed in isolation and contention scenarios are comparable. Figure 5a illustrates the average total number of LLC misses that occur during 100 runs for *disparity*, *tracking*, *mser* and *ndt_mapping* in isolation and contention, respectively. It can be observed that the total number of LLC misses is comparable in both scenarios, with an average difference of less than 1% in their counts. This demonstrates that there is no inter-core cache interference, which is consistent with assumption *A2*.

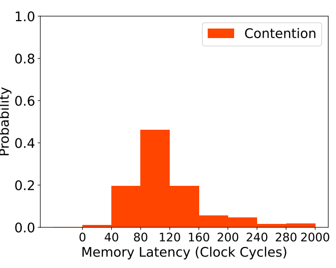
Validation of A6: Computation and Read-latency Additivity and A7: Profiled Critical Workload. Next, we show that the compute time C of an application remains the same in isolation and contention. We measure the worst-case execution time E and the total latency of read memory transactions L and determine the compute time C by: $C = E - L$

In Figure 5b, it is shown that the compute time of the application under consideration (*disparity*, *tracking*, *mser* and *ndt_mapping*) is similar in both the scenarios, with an average difference of less than 1.8%. Thus, assumptions *A6* and *A7* hold.

Validation of A5: Measurable Read Latency Distribution. Finally, we demonstrate the capability of measuring (an approximation of) the latency distribution of read memory transactions in a COTS platform – without redirecting memory transactions through the FPGA – as stated in *A5*.



(a) Isolation.



(b) Contention.

Figure 6 Impact of memory interference on the shape of normalized memory latency distribution for *disparity* on RT core.

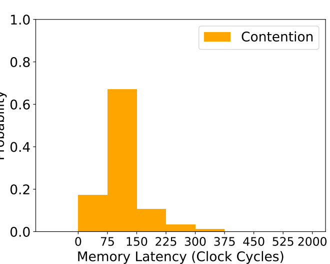


Figure 7 Impact of bin size on the shape of memory latency distribution.

Figure 6 shows the normalized read memory latency distribution obtained from the APM present in the evaluation platform (Xilinx Ultrascale+ MPSoC [40]) in isolation and contention. According to Figure 6a, the majority of individual memory read transactions for *disparity* have a latency of less than 80 clock cycles in isolation.

When multiple contending *MemBomb* applications are running in parallel, the *disparity* benchmark experiences a significant increase in memory latency, resulting in a shift of the memory latency distribution to the right (higher memory latency bins), as seen in Figure 6b. Under contention, the majority of individual memory read transactions have latency in the range of 41 to 160 clock cycles.

7.4 Configuration Parameters

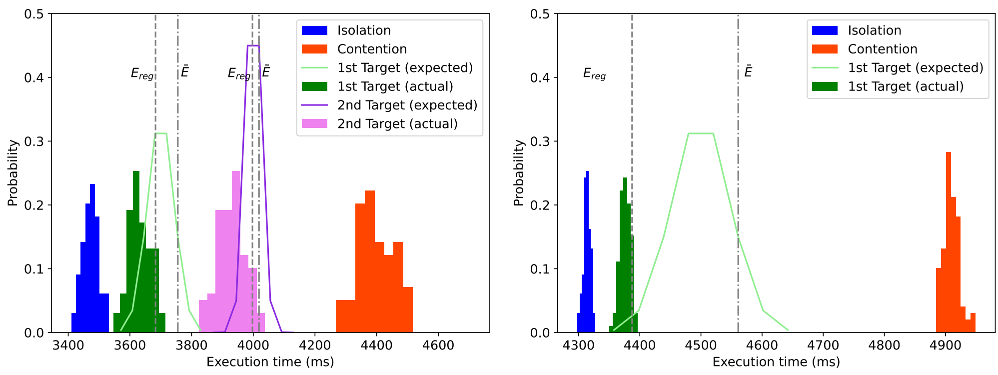
Configuring the proper system parameters is one of the primary challenges system designers face when implementing any regulating mechanism. In this subsection, we explain the key design parameters of our approach and the rationale behind their selection.

7.4.1 Regulation Interval

The choice of the regulation interval T_r is a trade-off between regulation granularity and overhead due to the generation of more frequent timer interrupts. The smaller regulation granularity is beneficial for finer grain control over the enforcement of our regulator. A regulation interval $T_r = 1$ ms has shown to yield good results and is set throughout the evaluation setup.

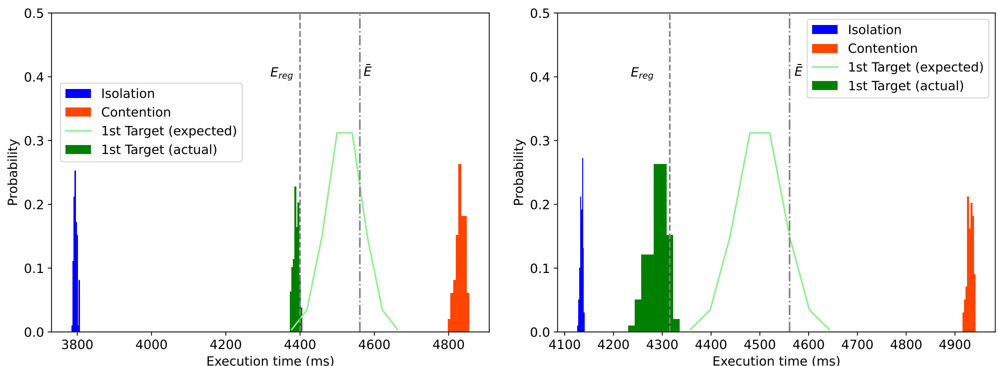
7.4.2 Total Bins

The number of bins defines the quantization that can be used to approximate the memory latency distribution. The PMU present in our evaluation platform offers 10 hardware counters as described in Section 6.1, which can be accordingly used to set 10 latency bins. However, we only dedicate 8 hardware counters for measuring memory latency distribution, resulting in 8 bins. The other two hardware counters are reserved for the purposes of (1) measuring the total number of read transactions as well as (2) the total read latency. This is done to validate the key system assumptions that are specified in Section 3.



(a) *disparity* on RT core and *MemBomb* on NRT cores.

(b) *tracking* on RT core and *MemBomb* on NRT cores.



(c) *mser* on RT core and *MemBomb* on NRT cores.

(d) *ndt_mapping* on RT core and *MemBomb* on NRT cores.

Figure 8 Execution Time Distribution (for 100 runs each).

7.4.3 Bin Size

The bin size of the memory latency distribution needs to be chosen in such a way that all possible individual read memory latencies can be covered while ensuring that distribution shifts can be effectively captured. Simultaneously, one must ensure that the bins are equally spaced and without discontinuities to provide a well-formed distribution snapshot when sampled. To determine the appropriate bin size, the APM is initially configured to measure the minimum and maximum memory latency values during a set of application runs. We observed a minimum read latency of 38 clock cycles and a maximum of approximately 600 clock cycles. Based on these values, we fix 40 clock cycles as the bin size. We also experimented with a larger bin size of 75 clock cycles with the same setup as shown in Figure 6b, which resulted in nearly empty bins with memory latency values greater than 375 clock cycles, as seen in Figure 7. We set the upper limit of the last bin to 2000 clock cycles in order to capture all conceivable memory latencies that a memory transaction may encounter.

7.5 Effectiveness of the Approach

The objective of this experiment is to show that, given $\bar{E}$ and α, Eq. 1 holds. Figure 8 summarizes the execution time distribution of applications during 100 runs and compares the target execution time $\bar{E}$ against the actual execution time E_{reg}. As a point of reference, the

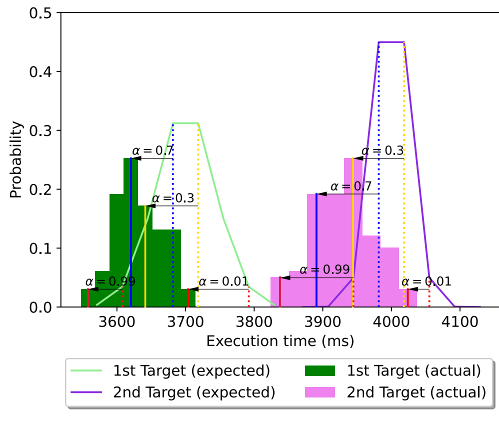


Figure 9 Validation of timeliness objective for various values of acceptable error α.

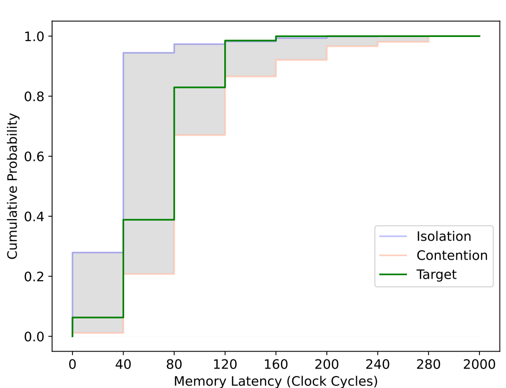


Figure 10 CDF for *disparity* on RT core.

execution time distribution in isolation (blue) and contention (orange) are also provided. The expected discretized execution time distribution of the target execution time is theoretically computed and is depicted as a discretized bell curve, whereas the actual execution time distribution is experimentally evaluated and depicted by a bar plot.

Figure 8a presents the target execution time $\bar{E}$ of 3755 ms and 4018 ms with an acceptable error $\alpha = 0.10\%$ for 1st and 2nd target execution times, respectively. Notably, as there are multiple possible normal distributions for a given $\bar{E}$, we fix $\sigma = \sqrt{\frac{\mu}{2}}$ and $\sigma = \sqrt{\frac{\mu}{6}}$ for the 1st and 2nd $\bar{E}$, respectively and then find the corresponding mean μ that is evaluated to 3700 ms and 4000 ms, respectively. Lowering/rising the standard deviation σ only narrows/widens the normal distribution curve and thus controls the tightness of the timeliness objective. The actual execution time E_{reg} for the given α is 3683 ms and 3997 ms , respectively and less than the target execution time. Hence the timeliness goal defined in Equation (1) is satisfied.

In order to validate the applicability of the approach for diverse workloads, we applied the same methodology to a number of different applications. We considered *tracking*, *mser* and *ndt_mapping* to be RT applications hosted on the RT core, while *MemBomb* running on the three NRT cores, as shown in Figure 8b, Figure 8c and Figure 8d, respectively. We use the same target execution time $\bar{E}$ of 4560 ms with an acceptable error $\alpha = 0.10\%$ for all three sets of experiments. Also, we use the same $\sigma = \sqrt{\frac{\mu}{2}}$. The actual execution time E_{reg} for *tracking*, *mser* and *ndt_mapping* was measured as 4387 ms, 4399 ms and 4315 ms, respectively, which is less than the target execution time and hence satisfies the timeliness goal defined in Equation (1).

Figure 9 shows the validation of the timeliness objective for various values of α for the same set of applications and experimental setup used in Figure 8a. We consider four values of α: 0.01, 0.3, 0.7 and 0.99. These are applied to both the expected and achieved target execution time distribution and highlighted by dashed and solid lines, respectively, in Figure 9. We found out that, for any value of α, the criteria $E_{reg} < \bar{E}$ holds. This provides empirical evidence to corroborate our expectation that the timeliness constraint formula presented in Equation (1) indeed holds for arbitrary values of α.

Finally, we illustrate the CDF of read memory latency observed by the *dispartiy* application in isolation and under contention, as well as the enforced reference CDF. The reference CDF $\bar{F}$ used in Figure 8 for the 1st target execution time $\bar{E}$ of 3755 ms is highlighted with green lines in Figure 10. The CDF in isolation (blue lines) and contention (orange lines) are

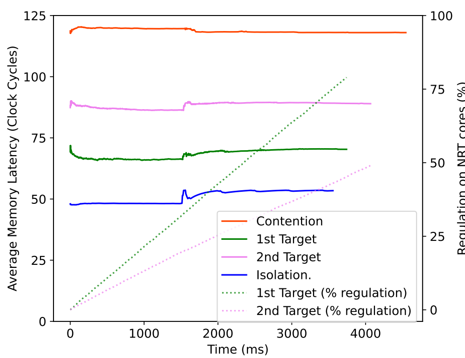


Figure 11 Impact of regulation on the average memory latency for *disparity* on RT core.

computed using the same PMFs previously shown in Figure 6. It can be noted that the $\bar{F}(k)$ computed for each bin k lies between the envelope defined by the CDFs measured in isolation (upper bound) and under contention (lower bound). These $\bar{F}(k)$ values are subsequently used by the memory latency distribution-driven regulator (Algorithm 1) to achieve an execution time E_{reg} that meets the timeliness objective.

7.6 Impact of Regulation on the Average Memory Latency

The selection of the target execution time $\bar{E}$ impacts the aggressiveness of the regulation, which in turn affects the average memory latency of an application. The average memory latency is defined as the total read memory latency divided by total number of read memory transactions over the $T_r = 1$ ms regulation interval.

The average memory latency of *disparity* under the same experimental setup as in Figure 8a is shown in Figure 11. However, instead of presenting the average memory latency over 100 runs, we present the WCET case: where the observed execution time is the highest.

It can be observed that the average memory latency for the 1st target execution time, with an observed WCET of 3714 ms, is around 70 clock cycles. For the 2nd target with an observed WCET of 4037 ms, the average memory latency is around 90 clock cycles. Consequently, the average memory latency is proportional to the target execution time $\bar{E}$.

As the target execution time for the 2nd target is more relaxed relative to the 1st target, the overall percentage of regulation that is enforced on the NRT cores decreases from 75% to 50% as seen in Figure 11. The percentage of regulation is calculated by dividing the total number of regulation intervals in which the NRT cores are suspended by the total number of regulation intervals in the experiment. Hence, the percent regulation that is enforced on the NRT cores is inversely proportional to the target execution time $\bar{E}$.

It is worthwhile to note that traditional DRAM bandwidth management-based regulation mechanisms [5, 23, 41] tend to bring the actual execution time as close as possible to the isolation scenario. However, our approach allows for the actual execution time to be anywhere between the execution time in contention to isolation.

7.7 Comparison with DRAM bandwidth-based regulation

To demonstrate that distribution-driven regulation is more beneficial than traditional DRAM bandwidth-based regulation mechanisms, we compare the slowdown ratio experienced by the applications running under the following scenarios (1) unregulated execution, in which the applications are running in parallel on their respective cores without any regulation

Table 3 Slowdown Ratio of benchmarks in contention without regulation and with different regulation mechanisms.

RT Core			NRT Cores		
disparity			*MemBomb on each NRT Core*		
Unregulated	MemGuard	Distribution-Driven	Unregulated	MemGuard	Distribution-Driven
1.28	1.03	1.03	3.79	16.67	7.05
disparity			*MemBomb (HB) on each NRT Core*		
1.25	1.03	1.03	1.41	8.07	3.49

mechanism, (2) a memory bandwidth management-based regulation (MemGuard [5])[1], and (3) distribution-driven regulation. We define the slowdown ratio of an application as *the ratio of execution time under contention to the execution time in isolation*.

We use the latest implementation of MemGuard [5] that regulates LLC write-backs in addition to LLC misses, ported to the partitioning hypervisor and configured for static bandwidth reservation. The key parameter used by MemGuard is the guaranteed (worst-case) bandwidth, which is approximately 960 MB/s for our evaluation platform based on the work in [24]. We allocated half of the said bandwidth for the application running in the RT core, and the remaining is distributed equally among the three applications running in the NRT cores.

Once the configurations for MemGuard have been selected, the parameters of the distribution-driven regulator (target execution time $\bar{E}$ and acceptable error α) are selected in such a way that the actual execution time E_{reg} for the application running on the RT core is the same under MemGuard and distribution-driven regulation. This allows for a fair comparison of slowdown ratios for applications running on NRT cores while keeping the same slowdown ratios for the application running on the RT core.

We conducted the evaluation with two different sets of applications. In the first set of applications, *disparity* is running on the designated RT core while synthetic *MemBomb* applications are running on the three NRT cores. In the second set of applications, only the *MemBomb* is modified to perform memory write operations for half of its duration periodically. We refer to this modified *MemBomb* application as *MemBomb Half Blast (HB)*.

Table 3 shows the slowdown ratios for different run settings compared to the execution times in isolation. We compare (1) unregulated runs in which the applications are executed concurrently in the respective cores with no regulation mechanism in place to (2) the proposed distribution-driven regulator and to (3) regulation done using MemGuard.

As expected, both regulation approaches achieve the same slowdown ratios of 1.03 for *disparity*. However, with MemGuard, both sets of applications running on the NRT cores suffer the highest slowdowns of 16.67 and 8.07, respectively. By contrast, the distribution-driven regulator is able to improve the slowdown ratio of the NRT applications on average by 2.2× compared to MemGuard.

8 Conclusion and Future Work

In this work, we presented a novel distribution-based regulation mechanism that enforces a *timeliness objective* formulated as a constraint on the probability of meeting any execution time target, which can be anywhere between the execution time in isolation and contention

[1] Comparison against a more recent work [23] is not possible due to the unavailability of memory utilization metric in our evaluation platform, which is necessary for the latter work.

scenario. The *timeliness objective* is met by directly controlling the distribution of total memory latency via regulation, which eventually impacts the distribution of the observed execution time.

We implemented our solution inside the Jailhouse-RT hypervisor [15] and deployed it on a COTS platform (Xilinx Ultrascale+ MPSoC) to demonstrate its effectiveness in meeting the *timeliness objective* for time-sensitive RT applications. Our approach can also be extended to handle multiple RT cores by assigning ranks to the RT cores based on their criticality level. The level of criticality then determines the order of suspension of the cores. If the observed CDF is below the reference CDF, the NRT cores are suspended first, followed by the RT core with the lowest criticality level, and so on, until the observed CDF no longer remains below the reference CDF. This is not immediately feasible with the same PMU due to the limited number of AXI ID filtering blocks. However, APM blocks can be instantiated on the on-chip FPGA, and memory traffic can be observed through-FPGA instead.

References

1. Ankit Agrawal, Renato Mancuso, Rodolfo Pellizzoni, and Gerhard Fohler. Analysis of Dynamic Memory Bandwidth Regulation in Multi-core Real-Time Systems. In *IEEE Real-Time Systems Symposium (RTSS)*, 2018.
2. Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: A predictable SDRAM memory controller. In *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2007.
3. ARM. An introduction to AMBA AXI. `https://developer.arm.com/documentation/102202`.
4. ARM. ARM® Cortex®-A53 MPCore Processor – Technical Reference Manual. `https://static.docs.arm.com/ddi0500/f/DDI0500.pdf`.
5. Michael Bechtel and Heechul Yun. Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.
6. Michael Bechtel and Heechul Yun. Cache Bank-Aware Denial-of-Service Attacks on Multicore ARM Processors. In *29th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2023)*, San Antonio, Texas, USA, May 2023.
7. Roberto Cavicchioli, Nicola Capodieci, and Marko Bertogna. Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms. In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2017.
8. Dakshina Dasari, Benny Akesson, Vincent Nélis, Muhammad Ali Awan, and Stefan M. Petters. Identifying the sources of unpredictability in COTS-based multicore systems. In *IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2013.
9. Giorgio Farina, Gautam Gala, Marcello Cinque, and Gerhard Fohler. Assessing Intel's Memory Bandwidth Allocation for resource limitation in real-time systems. In *IEEE International Symposium On Real-Time Distributed Computing (ISORC)*, 2022.
10. H. Fischer. *A History of the Central Limit Theorem: From Classical to Modern Probability Theory*. Sources and Studies in the History of Mathematics and Physical Sciences. Springer New York, 2010. URL: `https://books.google.com/books?id=v7kTwafIiPsC`.
11. Johannes Freitag and Sascha Uhrig. Closed Loop Controller for Multicore Real-Time Systems. In *Architecture of Computing Systems (ARCS)*, 2018.
12. Sebastian Hahn, Michael Jacobs, and Jan Reineke. Enabling Compositionality for Multicore Timing Analysis. In *International Conference on Real-Time Networks and Systems (RTNS)*, RTNS '16, 2016.
13. Denis Hoornaert, Shahin Roozkhosh, and Renato Mancuso. A Memory Scheduling Infrastructure for Multi-Core Systems with Re-Programmable Logic. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2021.

14 Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

15 J. Kiszka, V. Sinitsin, H. Schild, and contributors. Jailhouse Hypervisor. URL: https://github.com/siemens/jailhouse.

16 Tomasz Kloda, Marco Solieri, Renato Mancuso, Nicola Capodieci, Paolo Valente, and Marko Bertogna. Deterministic memory hierarchy and virtualization for modern multi-core embedded systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–14, 2019. doi:10.1109/RTAS.2019.00009.

17 D.S. Lemons, P. Langevin, and A. Gythiel. *An Introduction to Stochastic Processes in Physics*. Johns Hopkins Paperback. Johns Hopkins University Press, 2002. URL: https://books.google.com/books?id=Uw6YDkd_CXcC.

18 Claire Maiza, Hamza Rihani, Juan M. Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. *ACM Computing Surveys (CSUR*, 52(3):1–38, 2019.

19 Xiaosheng Mu, Luciano Pomatto, Philipp Strack, and Omer Tamuz. From blackwell dominance in large samples to renyi divergences and back again, 2019. doi:10.48550/arXiv.1906.02838.

20 Rodolfo Pellizzoni and Heechul Yun. Memory Servers for Multicore Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

21 Falk Rehm, Jörg Seitter, Jan-Peter Larsson, Selma Saidi, Giovanni Stea, Raffaele Zippo, Dirk Ziegenbein, Matteo Andreozzi, and Arne Hamann. The road towards predictable automotive high - performance platforms. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021.

22 Shahin Roozkhosh and Renato Mancuso. The potential of programmable logic in the middle: Cache bleaching. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 296–309, 2020. doi:10.1109/RTAS48715.2020.00006.

23 Ahsan Saeed, Dakshina Dasari, Dirk Ziegenbein, Varun Rajasekaran, Falk Rehm, Michael Pressler, Arne Hamann, Daniel Mueller-Gritschneder, Andreas Gerstlauer, and Ulf Schlichtmann. Memory Utilization-Based Dynamic Bandwidth Regulation for Temporal Isolation in Multi-Cores. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022.

24 Gero Schwäricke, Rohan Tabish, Rodolfo Pellizzoni, Renato Mancuso, Andrea Bastoni, Alexander Zuepke, and Marco Caccamo. A Real-Time Virtio-Based Framework for Predictable Inter-VM Communication. In *IEEE Real-Time Systems Symposium (RTSS)*, 2021.

25 Alejandro Serrano-Cases, Juan M. Reina, Jaume Abella, Enrico Mezzetti, and Francisco J. Cazorla. Leveraging Hardware QoS to Control Contention in the Xilinx Zynq UltraScale+ MPSoC. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2021.

26 Moshe Shaked and J. George Shanthikumar, editors. *Stochastic Orders*. Springer New York, 2007. doi:10.1007/978-0-387-34675-5.

27 P. Sohal, R. Tabish, U. Drepper, and R. Mancuso. E-WarP: A System-wide Framework for Memory Bandwidth Profiling and Management. In *IEEE Real-Time Systems Symposium (RTSS)*, 2020.

28 Parul Sohal, Michael Bechtel, Renato Mancuso, Heechul Yun, and Orran Krieger. A Closer Look at Intel Resource Director Technology (RDT). In *International Conference on Real-Time Networks and Systems (RTNS)*, 2022.

29 Parul Sohal, Rohan Tabish, Ulrich Drepper, and Renato Mancuso. Profile-driven memory bandwidth management for accelerators and cpus in qos-enabled platforms. *Real-Time Syst.*, 58(3):235–274, September 2022. doi:10.1007/s11241-022-09382-x.

30 Lukas Sommer, Florian Stock, Leonardo Solis-Vasquez, and Andreas Koch. DAPHNE – An automotive benchmark suite for parallel programming models on embedded heterogeneous platforms: work-in-progress. In *International Conference on Embedded Software Companion (EMSOFT)*, 2019.

31 Ashley Stevens. Quality of Service (QoS) in ARM Systems: An Overview. In *ARM White paper*, 2014.

32 Meltem Sönmez Turan, Elaine Barker, John Kelsey, Kerry McKay, Mary Baish, and Michael Boyle. Recommendation for the Entropy Sources Used for Random Bit Generation, 2018. URL: `https://csrc.nist.gov/publications/detail/sp/800-90b/final`.

33 P. K. Valsan, H. Yun, and F. Farshchi. Taming Non-Blocking Caches to Improve Isolation in Multicore Real-Time Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

34 Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Addressing Isolation Challenges of Non-Blocking Caches for Multicore Real-Time Systems. *ACM Real-Time Systems*, 53(5):673–708, 2017.

35 S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. SD-VBS: The San Diego Vision Benchmark Suite. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

36 Bryan C. Ward, Jonathan L. Herman, Christopher J. Kenna, and James H. Anderson. Outstanding Paper Award: Making Shared Caches More Predictable on Multicore Platforms. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

37 Xilinx. AXI Performance Monitor LogiCORE IP Product Guide (PG037). `https://docs.xilinx.com/v/u/en-US/pg172-ila`.

38 Xilinx. AXI Traffic Generator v3.0 LogiCORE IP Product Guide (PG125). `https://docs.xilinx.com/v/u/en-US/pg125-axi-traffic-gen`.

39 Xilinx. Integrated Logic Analyzer v6.2 LogiCORE IP Product Guide (PG172). `https://docs.xilinx.com/v/u/en-US/pg037_axi_perf_mon`.

40 Xilinx. Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit. `https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html`.

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 (TC)*, 65(2):562–576, 2016.

42 Heechul Yun, Waqar Ali, Santosh Gondi, and Siddhartha Biswas. BWLOCK: A Dynamic Memory Access Control Framework for Soft Real-Time Applications on Multicore Platforms. *IEEE Transactions on Computers (TC)*, 66(7):1247–1252, 2017.

43 Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards Practical Page Coloring-Based Multicore Cache Management. In *ACM European Conference on Computer Systems*, EuroSys '09, 2009.

44 Matteo Zini, Daniel Casini, and Alessandro Biondi. Analyzing Arm's MPAM From the Perspective of Time Predictability. *IEEE Transactions on Computers (TC)*, 72(1):168–182, 2023.

45 Matteo Zini, Giorgiomaria Cicero, Daniel Casini, and Alessandro Biondi. Profiling and controlling I/O-related memory contention in COTS heterogeneous platforms. *Software: Practice and Experience*, 52(5):1095–1113, 2022.

46 Alexander Zuepke, Andrea Bastoni, Weifan Chen, Marco Caccamo, and Renato Mancuso. MemPol: Policing Core Memory Bandwidth from Outside of the Cores. In *29th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2023)*, San Antonio, Texas, USA, May 2023.

Quasi Isolation QoS Setups to Control MPSoC Contention in Integrated Software Architectures

Sergio Garcia-Esteban
Polytechnic University of Catalonia, Barcelona, Spain
Barcelona Supercomputing Center (BSC), Spain

Alejandro Serrano-Cases
Barcelona Supercomputing Center (BSC), Spain
Rapita Systems S.L., Barcelona, Spain

Jaume Abella
Barcelona Supercomputing Center (BSC), Spain

Enrico Mezzetti
Barcelona Supercomputing Center (BSC), Spain
Rapita Systems S.L., Barcelona, Spain

Francisco J. Cazorla
Barcelona Supercomputing Center (BSC), Spain
Rapita Systems S.L., Barcelona, Spain

Abstract

The use of integrated architectures, such as integrated modular avionics (IMA) in avionics, IMA-SP in space, and AUTOSAR in automotive, running on Multi-Processor System-on-Chip (MPSoC) is on the rise. Timing isolation among the different software partitions or applications thereof in an integrated architecture is key to simplifying software integration and its timing validation by ensuring the performance of each partition has no or very limited impact on others despite they share MPSoC's hardware resources. In this work, we contend that the increasing hardware support for Quality of Service (QoS) guarantees in modern MPSoCs can be leveraged via specific setups to provide strong, albeit not full, isolation among different software partitions. We introduce the concept of Quasi Isolation QoS (QIQoS) setups and instantiate it in the Xilinx Zynq UltraScale+. To that end, out of the millions of setups offered by the different QoS mechanisms, we identify specific QoS configurations that isolate the traffic of time-critical software partitions executing in the core cluster from that generated by contender partitions in the programmable logic. Our results show that the selected isolation setup results in performance variations of the partitions run in the computing cores that are below 6 percentage points, even under scenarios with extremely high traffic coming from the programmable logic.

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

Keywords and phrases Multicore, Interference, QoS

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.5

Funding This work has been supported by Grant PID2019-107255GB-C21 funded by MCIN/AEI/10.13039/501100011033 and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 772773).

1 Introduction

MPSoCs are progressively used in safety-critical domains, like avionics and space, to cater for augmented performance requirements. Besides the sheer computational power they provide, MPSoCs increasingly incorporate substantial Quality of Service (QoS) features [60, 44]. QoS

© Sergio Garcia-Esteban, Alejandro Serrano-Cases, Jaume Abella, Enrico Mezzetti, and Francisco J. Cazorla;
licensed under Creative Commons License CC-BY 4.0

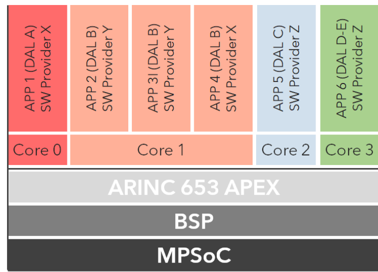


Figure 1 IMA setup with multiple applications and Design Assurance Levels.

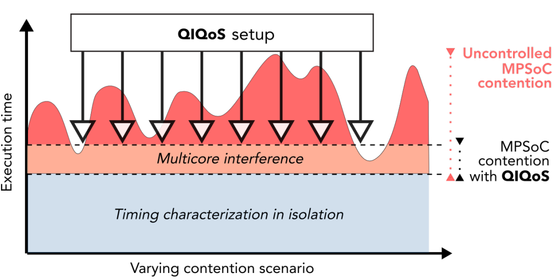


Figure 2 Representation of interference mitigation impact under a QIQoS setup.

support was originally designed for performance optimization and load balancing, but is increasingly considered as a means to better control contention effects among co-running tasks and enforce a more deterministic timing behavior [48, 50]. QoS support is provided as software-controlled "knobs" that allow to intentionally bias the execution towards a given task by providing it with privileged access to hardware shared resources, more bandwidth in an interconnect, reserved space in stateful hardware resources, and the like.

When an MPSoC is used to support *exclusively a monolithic application* using several or all underlying cores (e.g. a single ARINC 653 partition), deployed for example by a single avionics original equipment manufacturer or TIER1 provider, the contention each process can suffer can be bounded already in early software development stages: contender information is available as they are part of the same monolithic application. In such scenario, different QoS setups may be explored to find a setup that satisfies the performance requirements of all processes [48].

However, this is seldom the case and instead an *IMA-MPSoC setup* is deployed: multiple applications from different software providers are typically integrated on the same target MPSoC, with a view to reducing integration and validation costs. In this line, integrated architectures [1] like IMA [12] build on time and space partitioning concepts, as defined by the ARINC 653 [4] open standard, to simplify the allocation of computing resources to different applications. This effectively allows integrating several applications onto less computing hardware, as illustrated in Figure 1. While IMA time partitioning ensures each application receives a given amount of CPU time, the actual progress made by an application in the time window also depends on the share of hardware resources the application receives.

MPSoC timing interference arises on contending accesses to shared hardware resources [19, 39, 55], which causes an application's execution time to depend on the use of resources made by applications in other software partitions. Waiting until late development and integration stages, when all providers make their applications available, to address the timing dimension and assess timing interference has opposing effects. On the positive side, only the system's intended final configuration [1, 18, 21] is considered, which reduces the risk of overestimating the contention impact. On the negative side, there is a non-negligible risk of detecting software timing violations too late in the validation and verification stages, when reacting to time misconfigurations can result in unaffordable changes to the applications, system design and schedule, and the entailed regression testing.

[1] For instance, Integrated Modular Avionics (IMA) for avionics, IMA for Spacecrafts (IMA-SP) [56] in space, and AUTomotive Open System ARchitecture (AUTOSAR) [13].

In IMA-MPSoC scenarios, QoS support can be leveraged to anticipate late timing issues related to multicore contention by preserving partition-level time budgets against the interference of (unknown or partially known) contender applications. This is, in fact, aligned with CAST-32A [18] advisory circular and A(M)C 20-193 [21] for multicore certification in the avionics domain, establishing that separate determination of the WCET of an application, without any other applications executing, is only valid if the applicant can demonstrate that they build on a multicore Platform with robust partitioning or that time interference from other applications is avoided or mitigated for that application.

Concept. We contend that existing hardware QoS solutions can be used to enforce setups under which timing interference is mitigated and the execution time of an application is less exposed to contention from other co-running applications. We aim at devising a set of QoS setups, which we call Quasi Isolation QoS setups or QIQoS, that guarantee a high-degree of isolation (performance guarantees) to the applications regardless of the contention its co-runners put on MPSoC's hardware shared resources, hence meeting IMA assumptions and requirements. QIQoS setups are meant to reduce the impact on execution time when varying the contender loads on the system shared resources. Figure 2 illustrates the effects of a given QIQoS setup in reducing the sensitivity to contention scenario by enforcing an upper bound to the incurred timing interference. As an immediate effect, the enforcement of a proper QIQoS setup will *increase the representativeness of early time budgets*, typically obtained by running the application against synthetic aggressors, making them much tighter and stable. The gap between the multicore interference empirically observed during the timing verification campaign by deploying synthetic worst-case contention scenarios and that observed in the final system configuration will be sensibly narrowed.

Realization. In this work, we instantiate our QIQoS approach on the Xilinx Zynq UltraScale+ [60, 59] to address contention arising on accesses to the DDR main memory. The DDR memory controller (DDRMC) provides DDR access to the different computing elements on the MPSoC through its six ports, and offers a complex multi-layer QoS mechanism to control the traffic coming through each port. This includes traffic classes, port throttling, and per-traffic class resource allocation. Hence, the QoS of the DDRMC allows millions of configurations that are remarkably challenging to master for the end users [48]. Our first step is then, identifying several specific DDRMC QIQoS configurations that allow, to different degrees, isolating DDR traffic of critical tasks from that generated by other tasks. Subsequently, we expose the specific characteristics of the identified QIQoS from the standpoint of the type and degree of isolation they can assure.

When facing the increasing QoS support in modern MPSoCs, the challenge lies in making effective and consistent use of the available features. One of the proposed QIQoS setups exploits the timeout features at the port level, while the second QIQoS instead mainly builds on starvation prevention features. Both QIQoS setups use traffic classes and transaction throttling. Other common features to both include preventing specific settings for the DDRMC that, if wrongly set, could defy the benefits achieved through the QoS layer. These include limiting the allowed DDR memory commands that can be sent out of order to the DDR device, and preventing critical and non-critical tasks from sharing the same memory port.

Our results show that the proposed DDRMC QIQoS configurations can effectively isolate the execution of the critical tasks that run in the A53 and the R5 cores from the load coming from the programmable logic (PL) in the access to memory, despite the latter produces in our

experiments huge amounts of DDR accesses. The degree of isolation can be configured based on a set of parameters provided for each QIQoS setup, which allows end users to achieve the desired balance between isolation and performance. Under the most aggressive QIQoS setups, the performance variation across substantially different loads that the PL put on the DDRMC is as low as 6 percentage points.

The rest of this paper is organized as follows. Section 2 presents the related works. Section 3 introduces our target platform. Section 4 presents the principles of QIQoS setups and describes their application on the target platform. Section 5 provides the experimental results. Finally, Section 6 presents the main conclusions of this work.

2 Related Works

Domain-specific safety standards and support documents specifically identify the need for controlling and reducing the impact of contention arising on shared resources accesses as a fundamental requirement for certification [1, 18, 21, 34]. Massive research efforts have been devoted to cope with the impact of multicore timing interference on system performance and predictability [46, 48]. Whereas custom hardware designs have been proposed to balance predictability and performance [42, 30, 54, 37], in this work we focus on a COTS MPSoC and analyze the predictability of its memory controller.

Several works pursue analytically bounding the worst-case interference suffered by each application [17, 19, 24, 39, 57]. Regardless of the tightness of the adopted method, contention can be too large (e.g. some works report more than 20x performance degradation [55]), which ultimately leads to severe system under-utilization. While we are still interested in analyzing the timing interference, our main focus is to enforce a-priori control of contention via appropriate QoS configurations.

Controlling the impact of contention by determining how resources are shared among applications, as opposed to just analyzing it, allows to reduce the impact of contention and is a fundamental enabler for the analyzability of multicore systems [48, 43, 28]. Time and space partitioning of shared resources has been explored building on top of existing hardware and software solutions, typically handled either as part of the static system configuration [4, 28], or as part of more or less complex run-time monitoring of resource usage at RTOS or Hypervisor level [43]. Software solutions, in particular, constraint applications usage of shared resources, like the last level cache and DRAM [29, 16], to predetermined quotas. In this work, we do not focus on specific run-time support, which is in fact dependent on the full software stack, but we only focus on exploiting the existing hardware QoS support. In fact, software-based are deployed on top of existing hardware support for QoS and need to be consistent and compatible with the underlying QoS configuration at the risk of obtaining suboptimal or even counter intuitive results. For example, while it is possible to regulate the memory bandwidth of each core by constraining read/write requests, for example with MemGuard [16], the actual operation of bandwidth regulation and prioritization mechanism can be jeopardized by a QoS configuration assigning low priority in the memory controller to the high bandwidth thread. Nonetheless, QoS features can also be leveraged to support and improve the effectiveness of more complex software-level paradigms.

Hardware providers are increasingly aware of the importance of regulating the impact of contention and are providing hardware level solutions for controlling and apportioning the usage of shared resources such as Intel Resource Director Technology (RDT) [33] and ARM Memory System Resource Partitioning and Monitoring (MPAM) [9]. Initial assessment of those solutions from the standpoint of timing predictability has been conducted in [49, 63]

arising some concerns on the effectiveness of design and implementations of such modules. A custom FPGA implementation to regulate memory accesses in the Zynq UltraScale+ platform has also been proposed in [32]: despite its effectiveness, the approach entails non-negligible performance overheads due to routing core DDR requests via the PL.

More recently, the availability of increasingly-powerful QoS features [6, 7] in COTS platforms has prompted the exploitation of QoS elements, traditionally used for performance balancing and tuning, as an effective means to control multicore timing interference [48]. An adequate degree of software controllability of such QoS features is also a mandatory requirement [22]. An initial study on the QoS features in a representative platform for the avionics domain has been reported [48], providing evidence of how the manifold QoS features provide sufficient malleability to enforce different resource sharing scenarios. While sharing with [48] the focus on the Zynq UltraScale+ memory controller, in this work we focus on defining quasi isolation envelops for critical tasks against non-critical activities generated from the PL rather than exploring different performance trade-offs among software partitions. In fact, while the work in [48] fits the monolithic application scenario, where the goal is finding a setup that satisfies the performance requirements of all processes, it cannot be used in IMA-MPSoC scenarios to increase the representativeness of early time budgets by devising a set of QoS setups that guarantee a high-degree of performance isolation.

Partial explorations of the impact of SMT-related QoS modules on execution time performance have been conducted in [15, 31] for IBM and Intel processors respectively. The QoS support in the UltraScale+ platform is partially addressed in [41] where a specific QoS setup for the memory controller is used to explore possible throughput configurations for the DDR memory module. These works are focusing on preventing performance degradation rather than exploiting contention control for enabling stronger performance guarantees.

Finally, some works attempt to analytically model the effect of configuring QoS features and contention regulation mechanisms in general. The work in [32] builds on profile-based analytical predictive models to enforce a bandwidth regulation policy, including the use of a set of QoS parameters to regulate traffic from the PL logic. An analytical model of the QoS-400 module in the Zynq UltraScale+ is analyzed in [64]. Despite the expected higher accuracy of analytical characterization approaches, they are often building on partial and potentially misleading information on the hardware modules they are meant to model [14], due to increasing hardware complexity and poor documentation. For this reason, they can be useful for deriving early time estimates, they cannot be generally considered a robust and generic alternative to more empirical approaches, as the one proposed in this work.

3 Target System

We focus on an MPSoC IMA setup in which a system integrator is in charge of deploying in the same computing platform multiple applications with different functional safety requirements from different software providers. These systems are typically scheduled following a layered approach where the first level, either provided by an executive layer or a hypervisor, is responsible for scheduling the different software partitions and each partition is in turn responsible for executing single applications or processes. However, no or limited information is available on software components from other providers unless in the very final stage of system development. In particular, we do not consider relatively simpler monolithic system designs where software components are responsibility of a single provider, and hence simultaneously available for timing analysis purposes, including the early characterization of contention impact and the exploration of different system configurations and QoS setups [45, 48].

In an MPSoC IMA scenario, instead, different software modules are incrementally made available and integrated into the final intended configuration. The lack of early and precise information on the different software modules requires the application of compositional approaches for the analysis of multicore timing interference. Hence, timing budgets can be consolidated before the whole system is available and do not need to be re-determined whenever a new component is integrated.

One practical approach for early characterization of timing interference consists in using synthetic 'aggressor' programs [35] during early stages to test representative contention scenarios despite some software modules may not be available. These synthetic applications can be exploited to generate high load on shared resources. Thus, by running each application against aggressors already in early development stages, early figures on the impact of high contention scenarios on the application's execution time can be produced. However, as the amount of shared resources in multicore processors used in embedded domains is constantly increasing, the impact of aggressors on application's execution time is potentially huge (20x and higher) [36, 55], resulting in overly pessimistic execution time budgets.

In this IMA-MPSoC scenario, tighter figures on contention impact of each module can only be provided by limiting the amount of interference possibly suffered by the application or module under analysis, regardless of (or with limited correlation with) the co-runners. In this work, we show how available QoS support in modern MPSoCs can be exploited to enable early consolidation of tight contention-aware time budgets by providing quasi-isolation scenarios where the impact of contention is bounded by specific QoS configurations.

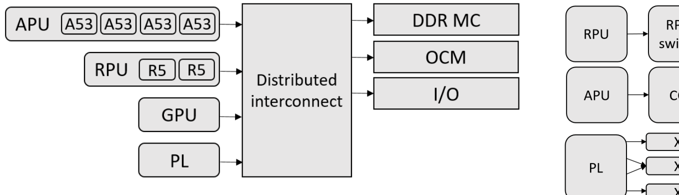
3.1 Introduction to the Zynq UltraScale+

The Zynq UltraScale+ MPSoC [60, 59] comprises four main types of hardware blocks: computing elements (CEs), communication elements, memory, and I/O controllers.

The CEs, see Figure 3(a), include a high-performance CPU cluster (called APU or application processing unit) comprising 4 Arm Cortex A53 cores [8], a real-time CPU cluster (called RPU or real-time processing unit) with 2 Arm Cortex R5 cores [10], an Arm Mali-400 GPU [11], and a programmable logic (PL) block that in real-time systems is usually deployed to synthesize components to support I/O or computing acceleration of some functionalities.

In terms of memories, the MPSoC includes an on-chip memory (OCM), the DDR SDRAM memory controller (DDRMC), and interfaces to access ROM and flash memories, which are generally not used for the normal operation of end-user applications and hence excluded from our discussions in the rest of the paper. The MPSoC also includes a complex I/O system that handles accesses to generic (e.g USB) and specific controllers (e.g CAN).

An Arm AXI-based distributed network orchestrates the communications among all elements – usually from (to) CE to (from) memory or I/O. It includes the cache-coherent interconnect (CCI) hardware block that controls aspects related to coherence and distributed memory; top-level switches like the RPU switch; and smaller or secondary switches – highlighted with an 'X' in Figure 3(b). The interconnect is heterogeneous and distributed meaning that the set of switches that each CE has to traverse to reach a given destination varies per CE. For instance, Figure 3(b) shows an abstraction of most relevant connection between the different elements in the system. It shows the interconnect IP blocks each CE has to traverse to reach the DDRMC. This path can be configured, e.g. the APU can use one or two ports to access memory, while the RPU can also send requests to the DDRMC via the CCI using a single DDRMC port.



(a) Computing elements (left), inter-connect (middle), and memories and I/O (right).

(b) Paths from computing elements to the DDR MC.

Figure 3 Different block diagrams of the Zynq UltraScale+ MPSoC.

3.2 Hardware support for QoS

The UltraScale+ offers a variety of hardware QoS mechanisms that help shaping the speed at which the requests are sent conveyed from source to destination. The most relevant ones are:

Static QoS. Every AXI request in a point-to-point communication is tagged with a QoS value (AXQoS) from 0 to 15 that the target of the communication can use to prioritize it (the higher the AXQoS value, the higher the priority).

Dynamic QoS. Different interconnect elements, when they can receive requests from different sources, can apply mechanisms that dynamically adjust the static QoS value to reach a given target metric like controlling the maximum number of outstanding requests.

QVN. QoS virtual networks (QVN) use tokens to control transaction flows to ensure that a transaction can always be accepted at its destination before it is sent by a source.

DDRMC QoS. The QoS at the memory controller offers a complex multilayered prioritization system that we analyze more in detail in the following section.

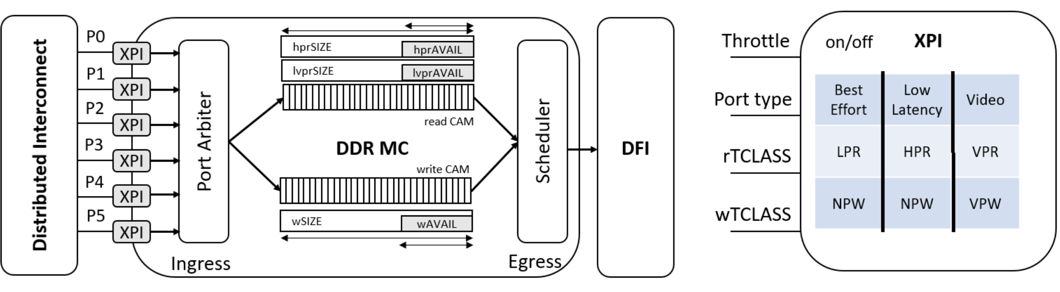
QoS on the interconnect is especially relevant when dealing with request flows from different CEs to different destinations, like DDRMC and I/O [64, 50]. However, when different CEs target the same destination, the speed at which the target processes the requests of each flow is the main factor determining the QoS each flow receives. In this work, we focus on memory contention on the DDR memory as it is one of the major bottlenecks in real-time systems [36, 55]. In fact, it is the last arbitration tier in charge of contain contention when other mechanism failed to provide isolation or timing guarantees. This trend continues [38] as more AI software is used in real-time systems processing huge amounts of data coming from different sensors like video cameras and radars. Hence, in this work, we focus on the QoS in the DDRMC, fix the same static QoS for all requests, and disable all dynamic QoS and QVN mechanisms.

3.3 The DDRMC in the Zynq UltraScale+

The DDR memory subsystem encompasses the DDR memory controller (DDRMC) – which comprises a DDR QoS module – and the DDR Physical Interface (DFI), see Figure 4. The former receives the AXI requests from the distributed interconnect via six different ports and converts them into DDR commands. The latter translates the DDR commands into signals to the external DDR3/4 compliant device.

3.3.1 DDRMC

The DDRMC dynamic scheduling optimizes bandwidth and latency using a programmable QoS controller. Traffic (i.e. flows of AXI requests) arrives to the DDRMC via the six AXI ports (XPI), referred to as Pi in Figure 3(b).



(a) Main QoS-related components of the DDRMC. (b) XPI configurations.

Figure 4 Block diagram of the DDR Memory subsystem in the Zynq UltraScale+.

Traffic Classes. Reads are classified into low, high, or video traffic classes (*LPR*, *HPR*, and *VPR*). Meanwhile writes into low or normal ($LPW{=}NPW$) and video (VPW).

Expired commands. VPR/VPW commands behave as low priority when they have not *expired* (i.e. there is not a transaction timeout). Once expired, they are promoted to a priority higher than that of the HPR/NPW commands. The timeout period for VPR and VPW transactions ($RTOUT$ and $WTOUT$, respectively) is configured via control registers.

CAMs. DDR commands (or simply commands) translated from incoming AXI requests are stored into the content addressable memories ($CAMs$) both *shared by all ports*.

- A 64-entry write CAM or $wCAM$ that is shared by all traffic classes (and all ports).
- A 64-entry read CAM or $rCAM$ split into two partitions based on traffic type.
  - The first partition ($hpr\ rCAM$) is used for HPR traffic classes.
  - The second partition ($lvpr\ rCAM$) is used for LPR/VPR traffic classes.

There is a single rCAM so that all ports with HPR traffic share the $hpr\ rCAM$ and all ports with LPR/VPR traffic share the $lvpr\ rCAM$. The size of the $hpr\ rCAM$ and $lvpr\ rCAM$ is controlled by configuration registers: *hprSIZE* and *lvprSIZE* respectively. Each partition can be configured to have from 1 to 64 entries, with the constraint that their addition must be equal to the $rCAM$ size, 64. Hence, for instance, if $hprSIZE = 24$ then $lvprSIZE = 40$.

CAM allocation is performed by the Port Arbiter (PA) that selects from all DDR ports the command to issue to the CAMs based on several levels of arbitration.

1. Reads are prioritized while there are VPR expired, or there are reads and no expired VPW. Writes are served when there are no reads, and if there are expired VPW and no expired VPR. The expiration period can be configured via setting timeouts for VPR/VPW.
2. HPR traffic has higher priority than LPR/VPR on the read channel and NPW/VPW has the same priority on the write channel, with VPR/VPW prioritized if they time out.
3. Priorities are given on per-command based on their static QoS (AXQOS signals).
4. Conflicts are resolved using round-robin arbitration.

Port throttling changes this behavior by throttling ports that has their throttle-enable control register set, when certain occupancy-related conditions are met:

- When the available entries in the $hpr\ rCAM$ partition are below an availability threshold, set via a control register ($hprAVAIL$), those ports with read traffic mapped to the HPR class can be throttled, if their port hpr throttling is enabled.

- Likewise, when the number of available entries in the *lvpr rCAM* is below an availability threshold (*lvprAVAIL*), those ports with read traffic mapped to the LPR class can also be throttled, if their port lpr throttling is enabled.
- When the count of wCAM available entries is below *wAVAIL*, those ports with write traffic mapped to the LPR=VPR class can also be throttled, if their port write throttling is enabled.

Note that ports with traffic mapped to the VPR/VPW class cannot be throttled. Naturally, and beyond the port throttling control, all ports are stalled when any CAM or any other internal resource of the DDRMC is exhausted.

Other. There are other DDRMC features that we have not used because their interaction with the ones we actually use is hard to control, as described in Section 4. These features are port aging (that moves a port to the highest priority when an outstanding request is not served after the established time); urgent transactions (indicating that there is a read/write urgent transaction); and regions, defined at port level to help mapping AXI static priorities and traffic classes.

3.3.2 DFI

When issuing commands from the CAMs to the DFI, command reordering is allowed to favor page hits, potentially causing out-of-order execution of the commands. A regulator limits the issue to up to 4 out-of-order commands. When it is disabled, no restriction is applied, resulting in no control over the number of out-of-order commands executed.

CAM deallocation. On the egress side of the CAMs, depicted as scheduler in Figure 4(a), the SoC allows setting a maximum starve period that a CAM can be without issuing a command to the DFI. There is one period per CAM (*hpr rCAM*, *lvpr rCAM*, and *wCAM*) before the queue goes into a "critical" state and it gets priority to send commands to the DFI. Note that reduced starving periods increase the switching among queues.

3.4 Software Partition Setup

The most natural and efficient way to use the Zynq UltraScale+ in IMA-MPSoC real-time systems is by consolidating different *software partitions* (SWP), ensuring that the needed mechanisms to simplify integration and testing are in place. All main real-time operating systems and hypervisors build on the concept of separation kernels that enable different software partitions to achieve the required safety and security goals. Examples include Lynx Secure [40] and DDC-I Deos [20], which are compliant with the highest-criticality levels in Avionics, i.e. DAL A in DO-178C [47]. SWPs are usually executed in a disjoint set of the available CEs in the underlying platform to reduce timing interactions among them, see Figure 1. In the Zynq UltraScale+, a SWP can span from using a single R5 or A53 core to use a subset of the R5 cores, A53 cores and integrate some acceleration in the PL.

To perform a reasoned exploration of configuration setups, we define the application deployment scenarios (ADS) shown in Table 1. We focus on two classes of applications, depending on whether they comprise critical tasks (CT) or not (NCT). In each ADS, we assume up to two SWPs with CT applications being deployed in the CPU clusters and NCT ones in the PL. Applications are assumed to be independent or share data via predictable communication channels [4]. The goal is to isolate the performance of the applications of the critical SWPs from the traffic coming from the PL, so that they can be analyzed in isolation.

Table 1 SWP active in each core under each ADS.

	R5	R5	A53	A53
ADS1	SWP1	–	–	–
ADS2	–	–	SWP1	–
ADS3	SWP1	–	SWP1	–
ADS4	–	–	SWP1	SWP2
ADS5	SWP1	SWP2	SWP1	SWP2

Other deployment scenarios are possible in which the PL is not used and the NCT runs, for instance, in the A53 cores. In that case, the NCT running in the A53 can be mapped to one port (e.g., P1), while the CT in the R5 can use a different port (e.g., P2), see Figure 3(b). This port selection is a configurable option [48]. In that scenario, the very same principles we describe in this work apply. In fact, the scenarios we address, with the NCT running in the PL, are more challenging since both, the R5 and A53, limit the number of requests in-flight, which further limits the pressure on memory. In particular, the R5 cores allow one in-flight load/store per core and the A53 allows a maximum of 3 in-flight loads per core. Instead, the PL can exploit more ports to memory, and we are able to instantiate several AXI Traffic Generators per port, allowing many more independent requests, hence resulting in higher pressure on memory (more details on the setup are provided in Section 5.1).

4 Quasi Isolation QoS Setups

4.1 Context and Approach

Meeting safety standards requirements against the complexity of current and upcoming MPSoCs is a challenging endeavor. Evidence must be provided that the contention tasks generate on each other is anticipated and controlled. Unfortunately, such evidence cannot be derived by acquiring full information about hardware behavior to build a comprehensive and accurate model on how resources are shared among co-running tasks. While reasonable, such approach is deeply invalidated in practice.

On the one hand, it is extremely improbable, if at all possible, that IP providers give full access to the currently-confidential technical documentation as required to derive detailed contention models. The number of examples in this direction are endless: from the very limited information on NoCs (e.g. the functional behavior of the Arm NIC-400 [5] is limited to few pages and, similarly, the description provided by NXP of its CoreNet Coherence Fabric in the T2080 TRM [26] is minimal and includes no information about its internal behavior in terms of buffering or prioritization), to the almost non-existent information about GPU timing behavior [11]. Such trend is not expected to change in the near future, with recent architectures like the NXP LX2160 [44] and the Xilinx Versal [62] equipping increasingly complex components with increasingly limited descriptions of their functional and timing behavior. The Zynq UltraScale+, target of this work, is not an exception: even just the memory controller exhibits several levels of prioritization (see Section 3.3.1) and there is not available information to derive the exact way the scheduler sends requests to the DFI, how exactly the drain of the CAMs occurs when a CAM goes critical, and many other details.

On the other hand, even if enough information was available, modern MPSoCs include a score of dynamic features that make modeling extremely hard without resorting to overly-pessimistic (conservative) assumptions. The Zynq UltraScale+ is a clear example of such scenario as it includes a relevant set of dynamic features that are triggered based on the

(dynamic) behavior of the traffic. Dynamic features include, among others the read and write timeout feature, which depends on how long a command is waiting until being served; the port throttling mechanism, which is triggered based on the availability of the partitions of the rCAM; the wCAM, which in turn, depends on the (dynamic) traffic coming via the different port to the DDRMC; and the prioritization mechanism in the PA, which builds on the status of expired/not expired commands.

Overall, while several reverse engineering and characterization of specific features have been carried out (from SMT processors, cache, memory, and GPUs), the complexity of the current MPSoCs, the limited information, and the number of different IP components intervening in the computation and communication, prevent the definition of a precise contention-aware functional and timing model in practice.

Hence, the challenge for the real-time research community and original equipment manufacturers and TIER1/2 companies in critical domains is to make the system as safe as possible building on the (limited) available information supported by empirical evidence. The remaining uncertainty (risks) are covered by specific mechanisms defined in safety standards like *safety nets* that can assume control of the system if the main MPSoC fails either in terms of hardware reliability or in terms of execution time violations [21].

4.2 Concepts and Benefits

Building on the considerations above, in this work, we do not attempt to develop a model that describes how the different QoS mechanisms work and make predictions for other applications, or how the current application would behave under a different QoS setup. Instead, in our target IMA-MPSoC setup (see Section 3.3.1) we aim at enforcing a high-degree of performance isolation. A Quasi Isolation QoS setup, QIQoS for short, is a particular configuration (setup) of the QoS mechanisms in the underlying platform that provides performance guarantees to a particular set of tasks running in different CEs. That is, QIQoS helps containing the impact that MPSoC contention generated by the NCT can have on the CT, and hence makes early time budgets more representative under a set of QoS setups.

We assess the quality of a specific QIQoS along three main axes: first, for performance guarantees, we look at the slowdown of the CT (the lower the better); second, for timing predictability, we evaluate CT's performance variability when run against different NCT (the lower the better), for instance different aggressor benchmarks that put variable high load on the shared resources; finally, for overall performance and fairness, we consider the average performance of the NCT (the higher the better). This last dimension can be used as a tie breaker among comparable QIQoS setups. We will formalize relevant metrics for evaluating QIQoS setups in Section 5.2.1.

When deployed, a QIQoS setup simplifies incremental integration by reducing the contention impact that co-runners (NCT) can have on the analysis tasks (CT). QIQoS allows deriving timing budgets that are robust against contention scenarios and do not build on any specific run-time support on commercial MPSoCs. Both aspects together are fundamental enablers for the integration and reuse of software modules from different vendors in mixed-criticality systems. The achieved quasi-isolation guarantees the applications' timing behavior, partially consolidated in the early development stages, will be confirmed at integration, reducing the risk of unexpected timing misbehavior and commercially disruptive rollbacks. The proposed QIQoS approach contrasts with previous works that assuming that both CT and NCT are known and focus on exploring different QoS setup that satisfies performance requirements [48]. That is while QIQoS focus on IMA-MPSoC setups, previous works [48] target a monolithic application as presented in Section 1.

4.3 Application to the DDRMC

We instantiate the QIQoS approach on the DDRMC by developing two specific and well-justified QoS isolation setups, each one exploiting a different set of QoS mechanisms at the DDRMC level. QIQoS1 exploits the timeouts defined at port level; QIQoS2 leverages CAM draining features (starving) in the DDRMC. A comprehensive overview of QIQoS parameters for each ADS is provided in Table 2.

4.3.1 Common elements to QIQoS1 and QIQoS2

We first develop on the main common features to both QIQoS. These arose from a series of empirical observations on how to configure some QoS features, since other configurations would prevent enforcing performance guarantees.

- **Use of private ports**. Each CT uses a private port to memory that is not shared with any other CT or NCT. We do so because some QoS features, like traffic class and port type, are set at the port level, so sharing the same port can produce uncontrolled CT's performance drop. Taking Figure 3(b) as a reference, ports P0-P2 are reserved for the CT that run in the R5 and/or A53 cores, and P3-P5 for the PL.
- **Reduced command reordering**. In all QIQoS we set the maximum number of out-of-order commands that can be sent from the CAMs to the DFI to the minimum value allowed (4). Without this limitation, the DDRMC is allowed to prioritize many memory commands over an older memory command if they hit in an open page. This is done for performance-improvement reasons obtained by enabling the DDRMC to increase page hits. However, if the memory command that are bypassed by more recent commands belongs to the CTs, out-of-order commands may have disruptive effects on CTs predictability [36, 55].
- **Avoidance of incoherent QoS setups**. We prevent the incoherent QoS setups [48] by configuring the port type in accordance with the traffic class. Best Effort (BE) ports type are mapped to LPR and LPW traffic classes (respectively for reads and writes), low latency (LL) ports to HPR and LPW traffic classes, and video priority (VP) ports to VPR and VPW traffic classes. This setup is summarized in Figure 4(b). In Table 2, port type and traffic assignments follow these rules for every port (Pi) and under any ADS and QIQoS.
- **Maximization of resource usage**. In the same line, we force the number of entries assigned to HPR and LPR/VPR traffic in the rCAM to match the total number of entries. In particular, we assign 32 of the 64 entries to each partition ($hprSIZE = lvprSIZE = 32$). In Table 2 we see that this criterion holds for all ADS.

Another commonality of all QIQoS setups is that some additional QoS features of the DDRMC are not considered, namely: port aging, urgent transactions, and regions defined at port level. While these features provide additional capabilities to control the ingress of requests to and the egress of memory commands from the DDRMC, they are hard to master in conjunction with the other QoS mechanisms in use. On the one hand, the number of possible QoS configurations increases exponentially. On the other hand, their use can easily produce non-linear effects or even jeopardize the effect of other QoS features, ultimately precluding any chance to achieve the required isolation [48].

4.3.2 QIQoS1

The first QIQoS setup leverages on controlling the ingress of requests to the CAMs to provide isolation for the CT. To that end, QIQoS1 builds on the timeout feature at port level, while the egress prioritization (i.e. the starving feature) is not used. The rows $(ADSi, QIQoS1)$ in Table 2 summarize the parameters enforced under QIQoS1 for each ADS.

Table 2 Parameters of each QIQoS for the DDRMC of the Zynq UltraScale+ for each ADS.

		Port																CAM				Other
		P0				P1				P2				P3,P4,P5				Ingres			Egress	
		TOUT	Type	Traff	Throt	TOUT	Type	Traff	Throt	TOUT	Type	Traff	Throt	TOUT	Type	Traff	Throt	HPR rCAM	LPR rCAM	wCAM	Starving	
ADS1	QIQoS1	RT=1 WT=**X**	VP	VPR VPW	Off	-	-	-	-	-	-	-	-	-	LL	HPR LPW	On	hprSIZE=32 hprAVAIL=**Y**	lvprSIZE=32	wSIZE=64 wAVAIL=**Y**	ShprrCAM=Off SlprrCAM=Off SwCAM=Off	X = (32, 128, 324, 936, 1024, 2048) Y = (1,2,4,8,12)
	QIQoS2	-	LL	HPR LPW	Off	-	-	-	-	-	-	-	-	-	BE	LPR LPW	On	hprSIZE=32	lvprSIZE=32 lvprAVAIL=**Y**	wSIZE=64 wAVAIL=**Y**	ShprrCAM=1 SlprrCAM=40 SwCAM=**Z**	Y = (1,2,4,8,12) Z = (1,16,32,64)
ADS2	QIQoS1	-	-	-	-	RT=1 WT=**X**	VP	VPR VPW	Off	-	-	-	-	-	LL	HPR LPW	On	hprSIZE=32 hprAVAIL=**Y**	lvprSIZE=32	wSIZE=64 wAVAIL=**Y**	ShprrCAM=Off SlprrCAM=Off SwCAM=Off	X = (32, 128, 324, 936, 1024, 2048) Y = (1,2,4,8,12)
	QIQoS2	-	-	-	-	-	LL	HPR LPW	Off	-	-	-	-	-	BE	LPR LPW	On	hprSIZE=32	lvprSIZE=32 lvprAVAIL=**Y**	wSIZE=64 wAVAIL=**Y**	ShprrCAM=1 SlprrCAM=40 SwCAM=**Z**	Y = (1,2,4,8,12) Z = (1,16,32,64)
ADS3	QIQoS1	RT=1 WT=**X**	VP	VPR VPW	Off	RT=1 WT=**X**	VP	VPR VPW	Off	-	-	-	-	-	LL	HPR LPW	On	hprSIZE=32 hprAVAIL=**Y**	lvprSIZE=32	wSIZE=64 wAVAIL=**Y**	ShprrCAM=Off SlprrCAM=Off SwCAM=Off	X = (32, 128, 324, 936, 1024, 2048) Y = (1,2,4,8,12)
	QIQoS2	-	LL	HPR LPW	Off	-	LL	HPR LPW	Off	-	-	-	-	-	BE	LPR LPW	On	hprSIZE=32	lvprSIZE=32 lvprAVAIL=**Y**	wSIZE=64 wAVAIL=**Y**	ShprrCAM=1 SlprrCAM=40 SwCAM=**Z**	Y = (1,2,4,8,12) Z = (1,16,32,64)
ADS4	QIQoS1	-	-	-	-	RT=1 WT=**X**	VP	VPR VPW	Off	RT=1 WT=**X**	VP	VPR VPW	Off	-	LL	HPR LPW	On	hprSIZE=32 hprAVAIL=**Y**	lvprSIZE=32	wSIZE=64 wAVAIL=**Y**	ShprrCAM=Off SlprrCAM=Off SwCAM=Off	X = (32, 128, 324, 936, 1024, 2048) Y = (1,2,4,8,12)
	QIQoS2	-	-	-	-	-	LL	HPR LPW	Off	-	LL	HPR LPW	Off	-	BE	LPR LPW	On	hprSIZE=32	lvprSIZE=32 lvprAVAIL=**Y**	wSIZE=64 wAVAIL=**Y**	ShprrCAM=1 SlprrCAM=40 SwCAM=**Z**	Y = (1,2,4,8,12) Z = (1,16,32,64)
ADS5	QIQoS1	RT=1 WT=**X**	VP	VPR VPW	Off	RT=1 WT=**X**	VP	VPR VPW	Off	RT=1 WT=**X**	VP	VPR VPW	Off	-	LL	HPR LPW	On	hprSIZE=32 hprAVAIL=**Y**	lvprSIZE=32	wSIZE=64 wAVAIL=**Y**	ShprrCAM=Off SlprrCAM=Off SwCAM=Off	X = (32, 128, 324, 936, 1024, 2048) Y = (1,2,4,8,12)
	QIQoS2	-	LL	HPR LPW	Off	-	LL	HPR LPW	Off	-	LL	HPR LPW	Off	-	BE	LPR LPW	On	hprSIZE=32	lvprSIZE=32 lvprAVAIL=**Y**	wSIZE=64 wAVAIL=**Y**	ShprrCAM=1 SlprrCAM=40 SwCAM=**Z**	Y = (1,2,4,8,12) Z = (1,16,32,64)

CT. Read requests from the CT are assigned the VPR traffic class. QIQoS1 sets the timeout values for reads coming from the port(s) used by the CT to one $RT = RTOUT = 1$ as read operations are usually blocking and any delay they can suffer directly affects performance.

Write operations (e.g. stores) instead are assigned the VPW traffic class. They can be processed "off-line" by the A53 as they are not blocking – to a certain extent. For this reason, we set variably high $WT = WTOUT$ values, symbolized with $\mathbf{X}$ in QIQoS1 entries in Table 2. The lower the value of WTOUT, the more frequently requests of the CT – mapped to the VPR/VPW traffic classes – will transition into expired mode and, therefore, will be prioritized over the other requests. Hence, by varying WTOUT we aim to achieve different balance points between CT isolation and NCT performance.

NCT. Read traffic of the NCT is set as HPR. This allows to segregate requests from the CT and NCT in the rCAM with a view to favor isolation: the HPR traffic from the NCT will use the HPR part of the rCAM, while the VPR traffic from the CT will exploit the LPR part of the rCAM. It should be noted that, under normal operation, the NCT HPR traffic do have a higher priority than CT VPR traffic. However, the use of sufficiently small values for RTOUT and WTOUT ensures that the CT VPR traffic is steadily expired and hence its priority is promoted to surpass that of the NCT.

By varying the number of entries that must be available in the hpr rCAM partition (hprAVAIL), symbolized with $\mathbf{Y}$ in QIQoS1 entries in Table 2), it is possible to control the contention generated by the NCT. With high availability thresholds, the port assigned to the NCT will be throttled down more frequently (and vice versa), hence stalling NCT's generated traffic.

Write traffic of the NCT is set as LPW=NPW traffic classes that are not subject to timeout (only VPR/VPW classes are). For the wCAM availability (wAVAIL) we apply the same value ($\mathbf{Y}$) set for hprAVAIL to control the write traffic of the NCT by stalling write traffic from the ports of the NCT (P3-P5) more frequently for low wAVAIL values. Also note that as the CT port (P0-P2) type is *video priority*, CT traffic will not be stalled due to low availability values (it will only when the CAM gets fully occupied).

4.3.3 QIQoS2

Unlike QIQoS1, that relies on the timeout feature to isolate the requests from the CT, QIQoS2 builds on CAM egress control.

Traffic classes. In particular, the read/write requests of the CT are respectively set to HPR/LPW traffic class and the read/write requests of the NCT to the LPR/LPW traffic class (respectively). Hence, since there is no VPR/VPW traffic class, the timeout feature is not used. The main rationale behind QIQoS2 is that no request is upgraded due to a timeout and hence under the default traffic class arbitration policy HPR traffic class is prioritized over LPR and VPR traffic classes.

Starving control. QIQoS2 combines traffic class control as presented above with the CAM egress control feature, or starvation control, to ensure that the time commands of CT are in the CAMs is bounded. In particular:

- We set the starving attribute of the rCAM's hpr partition to 1 ($ShprrCAM = 1$) so as to make reads of the CT to be served as soon as possible.
- The read traffic of the NCT is mapped to the lvpr rCAM partition for which we set a maximum starving period of 40 cycles ($SlvprrCAM = 40$).
- Finally, we explore different values for the starving period of the wCAM, which is shared by writes from all ports, to assess its impact on predictability and performance. This is symbolized as **Z** in QIQoS2 entries in Table 2.

Like $QIQoS1$, $QIQoS2$ controls CAM availability (**Y** in QIQoS2 entries in Table 2): with high availability thresholds causing the port assigned to the NCT to be throttled down more frequently, hence stalling NCT traffic. Also note that, under QIQoS1, NCT reads and CT reads are respectively mapped to the hpr and lpr rCAM partitions, as opposed to QIQoS2, where the hpr rCAM partition holds the CT reads and the lpr rCAM partition holds NCT reads.

4.3.4 Generalization

The concept of QIQoS, i.e. exploiting specific QoS setups to guarantee a high-degree isolation of some applications regardless of the contention co-runners may generate on MPSoC's shared resources, can be extended to other QoS mechanisms and resources. However, the concrete instantiation requires some adaptations with respect to what has been presented in this work.

In this work we focused on the realization of QIQoS on the Zynq UltraScale+. This decision is motivated not only by the complexity of the QoS mechanisms in its DDRMC, but also because of the industrial relevance of the Zynq UltraScale+, which is already considered for avionics certification when running different functionalities (subsystems) [59]. In terms of potential reuse, the Zynq UltraScale+ instantiates the Synopsis Universal DDRMC(uMCTL2) [51] which is quite configurable, allowing designers to tailor it for optimizing latency, bandwidth, and area. Any MPSoC implementing the same DDRMC can directly benefit from the results of this work.

5 Experimental Setup and Results

In this work we focus on a ZCU102 Evaluation Board that comes as part of the Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit [58]. The board is equipped with a Xilinx Zynq UltraScale+ MPSoC [60]. We run on bare-metal (no operating system) and the external

code reduces to the First Stage Boot Loader (FSBL) provided by Xilinx toolchain (Vitis-2019.2). This contributes reducing non-hardware interference (noise) in the experiments. We developed low-overhead software to configure the board by writing to specific control registers and read execution cycles. Under ADS4 and ADS5, we force each SWP to access a different subset of the L2 cache sets, preventing them from evicting each other's data in the LLC. The NCTs were synthesized as accelerators on the PL, which has a direct connection to the three uppermost DDRc ports using the high-performance non-coherent ports ($hp0_lpd$, $hp1_fpd$, $hp2_fpd$, $hp3_fpd$) in its default configuration. The board has different clock domains, and each of them was configured to its maximum allowed frequencies, i.e., PL 250MHz, APU 1200MHz, LPD-interconnect/RPU 500MHz, and FPD-interconnect/DDRc 533.500MHz.

5.1 Experimental Setup

Kernels. We have reviewed different performance-demanding applications relevant for existing and forthcoming safety-critical systems, such as those providing object detection and navigation capabilities. While some of those applications can be run in accelerators, many of them are run on the CPU, either because accelerators are too busy, or because their working set is not overly large and the overheads to issue kernels and transfer data do not pay off [48]. For instance, radar-based object detection uses small matrices, and LiDAR-based object detection may find accelerators busy running heavier camera-based object detection. Hence, both are examples of data-intensive workloads often run in the CPU. The schematic of the hierarchy of their components is shown in Figure 5. A key element in many of those applications is the use of Convolutional Neural Networks (CNNs) for camera and LiDAR-based object detection [52, 2, 3]. In those CNNs, a central element is *matrix multiplication* (MMB) [52], which has been shown to account for most of the execution time (between 67% and 98.5% across deployments [23, 25]). Along with MMB, some other compute-intensive CNN layers rely on *image-to-columns* (I2C) used for tensor lowering to enable matrix convolutions needed by neural networks, and the *rectifier* (RELU) function in neural networks defined as the positive value of its argument [3]. Libraries for CNNs also include other matrix-based operations such as the pervasive *vector-multiply-add* (VMA) and *matrix transpose* (MT) [52]. Those kernels are also present in other key applications such as, for instance, commercial automotive radar applications [27, 53], which build upon MMB to compute the (self) covariance of the input radar data. Overall, as CT applications, we deploy the following benchmarks: MMB, I2C, RELU, VMA, and MT.

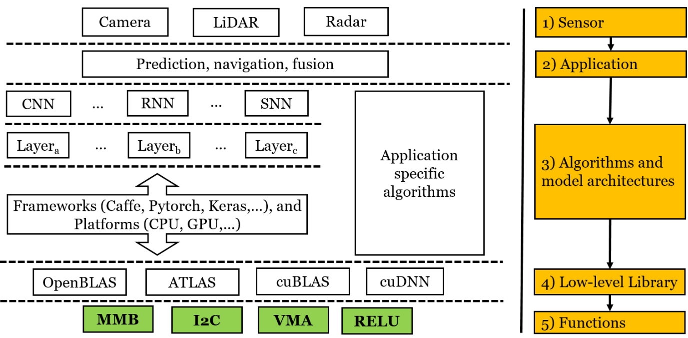


Figure 5 Hierarchy of applications and their components with mapping to specific kernels.

NCT. In the PL, as NCT (i.e. aggressor benchmarks), we instantiate one or several AXI Traffic Generator [61] (ATG) modules to generate variable traffic to stress the DDR. The ATG generates read and write traffic with a burst size of 16 bytes. Operations are strided so they access all DDR banks to maximize the chance of generating page misses on the kernels (CT). Note that it is generally not possible to know the exact memory access patterns of the kernels. Likewise, it is not feasible to interleave the request of the kernels and the NCT at will. For instance, if the kernel accesses banks (B0, B0, B0, B1, B1, B2) it is not possible to force the NCT to access the same bank as the kernel but few cycles before to ensure that the kernel suffers a page miss on every memory access. The degree of *controllablity* required is not achievable in real hardware platforms, not only because of the noise incurred by the RTOS on the NCT but also due to the inherent hardware jitter arising from execution in out-of-order execution processor pipelines and multi-level cache systems.

We define four configurations where NCT (i.e. ATGs) produces an increasing load: Low, Medium, High, and Very high. Under each of these ATG setups, we instantiate an increasing number of ATG units per port (P3, P4, and P5), see Figure 3(b). In particular we instantiate 1 (L), 3 (M), 5 (H), 9 (V) ATGs per port, each one constantly accessing all banks. Hence, the pressure they put in the memory controller is huge and much higher than expected by a regular accelerator, which combines memory accesses phases with computing phases.

When run in isolation, the PL achieves the following bandwidth results (in brackets the relative percentage of the peak bandwidth measured in giga transfers per second): 6.1 GT/s (14%) for L, 17.5 GT/s (40%) for M, 27.8 GT/s (64%) for H, and 43.2 GT/s (100%) for V.

5.2 Experimental Results

5.2.1 Evaluation metrics

QIQoS aim at providing guarantees on the performance of the CT under different loads generated by the NCT. We use three main metrics to assess the effectiveness of a QIQoS setup.

M1. Minimize the slowdown, i.e. maximize the relative performance (rperf), of the CT. A rperf of X means a slowdown of $1/X$ (e.g., rperf=50% means $2x$ slowdown).

M2. Reducing CT's rperf variability across different loads that the NCT (the PL in our case) can put on the DDRMC. Note that M1 and M2 contribute to the primary goal of finding a QoS setup that satisfies the performance requirements of all processes and increase the representativeness of early time budgets by achieving high-degree of performance isolation.

M3. A secondary goal is maximizing NCT rperf, in particular preventing that the NCT receive no service, as long as the target minimum thresholds set for M1 and M2 are achieved.

In terms of M1 and M3, when a SWP encompasses several tasks either as CT or NCT, as it is the case in ADS3 and ADS5, we report the average rperf of all CT tasks and the average of all NCT tasks, respectively. For instance, under ADS3 we report as CT rperf the average of the rperf of the R5 and A53 tasks in each SWP (i.e., SWP1 and SWP2). Likewise, as rperf of the NCT the average of the rperf of all ATGs. Our results show that in all scenarios our results show that the variability in the rperf of the tasks is less than 7 percentage points.

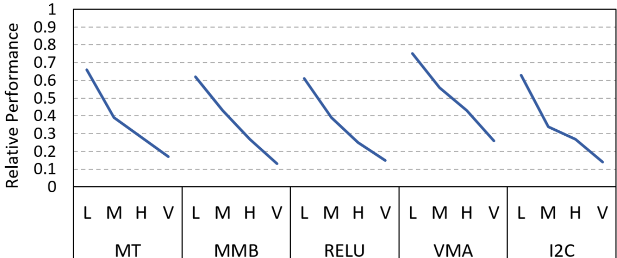


Figure 6 rperf of the CT (kernels) without any QIQoS setup.

5.2.2 Uncontrolled contention

When running the different kernels under the default QoS setup (i.e. no QIQoS) in the Zynq UltraScale+ increasing the traffic load sent from the PL to the DDRMC (L, M, H, and V), we observe that the impact of DDRMC contention on the performance of the benchmarks is huge, see Figure 6. The reported results have been collected while running the benchmarks in one core of the A53 (ADS2), but exactly the same trend is also observed when the benchmarks are run in an R5 core. Even under the smallest PL, i.e. ATG, load (L), rperf drops down to the range $0.60 - 0.75$ for the different benchmarks. As we increase the load, the DDRMC saturates reducing the rperf of the CT down to the range of $0.10 - 0.25$ for V.

5.2.3 Detailed Analysis

In this section, we analyze the challenging ADS4 setup that comprises two SWPs whose tasks are CT, and the ATGs in PL set as NCT, so all traffic coming from ports P3, P4 and P5 belongs to the NCT. For the sake of conciseness, we also focus on the kernel VMA, for which we provide detailed explanations. The results obtained for ADS1, ADS2, ADS3, and ADS5 and the wider set of kernels are explained in Section 5.2.4. Under ADS4 the baseline performance is that of a single copy of VMA running in isolation. This allows us to discriminate between the performance slowdown caused by deploying a second instance of VMA in another A53 core and the slowdown due to the traffic coming from the PL.

5.2.3.1 QIQoS1

The rperf variation observed for both CT and NCT in ADS4 scenario under QIQoS1 is reported in Figure 7. QIQoS1 parameters values are mapped to the x-axis, reporting different CAM availability thresholds ($\mathbf{Y} = hprAVAIL = wAVAIL$) within 1 and 12, and different $\mathbf{X}$=$WTOUT$ values from 2048 down to 32 (as defined in Table 2 for ADS4-QIQoS1). Note that for $QIQoS1$ we refer to both $hprAVAIL$ and $wAVAIL$ as $AVAIL$ for simplicity.

CT performance. Figure 7(a) focuses on VMA rperf under QIQoS1. Note that the rperf results are the same for *both* VMA copies, each running in a different A53 core. The different series represent an increase load of the PL: L, M, H, and V. We observe the following:

1. Under the PL load L (dashed black line), the impact of the NCT traffic on the CT is notably reduced: the slowdown is quite small and stable with rperf around 90% under all $WTOUT$ and $AVAIL$ setups. This slowdown, that is even observed under the highest isolation configurations (i.e. high $AVAIL$ values and low $WTOUT$ values) is caused by the contention the two VMA copies generate on each other.

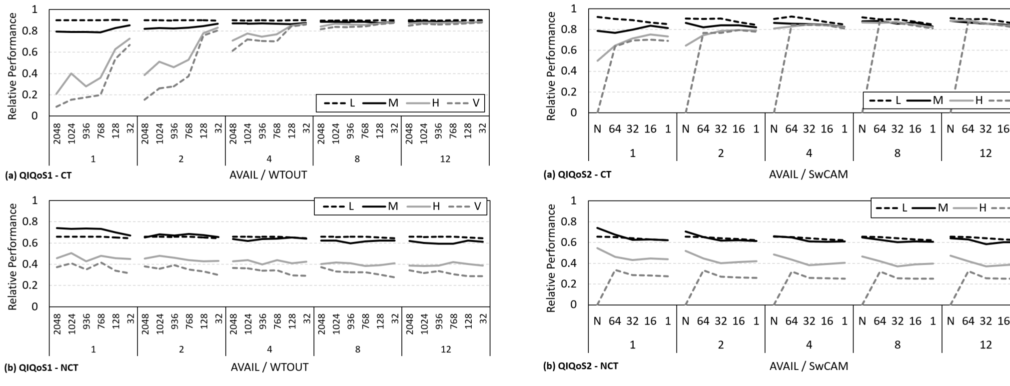


Figure 7 VMA (CT) and NCT rperf for ADS4 under QIQoS1.

Figure 8 VMA (CT) and NCT rperf for ADS4 under QIQoS2.

2. Under the PL load M (solid black line) results are also quite stable, though in this case CT's rperf sits between 80% (specially for small $AVAIL$ values) and 90%. Under both loads, L and M, the variability across setups is limited because the DDRMC can handle requests of both the PL and both VMA copies running in two A53.
3. Under the loads H and V, the impact of the PL traffic increases with higher variability across $AVAIL$ and $WTOUT$ setups. In particular, as we increase $AVAIL$, the ports used by the PL (NCT) are throttled more frequently, reducing their impact on the CT. When $AVAIL$ is set to $8-12$ entries, the slowdown NCT cause on the CT is considerably reduced and the variability across loads also heavily reduces. Also, for any $AVAIL$ value, as we decrease the value of the $WTOUT$, the CT gets less contention from the NCTs as CT requests are more frequently under expired mode and hence are prioritized over NCTs' requests.

NCT performance. Figure 7(b) shows the rperf of the PLs. We see that the impact of varying $WTOUT$ and $AVAIL$ is reduced, with the variability mainly arising because of different PL loads. Under L the PL suffers limited slowdown (i.e. its rperf is high) since the memory controller can comfortably provide the performance required. As we increase the load of the PL to M, H, and V the rperf decreases: for L the rperf stays over 60% while for V it stays below 40%. This is so because heavier loads saturate the DDRMC so they are more affected by the memory activity of the A53. Instead, lighter loads left some bandwidth unused allowing the A53 to inject their traffic with limited impact. As it can be observed, under M and L loads NCT performance is quite similar, even with higher performance under M than under L. Our results seem to suggest that this is due to arbitration among ports that cause higher ID ports to receive comparatively less service under lighter loads.

5.2.3.2 QIQoS2

Figure 8 shows the rperf variation for the CT and NCT in ADS4 scenario under QIQoS2. QIQoS2 relevant parameters are mapped to the x-axis, reporting different CAM availability thresholds ($\mathbf{Y} = lvprAVAIL = wAVAIL$) within 1 and 12, and different wCAM starving ($\mathbf{Z}{=}SwCAM$) values, from 64 down to 1 and no starving (as defined in Table 2 for ADS4-QIQoS2). Note that for $QIQoS2$ we refer to both $lvprAVAIL$ and $wAVAIL$ as $AVAIL$ for simplicity.

Table 3 Analysis of the rperf of the CT and NCT under various benchmarks on ADS4 ($AVAIL=12$ and $WTOUT=32$ for QIQoS1; and $AVAIL=12$ and $SwCAM=1$ for QIQoS2).

		VMA			
		L	M	H	V
QIQoS1	CT	0.90	0.88	0.88	0.87
	NCT	0.64	0.59	0.38	0.27
QIQoS2	CT	0.85	0.82	0.83	0.81
	NCT	0.62	0.60	0.39	0.25

(a)

		MT				MMB			
		L	M	H	V	L	M	H	V
QIQoS1	CT	0.99	0.97	0.96	0.95	0.91	0.88	0.88	0.88
	NCT	0.51	0.27	0.15	0.11	0.62	0.36	0.24	0.19
QIQoS2	CT	0.88	0.85	0.84	0.83	0.82	0.76	0.76	0.76
	NCT	0.59	0.47	0.29	0.19	0.60	0.38	0.23	0.17

		RELU				I2C			
		L	M	H	V	L	M	H	V
QIQoS1	CT	0.96	0.97	0.92	0.97	0.94	0.91	0.89	0.89
	NCT	0.46	0.37	0.12	0.14	0.51	0.29	0.18	0.12
QIQoS2	CT	0.81	0.81	0.80	0.79	0.81	0.77	0.76	0.75
	NCT	0.58	0.42	0.27	0.19	0.59	0.30	0.19	0.14

(b)

CT performance. Regarding the rperf of the CT (VMA), see Figure 8(a), we extract two main conclusions. On the one hand, for the “no starving” case, we see that the impact of the PL is higher. It is particularly relevant the V load for which the experiment, after executing more than 100x times its duration in isolation, did not finish. Hence, we concluded that starving must be enabled and do not further discuss “no starving” results. It is noted that while starving prevention is enabled by default, it is one of the parameters whose impact we wanted to explore and hence decided to observe the impact of disabling it. On the other hand, as we increase $AVAIL$ we see how the rperf of the CT slightly increases. CT’s rperf also increases as we decrease the starvation threshold as CT requests are kept shorter in the CAMs when $AVAIL$ is 1 or 2 (this effect reduces and even disappears for M and L, arguably because the load of the NCT on the DDRMC decreases). When $AVAIL$ goes beyond 2 (i.e. 4, 8, 12), it cancels out the impact of starvation prevention. Anyways, the impact on rperf is relatively small, so QIQoS keeps high quality results (high values for rperf and low variability of rperf across load) under all explored variations.

NCT performance. In Figure 8(b) shows the rperf of the NCT. Other than for the “no starving” setup that, as pointed out before, we exclude from the discussion, the variability is very small across the explored $AVAIL$ and $SwCAM$ values, with a slight decrease with higher values of both parameters. This is so because lower loads of the PL do not saturate the DDRMC, thus leaving some bandwidth for the A53 and R5 to execute with less impact on the PL. The major difference appears across PL loads, with rperf drop values around 30% for L, 40% for M, 60% for H, and 70% for V.

5.2.3.3 Metrics M1, M2, and M3

We assess the quality of QIQoS1 and QIQoS2 using metrics M1, M2, and M3 as defined in Section 5.2.1. For both QIQoS we choose the aggressive isolation setups: $AVAIL = 12$ and $WTOUT = 32$ for QIQoS1 and $AVAIL = 12$ and $SwCAM = 1$ for QIQoS2.

Results are shown in Table 3(a), comparing the QIQoS setups on VMA rperf for variable PL load, yet focusing on the specific scenario ADS4. Regarding M1, for VMA we see that the rperf of the CT is slightly higher with QIQoS1, varying from 0.87 to 0.90, than for QIQoS2 for which CT rperf varies from 0.81 to 0.85. For both QIQoS rperf is quite high. In terms of variability (M2) both are quite similar being 3 percentage points for QIQoS1 and 4 for QIQoS2. The results for M1 and M2 provide evidence that the developed QIQoS are very

competitive, achieving good rperf and isolation figures. This, in fact, allows to define SWPs timing budgets during early stages of the development process of a IMA-SoC product, with high confidence that those bounds are going to hold at operation. In this line, it is worth recalling that under the V load there are 9 ATGs *per port* constantly accessing memory to different banks. This is arguably a much higher load than an accelerator would put on the DDRMC. Finally, the performance of the NCT (M3) is quite similar across both setups.

5.2.4 Wider result set

Figures 7 and 8 provided a detailed set of results and analysis of the proposed QIQoS setups for one specific benchmark (VMA) under a single scenario (ADS4). In this section, we extend the analysis of results to all kernels and all ADS scenarios but, for the sake of conciseness, we restrict our focus on a specific PL load (L) and fixing the parameters $AVAIL = 12$ and $WTOUT = 32$ for QIQoS1 and $AVAIL = 12$ and $SwCAM = 1$ for QIQoS2. In any case, results for variable loads and kernels under ADS4 are reported in Table 3(b), as will be commented next.

Figure 9 shows the rperf of the CT (left) and the rperf of the NCT (right) for all benchmarks under all ADS scenarios, with PL load set to L. In both charts, results for each kernel under QIQoS1 and QIQoS2 are shown in consecutive bars. For instance, bars 1 and 2 compare the result of QIQoS1 and QIQoS2 for ADS1, bars 3 and 4 for ADS2, bar 5 and 6 for ADS3, and so on so forth. We observe that:

- In terms of the rperf of the CT (left chart), in general QIQoS1 provides slightly better performance than QIQoS2. The rperf reduces for more aggressive ADS staying between 0.5 and 0.8.
- In terms of the rperf of the NCT (right chart), in general results are quite similar across the proposed setups, with QIQoS2 slightly outperforming QIQoS1.

Regarding metrics M1, M2, and M3, under ADS4 the same conclusions derived for VMA under different PL loads generally hold for the other kernels, as reported in Table 3(b). First, QIQoS1 provides higher CT's performance than QIQoS2 (M1). Second, the variability of CT's rperf (M2) is quite similar for both QIQoS: 5 percentage points in the worst case for QIQoS1 and 6 for QIQoS2. And third, in terms of the rperf of the NCT both setups provide similar results with rperf gradually increasing as the load of the PL moves from L to V.

Overall, we conclude the same trends observed for VMA holds for the rest of the kernels and configurations, confirming that the proposed QIQoS achieved the intended goals.

6 Conclusions

In this work we have shown how hardware QoS support can be exploited in modern MPSoCs with the goal of providing a high degree of isolation to selected applications. We instantiate Quasi Isolation QoS setups (QIQoS), introduce and explain two particular QIQoS to achieve isolation in the DDR memory controller of the Zynq UltraScale+. The main lessons learned are that transaction timeout (QIQoS1) and CAM starving control (QIQoS2), both underpinned by traffic classes and port throttling, provide good isolation results. Overall, the proposed QIQoS guarantee that applications' execution time is much less sensitive to co-runners' contention, and hence, the timing estimates obtained when running the application against aggressors in early development stages become much tighter and stable across integrations. Our future work includes exploiting more features of the DDR memory controller to further isolate APU and RPU cores from the PL traffic, and also isolate cores in the APU and the RPU from each other.

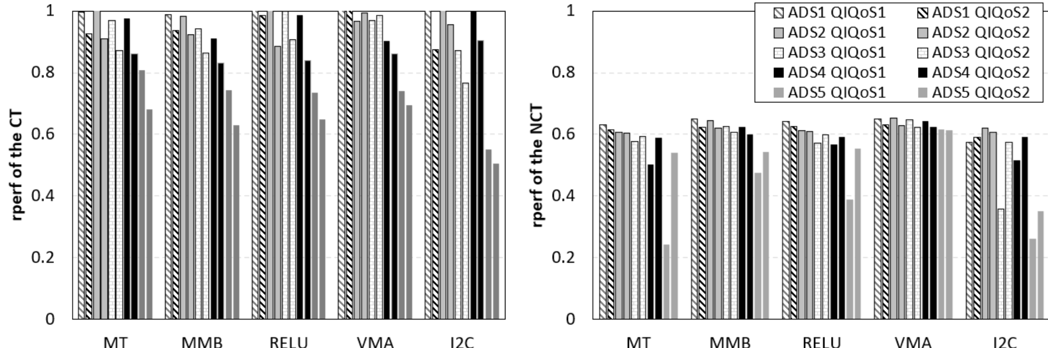


Figure 9 Minimum rperf for the CT (left chart) and NCT (right chart), for QIQoS1 and QIQoS2 under L load of the PL ($AVAIL = 12$ and $WTOUT = 32$ for QIQoS1; and $AVAIL = 12$ and $SwCAM = 1$ for QIQoS2).

References

1 Irune Agirre, Jaume Abella, Mikel Azkarate-askasua, and Francisco J. Cazorla. On the tailoring of CAST-32A certification guidance to real COTS multicore architectures. In *12th IEEE International Symposium on Industrial Embedded Systems, SIES, Toulouse, France, June 14-16, 2017*, pages 1–8. IEEE, 2017. doi:10.1109/SIES.2017.7993376.

2 ApolloAuto. Apollo 3.0 Software Architecture, 2018. URL: https://github.com/ApolloAuto/apollo/blob/master/docs/specs/Apollo_3.0_Software_Architecture.md.

3 ApolloAuto. Perception, 2018. URL: https://github.com/ApolloAuto/apollo/blob/r3.0.0/docs/specs/perception_apollo_3.0.md.

4 ARINC Inc. *ARINC Specification 653: Avionics Application Software Standard Standard Interface*, June 2012.

5 Arm. *ARM CoreLink NIC-400 Network Interconnect Technical Reference Manual.*

6 Arm. *ARM CoreLink QoS-400 Network Interconnect Advanced Quality of Service Supplement to ARM CoreLink NIC-400 Network Interconnect Technical Reference Manual*, 2017.

7 Arm. *ARM CoreLink QVN-400 Network Interconnect Advanced Quality of Service using Virtual Networks Supplement to ARM CoreLink NIC-400 Network Interconnect Technical Reference Manual*, 2017.

8 Arm. *ARM Cortex-A53 MPCore Processor Technical Reference Manual. Version r0p4*, 2022. URL: https://developer.arm.com/documentation/ddi0500/j/.

9 Arm. *Arm® Architecture Reference Manual Supplement Memory System Resource Partitioning and Monitoring (MPAM), for Armv8-A*, 2022.

10 Arm. *Cortex-R5 and Cortex-R5F Technical Reference Manual. Version r1p1*, 2022. URL: https://developer.arm.com/documentation/ddi0460/c/.

11 Arm. *Mali-400*, 2022. URL: https://developer.arm.com/Processors/Mali-400.

12 ARP4761. *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*, 2001.

13 AUTOSAR. *Technical Overview V2.0.1*, 2006.

14 Jingyi Bin, Sylvain Girbal, Daniel Gracia Pérez, Arnaud Grasset, and Alain Mérigot. Studying co-running avionic real-time applications on multi-core COTS architectures. In *Embedded Real Time Software and Systems (ERTS2014)*, 2014. URL: https://hal.science/hal-02271379.

15 Carlos Boneti, Francisco J. Cazorla, Roberto Gioiosa, Alper Buyuktosunoglu, Chen-Yong Cher, and Mateo Valero. Software-Controlled Priority Characterization of POWER5 Processor. In *35th International Symposium on Computer Architecture (ISCA)*, pages 415–426, 2008. doi:10.1109/ISCA.2008.8.

16 Marco Caccamo, Rodolfo Pellizzoni, Lui Sha, Gang Yao, and Heechul Yun. MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-Core Platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, USA, April 2013. IEEE Computer Society. doi:10.1109/RTAS.2013.6531079.

17 Jordi Cardona, Carles Hernandez, Enrico Mezzetti, Jaume Abella, and Francisco J. Cazorla. NoCo: ILP-based worst-case contention estimation for mesh real-time manycores. In *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, December 2018. doi:10.1109/rtss.2018.00043.

18 Certification Authorities Software Team. *CAST-32A Multi-core Processors*, 2016.

19 Dakshina Dasari and Vincent Nelis. An analysis of the impact of bus contention on the WCET in multicores. In *HPCC*, 2012. doi:10.1109/HPCC.2012.212.

20 DDC-I. Deos, a Time and Space Partitioned, Multi-core Enabled, RTOS Verified to DO-178C ED-12C DAL A, 2022. URL: https://www.ddci.com/products_deos_do_178c_arinc_653/.

21 EASA, FAE. *General Acceptable Means of Compliance for Airworthiness of Products, Parts and Appliances (AMC-20). Amendment 23. Annex I to ED Decision 2022/001/R. AMC 20-193 Use of multi-core processors.*, 2022. URL: https://www.easa.europa.eu/en/document-library/certification-specifications/amc-20-amendment-23.

22 Falk Rehm and Jörg Seitter. Software Mechanisms for Controlling QoS. In *2021 Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Virtual Conference, February 01-05, 2021*, pages 1485–1488, 2016.

23 Fernando Fernandes dos Santos, Lucas Draghetti, Lucas Weigel, Luigi Carro, Philippe Navaux, and Paolo Rech. Evaluation and Mitigation of Soft-Errors in Neural Network-Based Object Detection in Three GPU Architectures. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 169–176, 2017. doi:10.1109/DSN-W.2017.47.

24 Gabriel Fernandez, Javier Jalle, Jaume Abella, Eduardo Quiñones, Tullio Vardanega, and Francisco J. Cazorla. Resource usage templates and signatures for COTS multicore processors. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pages 155:1–155:6. ACM, 2015. doi:10.1145/2744769.2744901.

25 Javier Fernández, Jon Perez, Irune Agirre, Imanol Allende, Jaume Abella, and Francisco J. Cazorla. Towards functional safety compliance of matrix–matrix multiplication for machine learning-based autonomous systems. *Journal of Systems Architecture*, 121:102298, 2021. doi:10.1016/j.sysarc.2021.102298.

26 Freescale semicondutor. QorIQ T2080 Reference Manual, 2016. Also supports T2081. Doc. No.: T2080RM. Rev. 3, 11/2016.

27 Jonah Gamba. Automotive radar applications. In *Radar Signal Processing for Autonomous Driving*, pages 123–142. Springer Singapore, Singapore, 2020. doi:10.1007/978-981-13-9193-4_9.

28 Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. A Survey on Cache Management Mechanisms for Real-Time Embedded Systems. *ACM Computing Surveys*, 48(2):32:1–32:36, November 2015. doi:10.1145/2830555.

29 Giovani Gracioli, Rohan Tabish, Renato Mancuso, Reza Mirosanlou, Rodolfo Pellizzoni, and Marco Caccamo. Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. In Sophie Quinton, editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133, pages 27:1–27:25, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECRTS.2019.27.

30 Danlu Guo, Mohamed Hassan, Rodolfo Pellizzoni, and Hiren Patel. A Comparative Study of Predictable DRAM Controllers. *ACM Trans. Embed. Comput. Syst.*, 17(2), February 2018. doi:10.1145/3158208.

31 Andrew Herdrich, Ramesh Illikkal, Ravi Iyer, Ronak Singhal, Matt Merten, and Martin Dixon. SMT QoS: Hardware Prototyping of Thread-level Performance Differentiation Mechanisms. In *USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, 2012. USENIX Association.

32 Denis Hoornaert, Shahin Roozkhosh, and Renato Mancuso. A Memory Scheduling Infrastructure for Multi-Core Systems with Re-Programmable Logic. In Björn B. Brandenburg, editor, *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, volume 196, pages 2:1–2:22, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ECRTS.2021.2`.

33 Intel. *Intel® Resource Director Technology (Intel® RDT) on 2nd Generation Intel® Xeon® Scalable Processors Reference Manual, Rev. 1.0*, 2019.

34 International Organization for Standardization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.

35 Dan Iorga, Tyler Sorensen, John Wickerson, and Alastair F. Donaldson. Slow and Steady: Measuring and Tuning Multicore Interference. In *26th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'20)*, pages 200–212, April 2020. `doi:10.1109/RTAS48715.2020.000-6`.

36 Javier Jalle, Mikel Fernandez, Jaume Abella, Jan Andersson, Mathieu Patte, Luca Fossati, Marco Zulianello, and Francisco J. Cazorla. Bounding Resource-Contention Interference in the Next-Generation Multipurpose Processor (NGMP). In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016. URL: `https://hal.science/hal-01259133`.

37 Javier Jalle, Eduardo Quiñones, Jaume Abella, Luca Fossati, Marco Zulianello, and Francisco J. Cazorla. A Dual-Criticality Memory Controller (DCmc): Proposal and Evaluation of a Space Case Study. In *RTSS*, pages 207–217. IEEE Computer Society, 2014. `doi:10.1109/RTSS.2014.23`.

38 Matthias Jung, Sally A. McKee, Chirag Sudarshan, Christoph Dropmann, Christian Weis, and Norbert Wehn. Driving into the memory wall: the role of memory for advanced driver assistance systems and autonomous driving. In Bruce Jacob, editor, *Proceedings of the International Symposium on Memory Systems, MEMSYS*. ACM, 2018. `doi:10.1145/3240302.3240322`.

39 Hyoseung Kim, Dionisio De Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding and Reducing Memory Interference in COTS-Based Multi-Core Systems. *Real-Time Syst.*, 52(3):356–395, May 2016. `doi:10.1007/s11241-016-9248-1`.

40 LYNX Sofware technologies. LynxSecure Separation Kernel Hypervisor, 2022. URL: `https://www.lynx.com/products/lynxsecure-separation-kernel-hypervisor`.

41 Kristiyan Manev, Anuj Vaishnav, and Dirk Koch. Unexpected Diversity: Quantitative Memory Analysis for Zynq UltraScale+ Systems. In *2019 International Conference on Field-Programmable Technology*, pages 179–187, 2019. `doi:10.1109/ICFPT47387.2019.00029`.

42 Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Duetto: Latency Guarantees at Minimal Performance Cost. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1136–1141, 2021. `doi:10.23919/DATE51398.2021.9474062`.

43 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. `doi:10.1109/ECRTS.2014.20`.

44 NXP Semiconductors. QorIQ LX2160A Reference Manual, 2021. Supports LX2120A and LX2080A.

45 Xavier Palomo, Enrico Mezzetti, Jaume Abella, Reinder J. Bril, and Francisco J. Cazorla. Accurate ILP-Based Contention Modeling on Statically Scheduled Multicore Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 15–28. IEEE, 2019. `doi:10.1109/RTAS.2019.00010`.

46 Jon Pérez-Cerrolaza, Roman Obermaisser, Jaume Abella, Francisco J. Cazorla, Kim Grüttner, Irune Agirre, Hamidreza Ahmadian, and Imanol Allende. Multi-core Devices for Safety-critical Systems: A Survey. *ACM Comput. Surv.*, 53(4), 2020.

47 RTCA and EUROCAE. *DO-178C - ED-12C, Software Considerations in Airborne Systems and Equipment Certification*, 2011.

48 Alejandro Serrano-Cases, Juan M. Reina, Jaume Abella, Enrico Mezzetti, and Francisco J. Cazorla. Leveraging Hardware QoS to Control Contention in the Xilinx Zynq UltraScale+ MPSoC. In *33rd Euromicro Conference on Real-Time Systems, ECRTS 2021, July 5-9, 2021, Virtual Conference*, volume 196, pages 3:1–3:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ECRTS.2021.3.

49 Parul Sohal, Michael Bechtel, Renato Mancuso, Heechul Yun, and Orran Krieger. A Closer Look at Intel Resource Director Technology (RDT). In *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, pages 127–139, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3534879.3534882.

50 Parul Sohal, Rohan Tabish, Ulrich Drepper, and Renato Mancuso. Profile-driven memory bandwidth management for accelerators and CPUs in QoS-enabled platforms. *Real Time Syst.*, 58(3):235–274, 2022.

51 Synopsys. Synopsys Enhanced Universal DDR Memory Controller (uMCTL2). URL: https://www.synopsys.com/dw/ipdir.php?ds=dwc_ddr_universal_umctl2.

52 Hamid Tabani, Roger Pujol, Jaume Abella, and Francisco J. Cazorla. A Cross-Layer Review of Deep Learning Frameworks to Ease Their Optimization and Reuse. In *IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*, 2020. doi:10.1109/ISORC49007.2020.00030.

53 Lee Teschler. The basics of automotive radar, 2019. URL: https://www.designworldonline.com/the-basics-of-automotive-radar/.

54 P. Valsan and H. Yun. MEDUSA: A Predictable and High-Performance DRAM Controller for Multicore Based Embedded Systems. In *2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pages 86–93, Los Alamitos, CA, USA, August 2015. IEEE Computer Society. doi:10.1109/CPSNA.2015.24.

55 Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming Non-Blocking Caches to Improve Isolation in Multicore Real-Time Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14*, 2016. doi:10.1109/RTAS.2016.7461361.

56 James Windsor, Marie-Hélène Deredempt, and Regis De-Ferluc. Integrated modular avionics for spacecraft — User requirements, architecture and role definition. In *IEEE/AIAA 30th Digital Avionics Systems Conference*, 2011. doi:10.1109/DASC.2011.6096141.

57 Xi-Yue Xiang, Saugata Ghose, Onur Mutlu, and Nian-Feng Tzeng. A model for Application Slowdown Estimation in on-chip networks and its use for improving system fairness and performance. In *34th IEEE International Conference on Computer Design, ICCD, Scottsdale, AZ, USA*, pages 456–463. IEEE Computer Society, 2016. doi:10.1109/ICCD.2016.7753327.

58 XILINX. Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit. URL: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html#information.

59 XILINX. *Rockwell Collins Uses Zynq UltraScale+ RFSoC Devices in Revolutionizing How Arrays are Produced and Fielded: Powered by Xilinx*, 2018. URL: https://www.xilinx.com/video/corporate/rockwell-collins-rfsoc-revolutionizing-how-arrays-are-produced.html.

60 XILINX. *Zynq UltraScale+ Device. Technical Reference Manual. UG1085 (v2.1)*, 2019.

61 XILINX. LogiCore AXI Traffic Generator. https://www.xilinx.com/products/intellectual-property/axi_tg.html, 2022.

62 XILINX. XILINX VERSAL. AI EDGE. https://www.xilinx.com/products/silicon-devices/acap/versal.html, 2022.

63 Matteo Zini, Daniel Casini, and Alessandro Biondi. Analyzing Arm's MPAM From the Perspective of Time Predictability. *IEEE Transactions on Computers*, 72(1):168–182, 2023. doi:10.1109/TC.2022.3202720.

64 Matteo Zini, Giorgiomaria Cicero, Daniel Casini, and Alessandro Biondi. Profiling and controlling I/O-related memory contention in COTS heterogeneous platforms. *Software: Practice and Experience*, 52(5):1095–1113, 2022. doi:10.1002/spe.3053.

FusionClock: Energy-Optimal Clock-Tree Reconfigurations for Energy-Constrained Real-Time Systems

Eva Dengler
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Phillip Raffeck
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Simon Schuster
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Peter Wägemann
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Abstract

Numerous embedded real-time systems have, besides their timing requirements, strict energy constraints that must be satisfied. Examples of this class of real-time systems are implantable medical devices, where knowledge of the worst-case execution time (WCET) has the same importance as of the worst-case energy consumption (WCEC) in order to provide runtime guarantees. The core hardware component of modern system-on-chip (SoC) platforms to configure the tradeoff between time and energy is the system's clock tree, which provides the necessary clock source to all connected devices (i.e., memory, sensors, transceivers). Existing energy-aware scheduling approaches have shortcomings with regard to these modern, feature-rich clock trees: First, with their reactive, dynamic (re-)configuration of the clock tree, they are not able to provide static guarantees of the system's resource consumption (i.e., energy and time). Second, they only account for dynamic voltage/frequency scaling of the CPU and thereby miss the reconfiguration of clock sources and clock speed for the other connected devices on such SoCs. Third, they neglect the reconfiguration penalties of frequency scaling and clock/power gating in the presence of the CPU's sleep modes.

In this paper, we present FusionClock, an approach that exploits a fine-grained model of the system's temporal and energetic behavior. By means of our developed clock-tree model, FusionClock processes time-triggered schedules and finally generates optimized code for a system where offline-determined and online-applied reconfigurations lead to the worst-case–optimal energy demand while still meeting given timing-related deadlines. For statically determining these energy-optimal reconfigurations on task level, FusionClock builds a mathematical optimization problem based on the tasks' specifications and the system's resource-consumption model. Specific components like transceivers of SoCs usually have strict requirements regarding the used clock source (e.g., phase-locked loop, RC network, oscillator). FusionClock accounts for these clock-tree requirements with its ability to exploit application-specific knowledge within an optimization problem. With our resource-consumption model for a modern SoC platform and our open-source prototype of FusionClock, we are able to achieve significant energy savings while still providing guarantees for timeliness, as our evaluations on a real hardware platform (i.e., ESP32-C3) show.

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

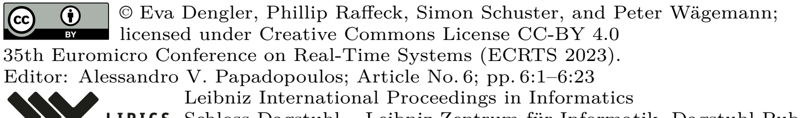
Keywords and phrases energy-aware scheduling, device-aware whole-system analysis, clock tree

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.6

Supplementary Material *Software (ECRTS 2023 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.9.1.2`
Software (Source Code): `https://gitlab.cs.fau.de/fusionclock`

Funding by the Deutsche Forschungsgemeinschaft (DFG) – 502947440 (WA 5186/1-1, Watwa).

Acknowledgements We thank Tim Rheinfels for sharing his expertise with the energy measurements.


© Eva Dengler, Phillip Raffeck, Simon Schuster, and Peter Wägemann;
licensed under Creative Commons License CC-BY 4.0
35th Euromicro Conference on Real-Time Systems (ECRTS 2023).
Editor: Alessandro V. Papadopoulos; Article No. 6; pp. 6:1–6:23
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Providing Static Time & Energy Guarantees. Developing systems that meet resource requirements (i.e., time and energy in this paper) with provably optimal resource usage is a central challenge in computer science [9, 32]. While the real-time–systems community has developed numerous analysis approaches to guarantee timing requirements with energy awareness in embedded single-core systems [3, 53], the combined handling of both time and energy constraints still goes beyond the current state of the art in view of modern system-on-chip (SoC) hardware platforms. Specific application scenarios that need to meet both timing and energy requirements include implantable medical devices, such as artificial cardiac pacemakers or defibrillators. Guaranteeing timeliness demands the worst-case execution time (WCET) [51] in order to build a schedule that eventually meets the given tasks' deadlines. Likewise, the tasks' worst-case energy consumption (WCEC) [22, 47, 48, 49] is a fundamental measure for enabling a guaranteed execution of jobs under given limited energy budgets. Having an accurate model of the target hardware platform's temporal and energetic behavior is essential in order to give static resource-consumption guarantees.

Configuring the Time & Energy Tradeoff with Clock Trees. Modern integrated SoC platforms [13] offer a huge variety of features and options to configure the tradeoff between performance (i.e., execution speed) and energy demand, with the heart of this configuration space being the system's *clock tree* [8, 40]. The purpose of the clock tree is to distribute available clock signals to all components on the SoC by utilizing different signal-forwarding mechanisms such as *multiplexers*, *scalers*, or *clock gates*. Figure 1 shows an example of such a clock tree, which will be detailed later. Since components provided with a clock source via an active signal through the clock-distribution network eventually lead to an increase in the whole system's energy consumption (i.e., power over time), these *clock gates* are also referred to as *power gates*. Besides gating power (i.e., on/off switching), the clock tree includes the possibility to scale frequencies up/down with multipliers by using *scalers* or select one out of multiple input signals with the use of *multiplexers*. In summary, modern clock trees of SoCs provide substantial configuration spaces for the tradeoff between time and energy, which has not yet been sufficiently addressed in the context of energy-constrained real-time systems.

Energy Demand of Devices. The clock tree not only spans the configuration options for the CPU: Modern SoC platforms are characterized by their multitude of integrated components, such as transceivers (e.g., WiFi, Bluetooth, LoRa), sensors (e.g., analog-to-digital converter, ADC), controllers (e.g., USB, SPI, DMA), or storage devices (e.g., non-volatile memory). From a generic point of view, all these components have the same behavior as the CPU with regard to clock gating: Switching off devices by clock gating them (when their service is not needed) significantly reduces the system's power and, therefore, is beneficial for energy savings. While numerous energy-aware real-time scheduling approaches account for the systems' energy demand [3, 53], they have shortcomings with (1) comprehensively modeling the resource demand of devices and (2) handling their hardware-related constraints with respect to multiple available clock sources: For example, the WiFi device on a SoC [13] typically requires a specific clock-tree configuration to operate. During operation, its power demand is up to 1105 mW, being significantly larger than the CPU's demand in run mode (i.e., at 1 MHz: 20 mW, at 160 MHz: 100 mW). In summary, to address optimal energy-consumption reduction in real-time systems, knowledge of the whole system's resource-consumption behavior, with all connected consumers, is inevitable.

Low-Power Phases & Sleep Modes. As with the mentioned devices, the CPU is a device itself and does not necessarily have a utilization of 100 % in embedded, energy-constrained settings. Lower utilizations, and thereby slack time, offer the possibility to enter sleep modes that decrease the system's power demand down to, for example, 0.15 mW for the previously mentioned SoC [13]. From a technical perspective, entering a sleep mode means a reconfiguration of the clock tree. An essential accompanying, unavoidable effect of clock-tree reconfigurations is the *penalty* for the reconfiguration. That is, both clock gating and clock scaling involve significant time and accompanying energy overheads, which, for example, come from the duration to stabilize the frequency of a phase-locked loop (PLL) clock source. To give an example, reconfiguring the clock to wake up from a deep sleep mode, enter run mode, and being able to execute user-provided code takes 70.26 ms on the example SoC [13], which needs to be accounted for (1) resource-optimal and (2) deadline-aware execution. With regard to the given real-time constraints, *break-even points* decide whether a specific clock-tree reconfiguration is beneficial: For example, entering and subsequently exiting a sleep mode could have adverse effects on the system's resource consumption. In this paper, we exploit a comprehensive clock-tree model including reconfiguration penalties (i.e., transition costs between clock configurations) to determine worst-case–optimal reconfigurations.

Application-Dependent Reconfiguration. Not all tasks in real-time systems make use of further devices besides the CPU. When considering chains of sense-compute-actuate tasks, the sense phase requires data from sensor devices, and the actuate phase could use transmitters to share results. Neither is the sensor required by the actuation phase nor the transmitter during sensing. For computation tasks, no further devices are needed. Having active devices while executing the compute phase leads to subpar energy demands. From a general perspective, task-agnostic clock-tree reconfigurations inevitably lead to subpar results. Thus, making use of application dependencies is essential for resource-optimal solutions.

Paper's Contributions. This paper introduces FusionClock, an approach that addresses the challenge of meeting deadlines of real-time tasks on single-core SoC platforms while determining an optimum for the energy demand with the use of WCET and WCEC knowledge. FusionClock handles time-triggered schedules and generates code for online reconfiguration of the system's clock tree, tailored for the specific device usage. The name FusionClock originates from our objective of resource-optimally fusing clock-tree reconfigurations with the application's device requirements. FusionClock is able to handle clock trees with multiple input sources and clock/power gates. The paper makes five contributions:

1. *Problem Formalization*: We present a generic quadratic optimization formulation that makes use of a resource-consumption model and is able to adhere to real-time constraints while optimally fulfilling the objective of minimizing worst-case energy demands.
2. *Resource-Consumption Model*: We developed a hardware model for clock-tree reconfigurations on a SoC platform, which is the basis to resource guarantees.
3. *Application-Aware Approach*: We propose an application-aware approach that exploits knowledge of clock requirements and device-aware resource-consumption analyses.
4. *Code Generation*: Based on the quadratic problem's solution for a given taskset and time-triggered schedule, our code-generation approach yields a functionally equivalent but resource-optimal schedule with code for reconfiguring the clock tree between jobs.
5. *Evaluation*: Relying on a hardware platform and employing accurate energy measurements, we demonstrate the effectiveness and validity of our open-source prototype of FusionClock.

2 Background & System Model

FusionClock targets embedded energy-constrained real-time systems executed on single-core SoC platforms. For the tasksets to be optimized, FusionClock assumes a strictly periodic, cyclic task model. As FusionClock optimizes the clock-tree configurations of pre-existing time-triggered schedules, we further assume the availability of a valid, non-preemptive schedule for the taskset. As common in real-time systems, each individual task τ_i can be described by the parameters of its period T_i, its WCET C_i, and its relative deadline D_i. After the hyperperiod H, which is the least common multiple of task periods, the schedule (with its determined clock-tree reconfigurations) repeats. Specific to our approach, we give WCET bounds dependent on the clock-tree configuration, replacing the single C_i with the mapping $C_i(conf)$. Additionally, the task description is extended by the set of device dependencies (e.g., τ_1 uses ADC device 2).

As an ahead-of-time mechanism, FusionClock has requirements on the hardware architecture. We assume static analyzability in the temporal and energetic domain: The hardware's structure allows for the derivation of a static, sound model for the purpose of WCET and WCEC analyses with acceptable analysis pessimism. Besides the feasibility of capturing the microarchitectural behavior, such suitability for resource analysis includes the possibility to describe the energy demand of particular program sections with a known system state (i.e., clock-tree and device configuration) as (monotonic) function of its execution time [48]. We further assume compositionality for energy and time [19, 37]: The validity of safely combining the individual resource demands of continuous sections to a total demand. The limited complexity of the hardware (RISC architecture, simple microarchitecture) found in the targeted system class [13] facilitates modeling for static analyses.

Apart from the processor, SoC systems in the targeted domain also feature numerous devices, such as sensors, actuators, or peripheral communication devices. Due to the nature of SoCs with their integration of various components, all peripherals are generally seen as devices. Those devices significantly influence the system's overall power consumption. We assume to have an accurate bound on the maximum power demand of those devices in all of their different operation modes as well as the transitions between different modes. Accurately determining such application-specific maximum power demands is possible, as shown by Cherupalli et al. [7], which validates FusionClock's related assumption. The same consideration of being able to accurately model time and energy penalties holds for clock-tree reconfigurations, as shown by Park et al. [36].

We assume a feature-rich clock tree that can be reconfigured at runtime by software, allowing fine-grained control over the power consumption of any devices in the system. As for peripheral devices, accurate upper bounds on the time and energy demand of clock-tree–configuration switches are statically determinable. Lastly, every sleep mode of the clock tree has a lower power consumption than all modes used for the execution of tasks. This assumption prevents that unexpected idle times consume more power than the execution of tasks, which is given for our target SoC [13].

The Clock Tree. The *clock tree* is the clock-distribution network within a system, routing the signal from a clock source to all components in the system, potentially modifying the source signal for specific devices. The complexity of this network heavily depends on the chosen hardware platform. Modern SoC platforms feature a variety of clock sources based on different technologies to serve diverse needs and provide a suitable signal source for different application and device demands. The provided clock sources differ from each other with

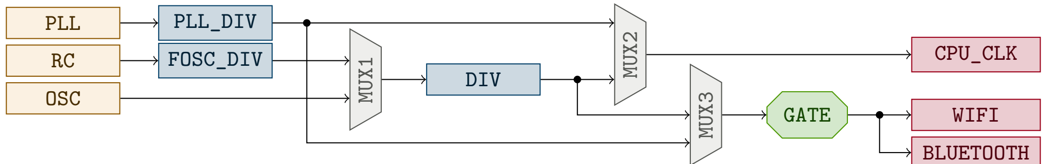


Figure 1 Example of a clock tree for modern SoC platforms [13]. The requirement for the clock-tree configuration are both task- and hardware-related: (1) Tasks can require a specific device (e.g., WiFi) and (2) devices can require a specific clock source with specific multiplexer/divider settings.

regard to, for example, precision, energy efficiency, signal stability, and robustness against environmental conditions [40]. In general, not every signal source may be suitable for every device or at least not for every operation mode of a device, for example, because a device operation requires a particularly high frequency that cannot be provided by every source.

Figure 1 shows an exemplary clock tree capable of clock-source selection and signal modification. In the example, the tree has three different clock sources (`PLL`, `RC`, `OSC`) and three different devices (`CPU_CLK`, `WIFI`, `BLUETOOTH`), whereas `CPU_CLK` denotes the source clock for the CPU itself. A network of intermediate nodes in the clock tree allows for the modification of the input signal to achieve the requested output signals. This network consists of mainly three different types of nodes: First, there are *clock gates* (`GATE`), the simplest form of modifying an input signal: It can either let the signal pass through the gate to the output or block it off, which can be used to deactivate devices or even complete peripheral busses (i.e., clock-tree subsystems). The second node type in a clock tree is a *scaler* (`PLL_DIV`, `FOSC_DIV`, `DIV`). It modifies the input signal by multiplying or dividing it with a factor before forwarding it to the remaining network. Finally, *multiplexers* (`MUX1`–`MUX3`) receive multiple input signals and, depending on the configuration, select one of them as the output.

This richness of features and configuration possibilities creates high flexibility, enabling trading off between performance and energy efficiency for all devices. It comes, however, with *penalties* for each clock-tree reconfiguration. A penalty describes the time and energy needed to perform the reconfiguration. On the hardware level, modifications to the clock-tree configuration come with varying penalties ranging from miniscule (i.e., few cycles) to significant overheads (i.e., hundreds of milliseconds) [13, 40]. A simple change of a (pre-)scaler, for example, usually requires only a single write to the corresponding divider register. More complex changes, on the other hand, can even require intermediate changes to a temporary helper clock before switching back to the reconfigured original clock (e.g., when switching between PLL clock sources in a microcontroller [33]). These reconfiguration penalties, with regard to both power consumption and time, heavily influence the system's behavior.

3 Problem Statement

In our target domain of embedded energy-constrained real-time systems, applications consist of a set of tasks, the canonically smallest workload unit. Each of these tasks may have different requirements regarding device usage and, thus, different demands on the clock-tree configuration. The naive approach to satisfy device demands is to unselectively choose one configuration that fits all tasks (*all-always-on* approach), but this comes with a higher consumption than necessary for some tasks, for example, because unneeded peripheral devices are enabled. Figure 2 illustrates this problem for a taskset comprising two tasks (`task_`τ_1, `task_`τ_2), with each task requiring one or more devices in addition to the CPU. The upper variant of the `hyperperiod` function displays the *all-always-on* variant mentioned above and

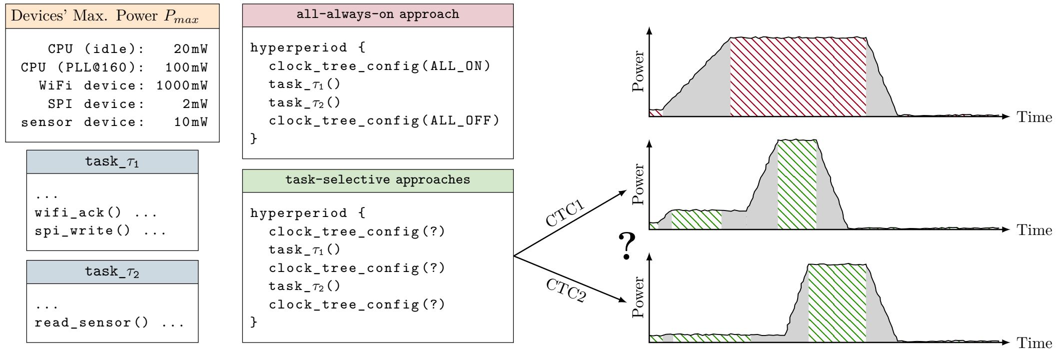


Figure 2 The decision when to apply clock-tree configurations (CTC), for example, to de-/activate devices, significantly influences the system's resource consumption. Different power consumptions and reconfiguration penalties affect both the execution time and the energy demand. The energy demands (i.e., integral over power) of reconfiguration penalties are illustrated as gray areas.

the higher-than-necessary power consumption caused by it. The lower variant is expected to have a lower power consumption, as depicted. The effectiveness of selective clock-tree configurations over the all-always-on approach depends on multiple factors, such as the execution time in one clock-tree configuration and the associated reconfiguration penalties. Thus, break-even points between resource demands determine optimal configurations. In summary, an operating system or runtime environment trying to optimize the handling of these different demands with regard to resource usage faces multiple problems:

Problem # 1: Purely dynamic, feedback-based energy minimization approaches pay for their flexibility with a loss of predictability for the system behavior, which is unacceptable in real-time systems executing under strict time and energy budgets.

Problem # 2: CPU-only, device-independent approaches miss the optimization potential of clock-tree reconfigurations that target arbitrary devices on the SoC.

Problem # 3: Modern clock trees with various features are subject to reconfiguration costs that have adversarial effects on reconfigurations and, thus, have to be selectively considered in a configuration-specific approach.

Problem # 1: Resource-Consumption Guarantees. As FusionClock targets real-time systems, optimizing solely for energy consumption does not suffice and threatens the correct system behavior if deadlines cannot be met. The temporal behavior of devices and the tasks using them depends, among other factors, on the active configuration of the clock tree. In the case of the CPU, for example, a lower frequency allows for a more energy-efficient execution while simultaneously prolonging the task execution. The fact that a task running at a lower frequency jeopardizes not only its own deadline but potentially the deadlines of all following tasks exacerbates the problem. In view of this complexity, only static guarantees enable a safe execution at runtime.

Problem # 2: CPU-only Modeling & Energy-Aware Scheduling. The body of related work for energy-aware real-time scheduling is substantial; we refer to the survey of Bambagini et al. [3] and to the overview of device-aware scheduling techniques [52] for further reading. Common to all these works is either the use of CPU-only models or the use of models that do not cover the complexity of modern clock trees in SoC platforms, especially for configuring devices. The energy consumption of peripheral devices such as those used for

communication (e.g., WiFi or Bluetooth) often heavily outweighs the demand of the CPU. Consequently, to have usable models for real-world scenarios, WCET and WCEC analyses have to consider the whole system, including all devices [48]. Otherwise, they are not able to give safe estimates of the total energy demand.

But even without modeling additional devices, the different configurations of embedded SoC platforms, which are available for online reconfiguration, yield a broad range of energy consumption: For example, the ESP32-C3 [13], a RISC-V microcontroller SoC, uses 100 mW for the highest available CPU frequency of 160 MHz, while only 20 mW are consumed at a CPU frequency of 1 MHz. For the available sleep modes, this demand drops to 1.3 mW for light sleep and 0.15 mW for deep sleep.

Problem # 3: Clock-Tree–Reconfiguration Penalties. To achieve minimal energy consumption, the specific requirements of all tasks in a system concerning device usage have to be considered. In theory, reconfiguring the clock tree enables us to address selective device demands of each task and to operate devices only in the state required by the current task. Knowledge about these device dependencies alone is, however, not enough, as every reconfiguration itself also influences the system behavior. Depending on the concrete parts of the clock tree that need to be changed, the degree of this influence varies. As such, complex changes come with time and energy penalties: Frequently switching clock-tree configurations tailored to the needs of each specific task can even amount to a higher resource consumption than operating the system at the same configuration for all tasks.

Our Approach. In order to deploy task-specific configurations, we *re*configure the clock tree during the system's runtime making substantial use of a-priori knowledge about the tasks: device usage, temporal properties (period, release time, deadline), and bounds on the WCET and WCEC. Modeling the clock tree in a graph, which includes accurate costs for the transitions between configurations, allows us to assess the influence of potential reconfigurations between tasks and consequently decide which reconfigurations are beneficial. By expressing the WCET of tasks dependent on the active clock configuration and considering transition penalties, we are able to determine the slack time in the schedule and use it as optimization potential by shifting task executions and idle phases for a more energy-efficient schedule. Combining all of the above enables us to generate an energy-optimal variant of the system's schedule by inserting reconfigurations and idle phases that optimize energy usage while simultaneously guaranteeing the correct real-time behavior of all tasks.

4 The FusionClock Approach

In the following, we detail FusionClock, our approach to determining application- and device-aware, guaranteed, worst-case–optimal clock-tree–reconfiguration schedules for embedded systems. Fundamentally, FusionClock is based on the realization that within a given time-triggered schedule, the search for a worst-case–optimal clock-tree configuration is equivalent to a *minimum-cost flow problem* within a suitably structured clock-tree–reconfiguration graph that incorporates the required application- and device-dependent knowledge as construction constraints and transition costs. This section is structured as follows: First, we describe the structure of FusionClock's central data structure, the *clock-tree–reconfiguration graph*. FusionClock uses this graph for flow restrictions in a quadratic-programming problem solvable by mathematic optimizers. We then show how an extension of this problem allows

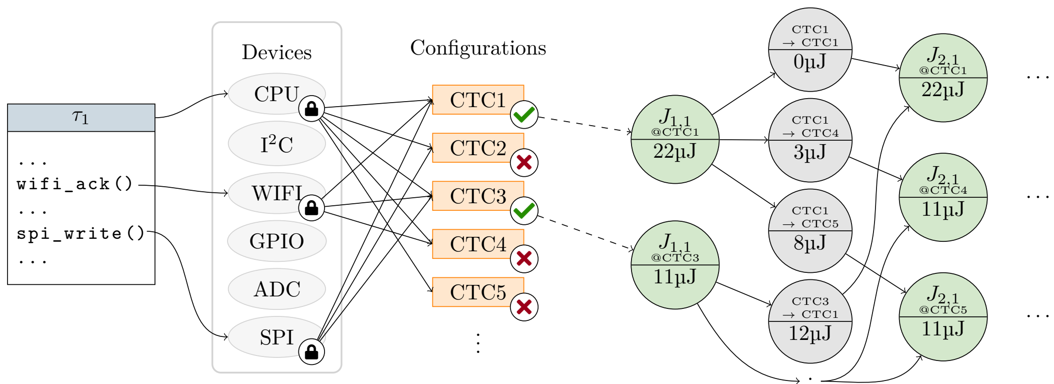


Figure 3 Construction of a *clock-tree–reconfiguration graph* from application- and device-dependent knowledge, here indicated for a job instance $J_{1,1}$ of task τ_1 along with the transitions to the job instance $J_{2,1}$ of the subsequent task τ_2.

the optimizer to redistribute slack within the schedule to reduce a hyperperiod's worst-case energy demand to automatically generate an optimized executable from the solver's output. Finally, we provide a complete formal depiction of the problem.

Clock-Tree–Reconfiguration Graph. The clock-tree–reconfiguration graph provides the basis to determine a schedule's worst-case–optimal sequence of clock-tree configurations (CTC), which is the sequence that will minimize the schedule's WCEC per hyperperiod by selecting optimal reconfiguration points and configurations. At its core, a time-triggered schedule provides a non-preemptive sequence of individual jobs $J_{i,k}$ for the different tasks τ_i within the taskset. In this work, we regard those tasks as the indivisible unit of processing. Here, especially for embedded/cyber-physical systems, it is typical for the individual tasks to interact with various devices within the system, both internal ones such as the CPU device as well as other devices (i.e., sensors, transceivers). However, usage of individual components here requires preparations: As outlined above, tasks can require a specific clock-source configuration to work. At the same time, power gating or frequency scaling different components is vital to minimize energy consumption, an energy-optimal execution of a taskset progresses through a series of different clock-tree configurations. If a task consists of multiple phases with very varied device usage patterns, it thus may be advisable to split those phases into a series of individual subtasks whose CTCs can be optimized individually. Devising a suitable splitting strategy, however, is out of scope of this work. As a first step towards optimization, FusionClock collects the static device-usage information for the individual tasks, which is highlighted with the 🔒 symbol in Figure 3. This is then combined with the SoC-specific knowledge of the individual requirements of a particular device, such as minimum bus frequencies or a specific clock source (e.g., WiFi can require a distinct, highly stable clock [13]), and the corresponding feasible clock-tree configurations. As displayed in Figure 3, these two information sources allow FusionClock to determine the set of feasible clock-tree configurations (✔) for this particular job as the intersection of the different device dependencies in an *application-aware* manner.

Even more, by performing a static WCET analysis and combining the result with the SoC's *resource-consumption model*, we further enrich the graph by attributing upper bounds on the energy consumption of every job's execution phase within the schedule. With this knowledge, it is possible to determine the optimal configuration for every individual phase

by comparing the different consumptions of the various viable configurations (see green nodes with @ symbols in Figure 3). However, as each clock-tree reconfiguration comes with transition penalties, those have to be considered when searching for an optimal configuration sequence. For example, performing a rather energy- and runtime-intensive reconfiguration to reach the optimal operation point for a rather short job can have adverse effects on the energy demand. If the former job was executed in a compatible but slightly more energy-intensive configuration, keeping this configuration can turn out cheaper than paying the reconfiguration penalties. To correctly model those penalties, we add an additional set of transition nodes to our flow graph (gray nodes with → labels): For every pair of job instances of subsequent jobs (in their respective feasible clock-tree configurations), we add a transition node on the edge to which we assign the particular transition costs obtained from a SoC-specific resource-consumption model for the clock tree. That way, a minimal-flow analysis through the graph is guaranteed to retrieve the optimal, penalty-aware reconfiguration sequence for the application's taskset. The right side of Figure 3 shows the initial section of an exemplary clock-tree–reconfiguration graph, displaying the first two job instances of the tasks τ_1 and τ_2.

Linear Constraints. We first express this minimal-flow analysis as an *integer linear program (ILP)*. Later, we extend the ILP and formulate a *quadratic program (QP)* to account for deadlines in multi-rate systems. In the ILP, we add a binary decision variable (n) for every node in the graph that describes whether the particular node is part of the optimal clock-tree–configuration switching sequence or not. Furthermore, we require that for every set of configuration alternatives of a particular job instance in the graph, at least one has to be taken (i.e., their decision variables have to sum up to one). Additionally, we enforce flow-preservation constraints within the graph: We assign binary decision variables to all transition nodes, and each configuration node's decision variable is equal to those of the sum of the transition nodes on its incoming and outgoing edges. Our constraints formulate a single consecutive path through the flow graph. With respect to the flow constraints, the energy-minimization objective of the sum of all binary decision variables multiplied by the respective energy cost of its particular job or transition yields an energy-optimal assignment to the decision variables, sufficient to reconstruct the optimal schedule. A full formalization, along with the QP extension described subsequently, is listed at the end of this section.

Quadratic Constraints for Slack Redistribution. This ILP is not yet sufficient as it neglects important aspects of temporal constraints. Not every energy-optimal configuration sequence guarantees deadline adherence. Also, in the case of multi-rate systems, the ILP formulation does not yet guarantee the correct handling of release times. Therefore, we refine the formulation of FusionClock to incorporate those constraints for the final *QP formulation*.

As indicated, FusionClock iterates upon a pre-existing time-triggered schedule. When considering an exemplary schedule, such as the one displayed in Figure 4, two observations are essential: (1) In addition to the actual job instances and their compute time, the schedule further contains slack time (ts_i). However, as the execution of the schedule is periodic and repeats after the hyperperiod H, the energy consumption of the complete hyperperiod has to be considered and optimized for. (2) As long as neither releases nor deadlines are violated, FusionClock can redistribute slack across this schedule by "compressing" and "stretching" appropriate parts of the schedule within those limits, as illustrated by the springs in Figure 4. Considering that sleep modes represent one of the most energy-efficient clock-tree states but – especially in the case of deep sleep modes – come with high reconfiguration/wake-up penalties, exploiting this slack redistribution is crucial. Therefore, we model and expose idle

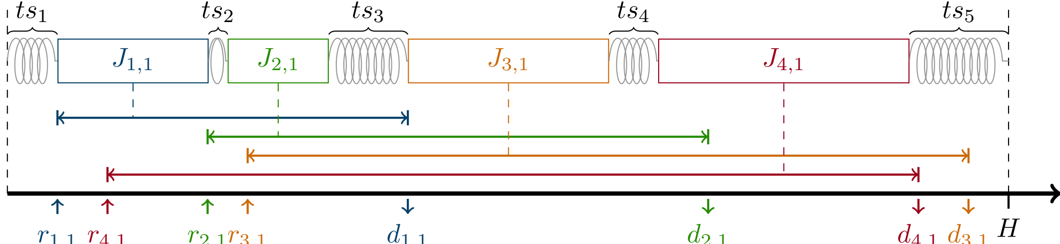


Figure 4 Time-triggered schedules with utilizations below 100 % contain slack time (ts_i). As long as no release or deadline requirements of the underlying tasks are violated, this slack can be redistributed. This image displays a simplified view, as the ILP does not pack fixed-duration jobs but the optimizer is allowed to select different clock-tree configurations for each job instance – each of which can shrink or expand the instance's variable (i.e., clock-frequency–dependent) WCET.

phases as first-class citizens in our optimization formulation: For every idle phase, we add an additional set of alternatives (i.e., different sleep modes and idling variants), along with their reconfiguration/wake-up penalties, to the clock-tree–reconfiguration graph. What sets these phases apart, however, is that they are of variable length: Their combined duration $\sum ts_i$, along with the selected job-instance variants' WCETs $\sum C_{i,j}(conf)$, has to sum up to the system's hyperperiod. That way, the solver is allowed to redistribute the slack for energy minimization. However, this extension comes with the cost of creating a QP based on the initial ILP: When multiplying the variable-length idle times by their selection variables to choose an appropriate sleep mode, we form a multiplication and, thereby, a quadratic optimization problem.

Additionally, we enforce that the scheduled work preceding any job instance's dispatch time (i.e., the time when it is scheduled to start executing) sums up to or surpasses the job instances' release time – this ensures that the optimizer includes sufficient idle time (e.g., $ts_1 + C_{\text{reconf}}(c_{s1}, c_{1,1}) + C_{1,1}(c_{1,1}) + C_{\text{reconf}}(c_{1,1}, c_{s2}) + ts_2 + C_{\text{reconf}}(c_{s2}, c_{2,1}) \geq r_{2,1}$). At the same time, we enforce that when further adding the selected configuration's WCET to that timespan, we still finish before the job instance's deadline: For example, $ts_1 + C_{\text{reconf}}(c_{s1}, c_{1,1}) + C_{1,1}(c_{1,1}) + C_{\text{reconf}}(c_{1,1}, c_{s2}) + ts_2 + C_{\text{reconf}}(c_{s2}, c_{2,1}) + C_{2,1}(c_{2,1}) \leq d_{2,1}$. Thereby, FusionClock guarantees timeliness of the optimized schedule. For providing guarantees, this formulation operates on worst-case values. In practice, job instances may not exercise their full WCET and WCEC. This is, however, not a problem, as idling in the same CTC or entering sleep modes earlier and thus sleeping longer only reduces the system's online energy consumption.

By solving the min-cost flow problem, FusionClock is thus able to determine the global, worst-case–optimal clock-tree configuration with optimized dispatch timings. FusionClock's code generation then extracts this information, in particular the adjusted dispatch timings as well as the CTC-selection variables, from the QP's solution to generate a minimal, tailored implementation of the optimal schedule for the taskset that includes the previously determined clock-tree reconfigurations.

Formalization. Consider a schedule as an alternating sequence of idle phases and job executions, both marked by common but unique indices. For the sake of readability, we omit the mapping of these global job-execution indices to the corresponding task and task-specific job (i.e., from $C_{\hat{\jmath}}$ to $C_{i,j}$), as this mapping is always reconstructable from the available knowledge when constructing concrete formalizations. With this notion and the variables described in Table 1, we are able to formulate the description given above:

$$\min \left(\begin{array}{l} \sum_{\hat{\jmath}\in\hat{\mathcal{J}}} \sum_{c=0}^{f_{\hat{\jmath}}-1} \underbrace{n_{\hat{\jmath},c}\ \mathcal{E}_{\hat{\jmath}}(c)}_{\text{energy costs of jobs}} + \sum_{i\in I} \sum_{c=0}^{f_i-1} \underbrace{n_{i,c}\ ts_{i,c}\ P_{i,c}}_{\text{energy costs of idle phases}} \\ + \sum_{i=0}^{N-1} \sum_{c=0}^{f_i-1} \sum_{c'=0}^{f_{(i+1)}-1} \underbrace{n_{(i,c)\rightarrow(i+1,c')}\ \mathcal{E}_{reconf}(c,c')}_{\text{energy penalty for reconfiguration}} \end{array} \right) \text{wrt.}$$

Linear constraints:

exactly one active configuration per job:

$$\forall i \in \{0,\dots,N-1\}: \quad \sum_{c=0}^{f_i-1} n_{i,c} = 1$$

flow-preservation constraint for incoming edges:

$$\forall i \in \{0,\dots,N-1\}: \forall c \in \{0,\dots,f_i-1\}: \quad \sum_{c'=0}^{f_{i-1}-1} n_{(i-1,c')\rightarrow(i,c)} = n_{i,c}$$

flow-preservation constraint for outgoing edges:

$$\forall i \in \{0,\dots,N-1\}: \forall c \in \{0,\dots,f_i-1\}: \quad \sum_{c'=0}^{f_{i+1}-1} n_{(i,c)\rightarrow(i+1,c')} = n_{i,c}$$

Quadratic constraints:

idle-phase durations and configuration-specific job WCETs sum up to the hyperperiod:

$$\underbrace{\sum_{i\in I} \sum_{c=0}^{f_i-1} n_{i,c}\ ts_{i,c}}_{\text{idle durations}} + \underbrace{\sum_{\hat{\jmath}\in\hat{\mathcal{J}}} \sum_{c=0}^{f_{\hat{\jmath}}-1} n_{\hat{\jmath},c}\ C_{\hat{\jmath}}(c)}_{\text{execution times}} + \underbrace{\sum_{i=0}^{N-1} \sum_{c=0}^{f_i-1} \sum_{c'=1}^{f_{(i+1)}-1} n_{(i,c)\rightarrow(i+1,c')}\ C_{reconf}(c,c')}_{\text{time penalty for clock-tree reconfiguration}} = H$$

preceeding work and idle time sums up to or surpasses release time:

$$\forall \hat{\jmath} \in \hat{\mathcal{J}}: \underbrace{\sum_{i\in I, i<\hat{\jmath}} \sum_{c=0}^{f_i-1} n_{i,c}\ ts_{i,c}}_{\text{idle durations}} + \underbrace{\sum_{\hat{\jmath}'\in\hat{\mathcal{J}}, \hat{\jmath}'<\hat{\jmath}} \sum_{c=0}^{f_{\hat{\jmath}'}-1} n_{\hat{\jmath}',c}\ C_{\hat{\jmath}'}(c)}_{\text{execution times (note the <)}} + \underbrace{\sum_{i=0}^{\hat{\jmath}-1} \sum_{c=0}^{f_i-1} \sum_{c'=1}^{f_{(i+1)}-1} n_{(i,c)\rightarrow(i+1,c')}\ C_{reconf}((c,c')}_{\text{time penalty for clock-tree reconfiguration}} \geq r_{\hat{\jmath}}$$

preceeding work and idle time plus job WCET adheres to deadline:

$$\forall \hat{\jmath} \in \hat{\mathcal{J}}: \underbrace{\sum_{i\in I, i<\hat{\jmath}} \sum_{c=0}^{f_i-1} n_{i,c}\ ts_{i,c}}_{\text{idle durations}} + \underbrace{\sum_{\hat{\jmath}'\in\hat{\mathcal{J}}, \hat{\jmath}'\leq\hat{\jmath}} \sum_{c=0}^{f_{\hat{\jmath}'}-1} n_{\hat{\jmath}',c}\ C_{\hat{\jmath}'}(c)}_{\text{execution times (note the }\leq)} + \underbrace{\sum_{i=0}^{\hat{\jmath}-1} \sum_{c=0}^{f_i-1} \sum_{c'=1}^{f_{(i+1)}-1} n_{(i,c)\rightarrow(i+1,c')}\ C_{reconf}(c,c')}_{\text{time penalty for clock-tree reconfiguration}} \leq d_{\hat{\jmath}}$$

Table 1 Overview of the notation used for FusionClock's ILP and QP formalization. Not shown for the sake of readability is the mapping from $\hat{\jmath}$ to the corresponding job j and task i, which should be trivially available when constructing concrete problem formalizations.

variable	meaning
H	hyperperiod
N	number of jobs and idle phases
$\hat{\mathcal{J}}$	ordered set of global indices corresponding to jobs in start-time order; $\forall \hat{\jmath} \in \hat{\mathcal{J}} : \hat{\jmath} < N$
I	ordered set of indices corresponding to idle phases (with variable length); $\forall i \in I : i < N$
f_i	number of possible clock-tree configurations for job/idle phase i
$n_{i,c}$	binary decision variable for configuration c of job/idle phase i
$n_{(i,c)\to(i',c')}$	binary decision variable for reconfiguration from c to c' between jobs/idle phases i and i'
$C_{\hat{\jmath}}(c)$	WCET of the job corresponding to the global job index $\hat{\jmath}$ in configuration c
$\mathcal{E}_{\hat{\jmath}}(c)$	WCEC of the job corresponding to the global job index $\hat{\jmath}$ in configuration c
$r_{\hat{\jmath}}$	absolute release time of the job corresponding to the global job index $\hat{\jmath}$
$d_{\hat{\jmath}}$	absolute deadline of the job corresponding to the global job index $\hat{\jmath}$
$ts_{i,c}$	duration of idle phase i in configuration c
$P_{i,c}$	power consumption for configuration c in idle phase i
$C_{\text{reconf}}(c, c')$	worst-case time penalty for reconfiguration from c to c'
$\mathcal{E}_{\text{reconf}}(c, c')$	worst-case energy penalty for reconfiguration from c to c'

5 Implementation of FusionClock

To show FusionClock's feasibility and evaluate its performance, we created a prototypical implementation of FusionClock on the ESP32-C3 SoC [13]. Figure 5 shows FusionClock's key components and data structures. After discussing important aspects of the hardware and their implications, this section explains how FusionClock uses this information.

FusionClock's Target Platform. The ESP32-C3 is a RISC-V single-core microprocessor, running up to 160 MHz. It features many devices, such as WiFi, Bluetooth, SPI, UART, and multiple low-power modes, along with frequency-scaling support for the CPU device. The SoC is partitioned into nine power domains, de-/activated in four predefined power modes (i.e., active, modem sleep, light sleep, and deep sleep). This offers for a broad tradeoff between energy consumption and performance, depending on the clock-tree configuration. As such, the ESP32-C3 constitutes a suitable test bed for our clock-tree–reconfiguration approach. For our evaluations, we designed a minimal custom PCB. This allows us to observe energy and timing behavior as accurately as possible while avoiding interference factors such as noisy switching regulators. By using the PCB, the hardware is sufficiently deterministic in its temporal and energetic behavior to derive a reliable, clock-tree–aware resource-consumption model from measurements. We detail this process in Section 6.1. This resource-consumption model is the basis for the *cost-annotated clock-distribution graph*: It captures all timing and energy costs for each clock-tree configuration as well as the reconfiguration penalties. Further, we derive the SoC's *device-dependency graph* from the relevant documentation [10, 12, 13] by manual inspection. It captures the dependence of individual devices on certain properties of the clock-tree configuration (e.g., minimal bus frequencies, power gates). This hardware-related model is the basis for our software-related contributions.

FusionClock's Workflow. FusionClock's input comprises two parts: (1) information about the clock tree, which splits up into the device-dependency graph and the cost-annotated clock-distribution graph, as derived above, and (2) information about the application tasks,

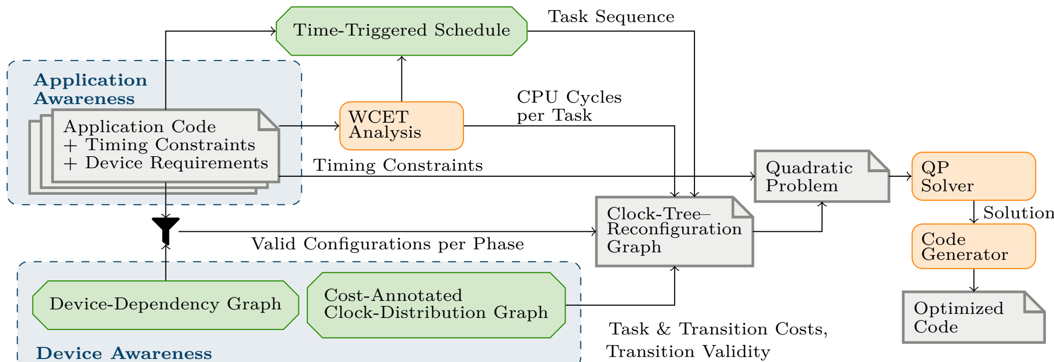


Figure 5 Overview of the FUSIONCLOCK approach with its central inputs, derived data structures, processing steps, and final output result.

their timing constraints, and device requirements. The information on the tasks' requirements is combined (▼) with the device-dependency graph to determine the valid configurations for each task. As a next step, we use our WCET analyzer (see Section 6.1) to obtain WCETs for all individual tasks. Along with the phase order extracted from a time-triggered schedule, this information then allows to construct the system's clock-tree–reconfiguration graph.

As explained in Section 4, the min-cost flow problem of the clock-tree–reconfiguration graph is translated into a formula understandable by a mathematical solver. After executing the solver, FUSIONCLOCK uses its output to extract the selected configurations for each task and the time of all variable-length idle tasks. This allows FUSIONCLOCK to generate code of a specialized system instance by prepending a special *prepare*-hook for each task. It inserts tailored reconfiguration code to transition the SoC to the new clock-tree configuration if the QP's decision variables indicate a reconfiguration. If this reconfiguration did not utilize its full WCET as accounted for in the QP, the task awaits the dispatch time of the subsequent phase to ensure compliance with the time-triggered schedule. Then, the task's workload is evaluated. In a similar fashion, for sleep phases, the system prepares the sleep timer and enters the sleep mode or – in case of active idle – the system idles for the predicted sleep time. This process ensures that the worst-case–optimal system configuration – as determined by the QP's solution – is effectively applied to the target system in a fully automatic manner.

6 Evaluation

This section first describes our evaluation setup (see Section 6.1). Then, we present the evaluation results (see Section 6.2), which consist of three parts: (1) a scalability test for the QP, followed by energy measurements on the SoC for (2) executable tasksets, and (3) a break-even analysis for sleep modes with the help of a benchmark from TACLeBench [14].

6.1 Evaluation Setup

Timing Analysis. Static analyses are conducted using the open-source toolkit PLATIN [38]. We add a custom RISC-V 32-bit architecture for the ESP32-C3 SoC in addition to the previously supported architectures PATMOS [41] and ARM. As the documentation [11, 12, 13] does not provide the microarchitectural details required to create a static hardware model for this chip, the model is based on measurements of individual instruction-cycle timings utilizing a hardware configuration tailored towards deterministic execution. In this configuration, all

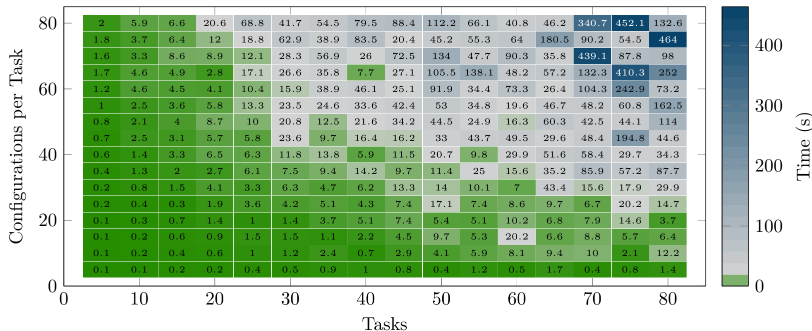


Figure 6 Heatmap showing the solver efficiency for a variable number of tasks on the x-axis and a variable number of possible configurations on the y-axis. With increasing values, the time needed to solve the QP also grows, with a largest value of 464 s, which is still considered acceptable.

application code resides within zero-wait–cycle accessible SRAM, bypassing the need to model flash-access latency and caching behavior. The backing measurements further exercise both pipelining- as well as alignment-related effects that originate from the RV32C (Compressed Instructions) extension [39]. While we cannot guarantee soundness of this model in the current prototype, we did not experience any underestimation in all subsequent evaluations.

Resource-Consumption Model. As the documentation of the energy consumption for the SoC does not provide detailed information about the energy consumption, we used a measurement-based approach to build an expressive energy-consumption model for FusionClock's clock-tree configurations. Here, to derive upper bounds, we base our model on the worst-observed power consumption of the individual clock-tree configurations c weighed by execution time, or expressed formally: $\mathcal{E}_{i,j}(c) = P_{max,c} \cdot C_{i,j}(c)$. Relying on the assumption that P_{max} bounds the maximum configuration-specific power demand and that the WCET bound (C) is safe, this linear approximation determines a valid upper bound of the WCEC. While FusionClock supports expressing the WCEC as a general function ($\mathcal{E}$), this model emphasizes safety over accuracy. To approximate $P_{max,c}$ in a measurement-based manner, we make use of the Joulescope JS220 energy-measurement tool [23], which allows for simultaneously measuring current/voltage, and consequently power demand. To make the model as reliable as possible, the worst-observed energy consumption over multiple runs is taken as a pessimistic reference value for the power consumption of the individual configurations under evaluation. When obtaining WCETs, our model differentiates between two types of tasks: CPU-centric workloads scaling with the underlying clock's frequency, for which we perform the static analyses with Platin as well as fixed-time workloads where a particular device latency determines the (fixed) execution time. For the device-related latencies, we have to rely on worst-observed timings. We found that modeling transition penalties between sleep modes and run modes, with the linear approximation mentioned above ($P_{max} \cdot WCET$), yielded comparably pessimistic results. As a result, we determine these penalties through measurement-based analysis. In all cases, our model provides upper bounds over all observed runs of our evaluation setup.

6.2 Evaluation Results

Solution Efficiency of the Quadratic Program. To solve QPs, a powerful mathematical solver such as Gurobi [17] is required. For evaluation, we deploy Gurobi in version `10.0.0` with a set of synthetic test cases on a machine equipped with two AMD EPYC 7702 64-core

processors with a total of 512 GiB of RAM. As test-case sizes, we use a varying number of jobs up to 80 and a varying number of possible clock-tree configurations per job up to 80 possible configurations, allowing a maximum of 6400 different reconfigurations between two subsequent jobs. We consider these sizes comparably large for the context of energy-constrained real-time systems running on modern SoCs. Each job in the test cases has, besides a synthetic WCET, a randomly generated release time and deadline provided to the solver. To allow the presented dynamic redistribution of slack for each job, we bridge every two jobs with an additional idle phase. The resulting evaluation times of Gurobi are presented in Figure 6. As expected, the time needed to solve the test cases grows with the number of configurations and jobs. The maximum time needed to solve the QP within the given evaluation scenario is 464 s, which we consider practical with regard to the fact that static analyses are conducted once during design time. Due to our approach of producing generic QP formulations, we further benefit from continuous improvements (e.g., parallelization) of mathematical solvers.

Break-Even–Point Evaluation. FusionClock strives to provide energy-optimal clock-tree–reconfiguration sequences for the individual tasksets. However, to likewise provide guarantees, FusionClock's underlying resource-consumption model contains some degree of pessimism. This effect is especially visible close to *break-even points*, that is, instances where the optimal CTC for a certain phase changes due to varying task parameters such as its WCET. Subsequently, we study this effect by the example of a variable-sized sleep phase, as the ESP32-C3's different sleep modes (i.e., active idle, light sleep, deep sleep) with their varying reconfiguration penalties and energy consumptions show a representative evaluation scenario. In this scenario, the system executes a single task, the `binarysearch` benchmark of the TACLeBench benchmark suite [14], before awaiting the hyperperiod's next execution. We then vary the length of the hyperperiod H to determine the break-even points between active idling, light sleep, and deep sleep. Figure 7 displays the maximum measurements over 5 runs for all modes, as well as the optimized estimation of the QP (dotted).

For the computation workload on this SoC, the highest CPU frequency is the most energy efficient in terms of instructions per Joule, and, as a consequence, the solver chooses this configuration for the `binarysearch` benchmark for every tested QP formulation. However, the same is not true for the idle phases: Here, minimizing Joule per time is required, and consequently, FusionClock correctly prefers the lowest CPU frequency in this case (active idle □). Still, regarding this metric, the SoC's two sleep modes fare even better, but their high, static reconfiguration penalties still make them unfavorable for short hyperperiods (i.e., $H \leq 8$ ms). Here, FusionClock accurately predicts the break-even point from active idle □ to light sleep △ (left side of Figure 7). For the second break-even point (around 3200 ms, right hand figure) signaling the switch from light sleep △ to deep sleep ○, we observe a deviation between the theoretical prediction based on our resource-consumption model P_{max} (dotted) and the observed, measured break-even point around values of $H = 6000$ ms. We trace this gap back to our pessimistic hardware model of the ESP32-C3 during light sleep: The maximum observed power consumption is 1.31 mW, while the consumption averages around 0.65 mW with a standard deviation of 0.11 mW, the maximum value manifesting in outliers. To illustrate this point, we introduce two additional variations of our power model for light sleep into Figure 7, where we instead base the model on the average value μ with an additional safety margin derived from the standard deviation σ: $P_{3sig} = \mu + 3\sigma$ and $P_{1sig} = \mu + \sigma$. These two models move the solver estimation substantially closer to the observed break-even point. Thus, this observation indicates one way of further improving FusionClock's current

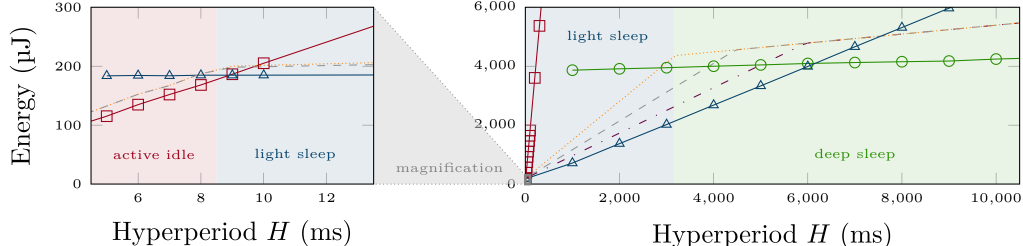


Figure 7 Break-even–point measurements for active idle □, light sleep △, and deep sleep ○ in comparison to QP results with different energy models. P_{max} (dotted) assumes the maximum measured power consumption for each phase for the QP. The models P_{3sig} (dashed) and P_{1sig} (dashdotted) assume a power consumption of $\mu + 3\sigma$ and $\mu + \sigma$ of the measured values for light sleep. In the left part, measurements from 5 ms to 10 ms are in steps of 1 ms, for the right in steps of 1000 ms, with additional measurement points for active idle □ to visualize the higher energy costs compared to both sleep modes. QP results are shown in 1 ms steps (left) and in 100 ms steps (right).

prototype: A more accurate sleep-energy prediction model with less pessimism shifts the reconfiguration sequence towards the observed measurements. In this evaluation, no case of the QP's predicted resource demand shows an underestimation compared to the actual measured demand of the code with its reconfigurations. Thus, the main conclusion for FusionClock from this evaluation is the ability to generate reconfiguration sequences to optimize the energy consumption of the system under observation while accounting for reconfiguration penalties between the respective modes.

Taskset Evaluation. We automatically generate test cases to evaluate whether our approach actually (1) provides a reliable upper bound for the energy consumption and (2) minimizes the energy in comparison to a device-unselective (i.e., all-always-on) application. The generation happens with the following steps: First, we generate tasksets according to an energy-aware generator [50] relying on the UUniFast algorithm [4]. Then, we used a simulation of RMA scheduling [31] at design time to create a time-triggered schedule for these tasksets, splitting tasks where necessary, which eventually creates a sequence of independent tasks. To simulate the tasks' device usage, we assume that device interaction consists of three phases: First *sensing* the environment, then *computing* the resulting action, followed by *actuating* depending on the computation outcome. Therefore, each task is put into one of two groups: 70 % are fixed-time slots, as these devices are assumed to have a non-frequency–scalable timing behavior. The remaining 30 % are considered compute-only tasks where the time depends on the CPU frequency of the SoC. Thereby, we achieve a similar distribution between these three phases, slightly favoring device interactions. As configuration options, we support here 5 clock-tree options in addition to 2 sleep modes, selectively reconfigurable for each task. The hyperperiods range from $H = 25$ ms to $H = 125$ ms over the tasksets consisting of 9 to 18 tasks within a multi-rate system having harmonic periods, which we consider as realistic for our target scenarios. The QP description groups all this information, for which the Gurobi solver then determines an optimized reconfiguration schedule. Next, FusionClock's code generator builds the necessary reconfiguration and idle tasks. For the task's temporal behavior, a loop waits to reach each task's determined WCET. As these evaluations on real hardware require manual interaction with the energy-measurement device over the course of several hours, we conduct this evaluation based on a practically manageable set of ten tasksets, which we randomly selected from the generated tasksets. Figure 8 shows the results of this evaluation by increasing utilizations: The left bars in blue (for the respective

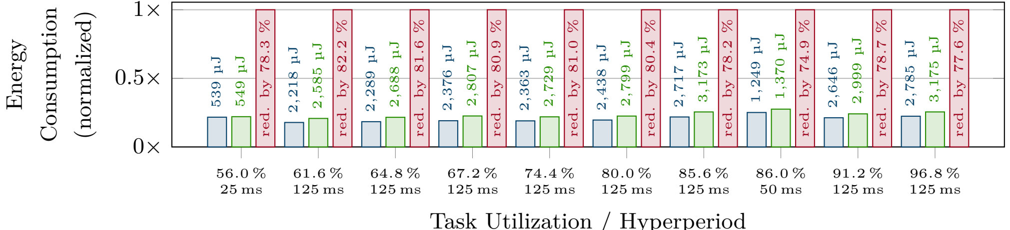


Figure 8 Normalized comparison of an all-always-on approach to the QP solution to the measured energy consumption. The tasksets are ordered by the utilization of the taskset before optimization.

utilization) show the measured, worst-observed energy consumption for each taskset over 50 runs for the utilizations. The middle (green) bars are the solution determined through FusionClock's QP solution, while the right bars show the task-unselective approach to clock configuration, which is also determinable with the QP by restricting the configuration space to the all-always-on option. In all experiments, the QP either selected active idling on the lowest frequency or the light sleep, as the penalty for the deep-sleep mode (i.e., 70.26 ms) is relatively high for these utilizations and hyperperiods. We observe close estimations between the measured and the predicted values: The smallest relative overestimation is in the first shown taskset with 1.8 % (10 µJ in total), while the largest relative difference is in the fourth taskset 15 % (431 µJ in total). The reason for this difference is due to the fact that the tasks consume less than their given budget by the P_{max} energy model. Throughout all evaluated benchmarks, the total measured energy demand is below FusionClock's estimation given by the solver with the help of our resource-consumption model. These values confirm the validity of our modeling approach. Regarding the energy minimization in contrast to the unselective approach, we observed significant improvements: Due to energy savings with the most energy-efficient configuration over time for fixed-time tasks (20 mW to 100 mW) and the energy savings for idle modes, the geometric mean over all observed improvements is 79.4 %. As a main conclusion, we state that FusionClock achieves significant energy improvements, while showing overestimations with acceptable accuracy of the resource model.

7 Related Work

While real-time systems with energy constraints are a well-explored topic, FusionClock goes beyond the current state of the art with its use of static reasoning for guarantees, its penalty-aware handling of multi-source clock trees, its exploitation of application-aware device constraints, and its final code generation. Due to FusionClock's multi-faceted approach to resource-consumption optimization, our work has several scopes of related work: (1) clock-tree (re)configuration, (2) energy-aware real-time scheduling, and (3) static resource-consumption analysis, which are subsequently discussed in this order.

Clock-Tree Reconfiguration. The recent work on *ScaleClock* [40] for the RIOT operating system [1] presents an approach for the online exploration of the clock tree. Instead of building a static clock-tree model, *ScaleClock* employs reconfigurations of the clock tree and thereby learns the model during the system's runtime. In contrast to *ScaleClock*, our FusionClock approach relies on a statically determined clock-tree model with the associated reconfiguration penalties. Having a static model is mandatory in order to give guarantees for timeliness and the execution within energy budgets, which is not possible with *ScaleClock*. As a commonality, we share the same argumentation as *ScaleClock*: Selective

clock reconfigurations play a key role in configuring the tradeoff between energy and time in modern SoCs. From FusionClock's perspective, *ScaleClock* is similar to *Power Clocks* [8] with its management of multiple clocks in embedded systems. In contrast to both existing clock-tree reconfiguration techniques, FusionClock has the major difference of giving static runtime guarantees and finding a resource-optimal configuration. Our energy optimizations are possible due to our underlying resource-consumption model of the target hardware.

Industry also tries to satisfy tool-supported configurations of clock trees: The integrated development platform of CubeMX [45] from STMicroelectronics has an option to brute-force clock-tree configurations until an option satisfying all device constraints is found. This clock-tree option is fixed since no reconfigurations happen during runtime prior to dispatching jobs. Therefore, CubeMX makes suboptimal use of resources. With FusionClock, we exploit the notion of the clock tree along with the tasks' WCET and WCEC under respective configurations in order to yield resource-optimal, job-specific configurations.

Energy Awareness in Real-Time Systems. Energy-aware real-time scheduling is a well-explored topic with a substantial body of related work, as surveyed by Bambagini et al. [3]. In this context, works closer related to FusionClock target the scheduling of *non-DVS components*; we refer to the overview on these techniques from Yang et al. [53]. These scheduling techniques partly share our notion of devices, which show an increase in power consumption when active. For example, these techniques cover the topics of device-aware procrastination [6] and preemption control [52]. However, these works have shortcomings in light of modern SoC platforms featuring feature-rich clock trees, which we address with FusionClock. In contrast to prior work, FusionClock has the expressiveness to cope with multiple clock input sources for energy minimization under timing constraints. Further, FusionClock is able to optimally select clock sources based on the fact that devices can have distinct clock-source constraints (e.g., a specific clock source running at its highest frequency). Additionally, due to our generic clock-tree abstraction, we support arbitrary changes and their respective context-sensitive penalties of clock-tree reconfigurations. Finally, the expressiveness of FusionClock's abstraction seamlessly integrates the handling of low-power/sleep modes. Consequently, FusionClock worst-case optimally answers the question of whether to race or pace to idle [26].

Recent work on energy-constrained real-time systems addresses the energy hotspot detection *EHDE* [44] with static analysis techniques. Similar to FusionClock's argumentation, their work emphasizes the need to account for devices and their dynamic range in power demand for energy-aware real-time scheduling. *EHDE* uses a notion of best-case execution time and WCET to move the activation and deactivation in the control-flow graph to yield energy reductions while maintaining a logically and temporally correct system. *EHDE*'s energy-optimization formulation, which includes the best case, helps to approximate the benefits of the code transformation. In contrast to their work, FusionClock has a comprehensive model of the system's clock tree along with the associated transition latencies under worst-case assumptions. That way, FusionClock is able to jointly account for task dependencies on hardware devices while selecting feasible and worst-case–optimal clock sources along with their configuration (i.e., multiplexer, scalers). Thereby, FusionClock generates systems that execute during runtime energy-optimal clock-tree reconfigurations under real-time constraints based on the tasks' WCEC and WCET.

WCET & WCEC Analysis. The resource-consumption analysis for the WCET is well-explored with numerous presented tools [2, 15, 16, 18, 20, 21, 24, 27, 29, 30, 38]. Likewise, several static WCEC-analysis approaches are presented in literature [22, 35, 47, 48, 49].

However, none of these worst-case approaches addresses the static analysis of penalties when reconfiguring the clock tree. The work closest related to FusionClock with regard to WCET analysis with frequency awareness is *FAST* from Seth et al. [42]. *FAST* is able to accurately model caching behavior, which we solve with our zero-wait–cycle accessible SRAM. In contrast to *FAST*, FusionClock has a notion of multiple clock sources and multiple devices driven by means of the SoC's clock tree.

FusionClock's implementation uses the tasks' WCET combined with the maximum power demand of the respective clock-tree configuration. That way, FusionClock is supposed to yield overestimations but leads to pessimism. Numerous related works exist on energy-cost models for the processors [5, 25, 28, 34, 35, 43, 46]. Employing more fine-grained, instruction-level energy models reduces the pessimism with regard to the energy demand of machine code. However, considering the relations of power consumers (i.e., nW to W) in energy-constrained systems, processing cores only take a minor portion of the whole system. Further reducing FusionClock's pessimism in maximum-power determination is possible with more sophisticated approaches as presented by Cherupalli et al. [7].

8 Conclusion

Clock trees are the heart of modern SoC platforms providing the heartbeat to any connected component. We argue that building FusionClock's clock-tree abstraction and employing this abstraction for energy minimization in real-time systems by means of a mathematical optimization problem advances the state of the art: While existing techniques for clock-tree reconfigurations are able to reduce the energy demand, they are not capable of providing both WCEC and WCET guarantees, being now possible with FusionClock. Thereby, FusionClock yields the optimal energy demand with respect to worst-case assumptions. Further, our abstraction integrates the scaling and gating of clocks for all devices, including the CPU itself, in a uniform way. Having such an abstraction is powerful in light of our whole-system resource-consumption optimization. One part of our future work is the reordering of tasks with respect to their precedence constraints, such that, for example, tasks benefitting from the same (or a similar) clock-tree configuration are executed subsequently. Further, the handling of interrupts is considered future work, which will allow us to handle a broader range of real-time applications. Our evaluation on a hardware platform along with measurements emphasizes what the name FusionClock expresses: Fusing the clock-tree configurations between jobs, which, in turn, can use power-consuming devices (e.g., transceivers, memory controllers, sensors), leads to energy-optimal execution sequences, while maintaining deadlines of tasks. The evaluations further outline that significant (i.e., around 80 %) energy savings are possible compared to clock-tree–agnostic approaches. Besides these savings, FusionClock's analysis time to determine worst-case–optimal solutions in large clock-tree–configuration spaces is considerably short (i.e., few minutes) for static-analysis approaches, which is due to our approach of formulating quadratic problems for fast mathematical solvers. As a future perspective, we envision that more approaches use clock-tree abstractions as the basis for resource-consumption optimizations in embedded systems.

Source code of FusionClock: `https://gitlab.cs.fau.de/fusionclock`

References

1. Emmanuel Baccelli, Cenk Gündogan, Oliver Hahm, Peter Kietzmann, Martine Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C. Schmidt, and Matthias Wählisch. RIOT: an open source operating system for low-end embedded devices in the iot. *IEEE Internet of Things Journal*, 5(6):4428–4440, 2018. doi:10.1109/JIOT.2018.2815038.
2. Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: an open toolbox for adaptive WCET analysis. In *Proceedings of the 8th International Workshop on Software Technolgies for Embedded and Ubiquitous Systems (SEUS '10)*, pages 35–46, 2010. doi:10.1007/978-3-642-16256-5_6.
3. Mario Bambagini, Mauro Marinoni, Hakan Aydin, and Giorgio C. Buttazzo. Energy-aware scheduling for real-time systems: A survey. *ACM Transactions on Embedded Computing Systems (ACM TECS)*, 15(1):7:1–7:34, 2016. doi:10.1145/2808231.
4. Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30:129–154, 2005. doi:10.1007/s11241-005-0507-9.
5. Naehyuck Chang, Kwanho Kim, and Hyung Gyu Lee. Cycle-accurate energy consumption measurement and analysis: Case study of ARM7TDMI. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design (ISLPED '00)*, pages 185–190, 2000. doi:10.1145/344166.344576.
6. Jian-Jia Chen and Tei-Wei Kuo. Procrastination determination for periodic real-time tasks in leakage-aware dynamic voltage scaling systems. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD '07)*, pages 289–294, 2007. doi:10.1109/ICCAD.2007.4397279.
7. Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, and John Sartori. Determining application-specific peak power and energy requirements for ultra-low-power processors. *ACM Transactions on Computer Systems (ACM TOCS)*, 35(3):9:1–9:33, 2017. doi:10.1145/3148052.
8. Holly Chiang, Hudson Ayers, Daniel B. Giffin, Amit Levy, and Philip Alexander Levis. Power clocks: Dynamic multi-clock management for embedded systems. In *Proceedings of the 2021 International Conference on Embedded Wireless Systems and Networks (EWSN '21)*, pages 139–150, 2021. URL: https://dl.acm.org/doi/10.5555/3451271.3451284.
9. Albert Cohen, Xipeng Shen, Josep Torrellas, James Tuck, Yuanyuan Zhou, Sarita Adve, Ismail Akturk, Saurabh Bagchi, Rajeev Balasubramonian, Rajkishore Barik, Micah Beck, Ras Bodik, Ali Butt, Luis Ceze, Haibo Chen, Yiran Chen, Trishul Chilimbi, Mihai Christodorescu, John Criswell, Chen Ding, Yufei Ding, Sandhya Dwarkadas, Erik Elmroth, Phil Gibbons, Xiaochen Guo, Rajesh Gupta, Gernot Heiser, Hank Hoffman, Jian Huang, Hillery Hunter, John Kim, Sam King, James Larus, Chen Liu, Shan Lu, Brandon Lucia, Saeed Maleki, Somnath Mazumdar, Iulian Neamtiu, Keshav Pingali, Paolo Rech, Michael Scott, Yan Solihin, Dawn Song, Jakub Szefer, Dan Tsafrir, Bhuvan Urgaonkar, Marilyn Wolf, Yuan Xie, Jishen Zhao, Lin Zhong, and Yuhao Zhu. Inter-disciplinary research challenges in computer systems for the 2020s. Technical report, National Science Foundation, USA, 2018.
10. Espressif Systems. *ESP32-C3-DevKitM-1*, 2022. v5.0.1. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32c3/hw-reference/esp32c3/user-guide-devkitm-1.html.
11. Espressif Systems. *ESP32-C3-Mini-1 Datasheet*, 2022. Version 1.3. URL: https://www.espressif.com/sites/default/files/documentation/esp32-c3-mini-1_datasheet_en.pdf.
12. Espressif Systems. *ESP32-C3 Technical Reference Manual*, 2022. Pre-release v0.7. URL: https://www.espressif.com/sites/default/files/documentation/esp32-c3_technical_reference_manual_en.pdf.
13. Espressif Systems. *ESP32-C3 Series Datasheet Ultra-Low-Power SoC with RISC-V Single-Core CPU*, 2023. Version 1.4. URL: https://www.espressif.com/sites/default/files/documentation/esp32-c3_datasheet_en.pdf.

14 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 '16)*, pages 1–10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/OASIcs.WCET.2016.2.

15 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.

16 Christian Ferdinand and Reinhold Heckmann. aiT: Worst-case execution time prediction by static program analysis. *Building the Information Society*, 156:377–383, 2004. doi:10.1007/978-1-4020-8157-6_29.

17 Gurobi Optimization, LLC. Gurobi optimizer reference manual. https://www.gurobi.com/.

18 Sebastian Hahn, Michael Jacobs, Nils Hölscher, Kuan-Hsun Chen, Jian-Jia Chen, and Jan Reineke. LLVMTA: an llvm-based WCET analysis tool. In *Proceedings of the 20th International Workshop on Worst-Case Execution Time Analysis (WCET '22)*, pages 2:1–2:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/OASIcs.WCET.2022.2.

19 Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards compositionality in execution time analysis: definition and challenges. *ACM SIGBED Review*, 12(1):28–36, 2015. doi:10.1145/2752801.2752805.

20 Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. The heptane static worst-case execution time estimation tool. In *Proceedings of the 17th International Workshop on Worst-Case Execution Time Analysis (WCET '17)*, pages 8:1–8:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/OASIcs.WCET.2017.8.

21 N. Holsti and S. Saarinen. Status of the Bound-T WCET tool. In *Proceedings of the 2nd International Workshop on Worst-Case Execution Time Analysis (WCET '02)*, pages 36–41, 2002.

22 Ramkumar Jayaseelan, Tulika Mitra, and Xianfeng Li. Estimating the worst-case energy consumption of embedded software. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '06)*, pages 81–90, 2006. doi:10.1109/RTAS.2006.17.

23 Jetperch LLC. *Joulescope JS220 User's Guide Precision DC Energy Analyzer*, 2022. Revision 1.3. URL: https://download.joulescope.com/products/JS220/JS220-K000/users_guide/.

24 Daniel Kästner, Markus Pister, Simon Wegener, and Christian Ferdinand. Timeweaver: A tool for hybrid worst-case execution time analysis. In *Proceedings of the 19th International Workshop on Worst-Case Execution Time Analysis (WCET '19)*, pages 1:1–1:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/OASIcs.WCET.2019.1.

25 Steve Kerrison and Kerstin Eder. Energy modeling of software for a hardware multithreaded embedded microprocessor. *ACM Transactions on Embedded Computing Systems (ACM TECS)*, 14(3):56:1–56:25, 2015. doi:10.1145/2700104.

26 David H. K. Kim, Connor Imes, and Henry Hoffmann. Racing and pacing to idle: Theoretical and empirical analysis of energy optimization heuristics. In *Proceedings of the 3rd International Conference on Cyber-Physical Systems, Networks, and Applications (ICCPS '15)*, pages 78–85, 2015. doi:10.1109/CPSNA.2015.23.

27 Raimund Kirner. The wcet analysis tool CalcWcet167. In *Proceedings of the International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA '12)*, pages 158–172. Springer, 2012. doi:10.1007/978-3-642-34032-1_17.

28 Sheayun Lee, Andreas Ermedahl, Sang Lyul Min, and Naehyuck Chang. An accurate instruction-level energy consumption model for embedded RISC processors. *SIGPLAN Notices*, 36(8):1–10, August 2001. doi:10.1145/384198.384201.

29 Xianfeng Li, Liang Yun, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1):56–67, 2007. doi:10.1016/j.scico.2007.01.014.

30 Björn Lisper. SWEET - a tool for WCET flow analysis. In *Proceedings of the 6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA '14)*, pages 482–485. Springer, 2014. doi:10.1007/978-3-662-45231-8_38.

31 Chang 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.

32 Dominique Méry, Bernhard Schätz, and Alan Wassyng. The pacemaker challenge: Developing certifiable medical devices (dagstuhl seminar 14062). *Dagstuhl Reports*, 4(2):17–38, 2014. doi:10.4230/DagRep.4.2.17.

33 Microchip. *PIC32 Family Reference Manual, Section 6. Oscillators*, 2011. Revision G. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/61112G.pdf.

34 Spiridon Nikolaidis, Nikolaos Kavvadias, Periklis Neofotistos, K. Kosmatopoulos, Theodore Laopoulos, and Labros Bisdounis. Instrumentation set-up for instruction level power modeling. In *Integrated Circuit Design*, pages 71–80, 2002. doi:10.1007/3-540-45716-X_8.

35 James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. Data dependent energy modeling for worst case energy consumption analysis. In *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems (SCOPES '17)*, pages 51–59, 2017. doi:10.1145/3078659.3078666.

36 Sangyoung Park, Jaehyun Park, Donghwa Shin, Yanzhi Wang, Qing Xie, Massoud Pedram, and Naehyuck Chang. Accurate modeling of the delay and energy overhead of dynamic voltage and frequency scaling in modern microprocessors. *IEEE Transactions on Compututer-Aided Design of Integrated Circuits and Systems*, 32(5):695–708, 2013. doi:10.1109/TCAD.2012.2235126.

37 Peter Puschner, Raimund Kirner, and Robert G Pettit. Towards composable timing for real-time programs. In *Proceedings of the 1st International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD '09)*, pages 1–5, 2009.

38 Peter P. Puschner, Daniel Prokesch, Benedikt Huber, Jens Knoop, Stefan Hepp, and Gernot Gebhard. The T-CREST approach of compiler and wcet-analysis integration. *16th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2013*, pages 1–8, June 2013. doi:10.1109/ISORC.2013.6913220.

39 RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*, December 2019. Document Version 20191213. URL: https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf.

40 Michel Rottleuthner, Thomas C. Schmidt, and Matthias Wählisch. Dynamic clock reconfiguration for the constrained iot and its application to energy-efficient networking. In *Proceedings of the 2022 International Conference on Embedded Wireless Systems and Networks (EWSN '22)*, pages 168–179, 2022. URL: https://dl.acm.org/doi/abs/10.5555/3578948.3578964.

41 Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. Patmos: a time-predictable microprocessor. *Real-Time Systems*, 54(2):389–423, 2018. doi:10.1007/s11241-018-9300-4.

42 Kiran Seth, Aravindh Anantaraman, Frank Mueller, and Eric Rotenberg. FAST: Frequency-aware static timing analysis. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS '03)*, pages 40–51, 2003. doi:10.1109/REAL.2003.1253252.

43 Yakun Sophia Shao and David M. Brooks. Energy characterization and instruction-level energy model of intel's xeon phi processor. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED '13)*, pages 389–394, 2013. doi:10.1109/ISLPED.2013.6629328.

44 Mohsen Shekarisaz, Lothar Thiele, and Mehdi Kargahi. Automatic energy-hotspot detection and elimination in real-time deeply embedded systems. In *Proceedings of the Real-Time Systems Symposium (RTSS '21)*, pages 97–109, 2021. doi:10.1109/RTSS52674.2021.00020.

45 STMicroelectronics. *User manual STM32CubeMX for STM32 configuration and initialization C code generation*, 2023. Rev. 39. URL: https://www.st.com/resource/en/user_manual/dm00104712.pdf.

46 Vivek Tiwari and Mike Tien-Chien Lee. Power analysis of a 32-bit embedded microcontroller. *VLSI Design*, 7(3):225–242, 1998.

47 David Trilla, Carles Hernández, Jaume Abella, and Francisco J. Cazorla. Worst-case energy consumption: A new challenge for battery-powered critical devices. *IEEE Transactions on Sustainable Computing*, 6(3):522–530, 2021. `doi:10.1109/TSUSC.2019.2943142`.

48 Peter Wägemann, Christian Dietrich, Tobias Distler, Peter Ulbrich, and Wolfgang Schröder-Preikschat. Whole-system worst-case energy-consumption analysis for energy-constrained real-time systems. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems (ECRTS '18)*, volume 106, pages 24:1–24:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. `doi:10.4230/LIPIcs.ECRTS.2018.24`.

49 Peter Wägemann, Tobias Distler, Timo Hönig, Heiko Janker, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. Worst-case energy consumption analysis for energy-constrained embedded systems. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS '15)*, pages 105–114, 2015. `doi:10.1109/ECRTS.2015.17`.

50 Peter Wägemann, Tobias Distler, Heiko Janker, Phillip Raffeck, Volkmar Sieh, and Wolfgang Schröder-Preikschat. Operating energy-neutral real-time systems. *ACM Transactions on Embedded Computing Systems (ACM TECS)*, 17(1):11:1–11:25, 2017. `doi:10.1145/3078631`.

51 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 Transactions on Embedded Computing Systems (ACM TECS)*, 7(3):36:1–36:53, 2008. `doi:10.1145/1347375.1347389`.

52 Chuan-Yue Yang, Jian-Jia Chen, and Tei-Wei Kuo. Preemption control for energy-efficient task scheduling in systems with a DVS processor and non-dvs devices. In *Proceedings of the 13th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '07)*, pages 293–300, 2007. `doi:10.1109/RTCSA.2007.56`.

53 Chuan-Yue Yang, Jian-Jia Chen, Tei-Wei Kuo, and Lothar Thiele. Energy reduction techniques for systems with non-dvs components. In *Proceedings of 12th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA '09)*, pages 1–8, 2009. `doi:10.1109/ETFA.2009.5347153`.

CAWET: Context-Aware Worst-Case Execution Time Estimation Using Transformers

Abderaouf N Amalou
Univ. Rennes, INRIA, CNRS, IRISA, France

Elisa Fromont
Univ. Rennes, IUF, INRIA, CNRS, IRISA, France

Isabelle Puaut
Univ. Rennes, INRIA, CNRS, IRISA, France

Abstract

This paper presents CAWET, a hybrid worst-case program timing estimation technique. CAWET identifies the longest execution path using static techniques, whereas the worst-case execution time (WCET) of basic blocks is predicted using an advanced language processing technique called Transformer-XL. By employing Transformers-XL in CAWET, the execution context formed by previously executed basic blocks is taken into account, allowing for consideration of the micro-architecture of the processor pipeline without explicit modeling. Through a series of experiments on the TacleBench benchmarks, using different target processors (Arm Cortex M4, M7, and A53), our method is demonstrated to never underestimate WCETs and is shown to be less pessimistic than its competitors.

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

Keywords and phrases Worst-case execution time, machine learning, transformers, hybrid technique

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.7

1 Introduction

The Worst-case execution time (WCET) of a task is its maximum execution time when varying its input data and hardware state. Knowledge of the WCET of all tasks in a system allows schedulability analysis techniques to demonstrate that all tasks will meet their timing requirements in real-time systems. The challenge addressed in this paper is to estimate WCETs for Commercial Off The Shelf (COTS) processors, for which the micro-architecture details are not fully known.

WCET estimation techniques can be divided into three broad categories [36]: *static*, *measurement-based*, and *hybrid* techniques.

Static techniques (ST, e.g., [16, 3]) operate on the Control Flow Graph (CFG) of the task, extracted from its binary code. The nodes in the CFG are Basic Blocks[1] (BB), and the edges represent the control flow between the BB. Static techniques proceed in two phases: in the first phase, the WCET of each BB is estimated using abstractions of the hardware state; in the second phase, the whole program's WCET is calculated by finding the worst path inside the CFG (e.g., this is achieved by employing the commonly used implicit path enumeration technique – IPET – [36]). Although the static techniques produce safe WCET estimates, using hardware abstractions on complex micro-architectures will inevitably lead to state explosion. Moreover, each new architecture demands the design of a new hardware abstraction, which is time-consuming and error-prone (especially without the processor's micro-architectural details).

[1] A basic block is defined as a sequence of instructions with a single entry point at the beginning and a single exit point at the end, without any branching or jumping to other instructions within the block.

© Abderaouf N Amalou, Elisa Fromont, and Isabelle Puaut;
licensed under Creative Commons License CC-BY 4.0
35th Euromicro Conference on Real-Time Systems (ECRTS 2023).
Editor: Alessandro V. Papadopoulos; Article No. 7; pp. 7:1–7:20
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Measurement-based techniques (MBT) (e.g., [9]) are empirical techniques that run the program end-to-end with varied input data and hardware states to gather measurements. The WCET is then estimated from the measurements by either returning the largest observed timing (with a configurable safety margin) or using statistical techniques such as extreme value theory to infer a probabilistic WCET from the observed values [29, 30]. Unless the worst input and hardware state are found, techniques in this category may produce unsafe results.

Hybrid techniques (HT) [4, 32, 11, 20, 21] combine the benefits of ST and MBT: the longest path is safely identified using techniques from ST, like IPET; measurements are used at the BB level, avoiding the costly and error-prone design of hardware abstractions. However, using measurements at the BB level in hybrid methods raises code coverage issues: each BB has to be executed at least once, and each BB's worst-case scenario must be covered.

In recent works, machine learning (ML) techniques are used in HT instead of measurements to predict the WCET of BBs [5, 18, 17, 24, 25, 2]. These techniques, named HT-ML in the following, train an ML model (e.g., neural network) on a large dataset of BB whose WCET is known. The ML model is then used to predict the WCET of previously unseen BB. HT-ML techniques have the following benefits:

(i) The time-consuming phase of HT-ML (training) is executed only once (per architecture) and does not need any design of a hardware abstraction like in ST.

(ii) Although the training phase may be long, prediction is fast and does not require thousands of measurements per BB.

(iii) HT-ML can process large amounts of execution scenarios for BB and identify patterns, allowing more accurate predictions.

Nevertheless, the current HT-ML methods use oversimplified code characterization. The features used for learning and prediction abstract too much information from the machine code, causing information that impacts timing to be lost. For example, not considering the ordering of machine instructions in a BB will make the technique unable to accurately learn the impact of pipelines on timing.

In this paper, we propose a novel HT-ML WCET estimation technique called CAWET, for **C**ontext-**A**ware **W**orst-case execution time **E**stimation using **T**ransformers. This technique uses the advanced machine learning algorithm Transformers-XL [8]. Unlike other HT-ML methods, which only consider static features, CAWET considers the internal dependencies within each BB and the context surrounding it when estimating its WCET. This is performed by treating the sequence of instructions in a BB as a natural language, where the timing of a BB depends not only on its own sequence of instructions but also on the sequence of BBs executed before it.

CAWET consists of two main stages: *training* and *deployment* (or *estimation*). As in all systems using Transformers, the Transformer model is first pre-trained in the learning phase to comprehend the vocabulary (in our context, assembly language). Then, the model is fine-tuned using extensive measurements on various basic blocks extracted from real codes. In this fine-tuning stage, the model learns how to calculate the WCET of each basic block by considering the context surrounding the block (previously executed BBs). During the estimation stage, the WCET of each BB is determined for all bounded-length contexts leading to the BB, extracted from the program's CFG. The maximum timing estimate for these contexts is then selected as the WCET of the basic block and used by IPET to calculate the WCET of the overall program.

CAWET is easy to deploy, as the training has to be done only once. Consideration of pipeline effects is performed automatically because of the consideration of the execution context of all basic blocks.

CAWET is evaluated on processors of varied complexity, including the basic pipeline-only cortex-M4, the more advanced cortex-M7 that features a cache, and the even more sophisticated cortex-A53. The quality of WCET estimates produced by CAWET is compared to those produced by WE-HML, the HT-ML technique closest to CAWET [2], on 13 programs from the TACLeBench benchmark suite [13]. Our results show that CAWET produces better estimates than its competitors on more diverse architectures.

Our contributions are:

- A new hybrid timing ML-based WCET analysis technique that uses Transformers-XL to estimate the WCET of basic blocks and considers dependencies between instructions.
- We take into account the execution context that surrounds the BB under analysis by automatically exploring all bounded-length paths that leads to it.
- We provide an empirical study on different targets and techniques. Our results show that this complex ML method is well-suited for timing estimation, with an average error of 23.8%, 102.2%, and 62.4%, on the Cortex M4, Cortex M7, and Cortex A53 processors, respectively.

The rest of this paper is organized as follows. Section 2 presents the CAWET HT-ML technique. The experimental methodology for evaluating it is detailed in Section 3, and experimental results are given in Section 4. Section 5 compares our approach to related techniques. We conclude in Section 6.

2 CAWET: Context-Aware WCET estimation using Transformers

CAWET is a hybrid context-aware WCET estimation technique that predicts an in-context WCET of individual basic blocks and then uses the predictions to calculate the overall program's WCET. A high-level overview of CAWET is given in Section 2.1. The two main phases of CAWET: training (using Transformers-XL) and prediction (i.e., deployment), are then respectively presented in Sections 2.2 and 2.3.

2.1 Overview of CAWET

CAWET consists of two main stages: *training* and *deployment* (or *estimation*). Both stages operate on individual basic blocks (BB) and account for the execution context of the BB under study (i.e., the sequence of BBs executed before it). CAWET relies on Transformers-XL, originally used in natural language processing, for their ability to learn long-term dependencies between words. In CAWET, the language under study is a sequence of BBs, each composed of a sequence of assembly instructions. The overall structure of CAWET is depicted in Figure 1.

In the training phase (left block of Figure 1), the Transformer model is first pre-trained on real programs to learn the vocabulary of the language it will process (in our context, assembly language) as it is usually done for large language models [10]. Then, the model is fine-tuned using extensive measurements on a large set of BBs extracted from real code. In this fine-tuning stage, the model learns how to calculate the WCET of each BB by considering the context surrounding it (i.e., previously executed BBs).

During the estimation stage (right block of Figure 1), the WCET of each BB is determined. Since there might be different execution paths leading to the BB under study, prediction operates on the set of contexts corresponding to these paths, with care taken to avoid combinatorial explosion, as further explained in Section 2.3. The prediction phase first computes the list of contexts of the BB under study (BB number 8 in the Figure). The result

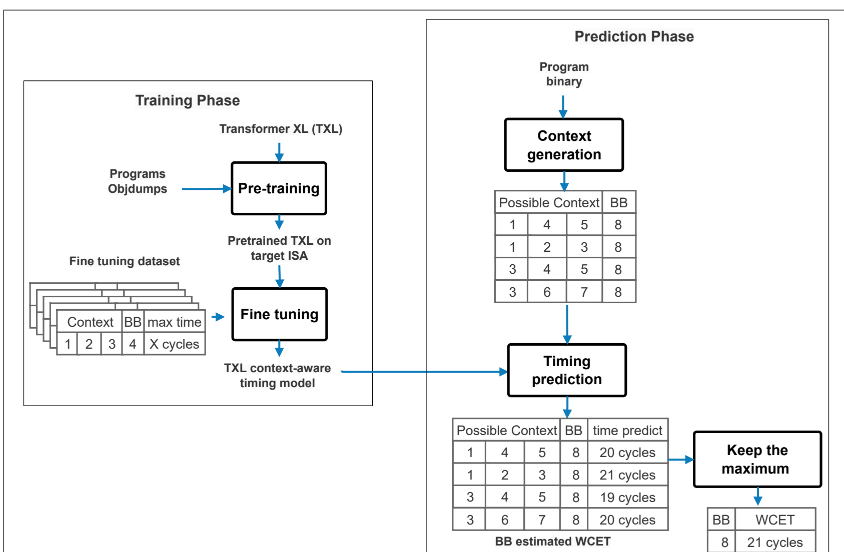


Figure 1 Overview of CAWET.

in the example is a list of 4 contexts, made of the sequence of BBs executed before BB 8: $(1, 4, 5), (1, 2, 3), (3, 4, 5)$, and $(3, 6, 7)$. The timing of BB 8 is estimated for each context. The maximum timing estimate is then selected as the WCET of the BB and used by IPET to calculate the WCET of the overall program.

2.2 Training phase using Transformers-XL

A *transformer* is a neural network architecture originally designed for natural language processing, which can perform tasks such as language translation, text summarizing, and text-to-speech. It was first proposed in [35], and one of its main advantages is using self-attention mechanisms that enable the model to weigh different parts of the input data when making predictions. However, as defined in [35], the original transformer architectures have a fixed-length context window and may struggle to handle sequential data with long-term dependencies. To address this limitation, *Transformers-XL* (TXL) [8] were introduced. A TXL is a variation of the transformer architecture that uses a so-called *memory-augmented attention* to better remember and utilize information from earlier in the sequence. We use a TXL architecture in CAWET because it improves the ability of the transformer to handle long-term dependencies, which is necessary for handling long sequences of code.

Estimating the WCET of a given BB given its context is performed by first processing the context (formed by the BB executed before the analyzed BB as well as the analyzed BB), followed by processing the BB under analysis. This results in two embedding matrix representations (a global attention matrix for the context and a local attention matrix for the BB under analysis) that are then concatenated. The resulting embedding representation is given as input to a fully connected layer, producing a single scalar value (the timing estimate for the analyzed BB). Figure 4 provided in the Appendix illustrates this process.

The training of a TXL consists of two stages (*pre-training* and *fine-tuning*). During the pre-training stage, the TXL is trained to learn the structure of assembly instructions in text format using self-supervised learning. This classical self-supervised learning phase [10] is achieved by masking random operations or operands in the sequence and (pre)training the model to reconstitute (i.e., predict) them as output. To perform this pre-training phase,

thousands of disassembled binary programs are used without needing labeled information. Details about the hyper-parameters of the TXL architecture are provided in Table 10 in the Appendix.

In the fine-tuning stage, a set of programs, the target processor, and a measurement tool are required. BBs execution time is measured using the measurement tool. Then, the instruction sequences are tokenized using *sentence piece* [23], a well-known tokenization technique trained in our work on the target assembly instructions. The training dataset for the fine-tuning stage is then built using the maximum observed timing of each BB, the tokenized BB, and its context. Contexts have a maximum size; the context size, expressed as a number of basic blocks, is a hyperparameter of the Tranformer-XL.

2.3 Prediction phase

CAWET predicts the WCET of BBs by considering their execution context. The results from CAWET can then be used by a static WCET estimation tool. Section 2.3.1 presents the concepts and notations CAWET relies on. Section 2.3.2 then details the context generation. Section 2.3.3 describes how the WCET of a BB is obtained from the predictions and the overall WCET of the program is finally calculated.

2.3.1 Concepts and notations

The concepts and notations used in CAWET are standard concepts used in compilers. They are illustrated in Figure 3, which will be reused later to illustrate how CAWET works.

▶ **Definition 1** (Control Flow Graph). *A Control flow graph (CFG) is a directed graph where each node represents a BB, and each edge represents the control flow from one BB to another.*

▶ **Definition 2** (SESE regions, SESE trees). *A Single Entry Single Exit (SESE) region, as defined in [19], is a sub-graph of a CFG that can only be entered by one edge and exited by one edge. A property of SESE regions is that they can be arranged into a tree, constructed in linear time.*

An example of CFG (with 7 BBs numbered from 1 to 7), and its SESE regions is depicted in Figure 2 (A). The dotted arrow in the figure represents the back edge of the loop composed of BB 5 and 6. The SESE tree that corresponds to the CFG is depicted in Figure 2 (B). The rationale behind using SESE regions is to have subsets of the CFG that are simple enough to explore all paths exhaustively, with the overall objective of avoiding combinatorial explosion when generating the possible contexts of a BB.

▶ **Definition 3** (Cyclomatic complexity). *Cyclomatic complexity is a software metric that measures the number of independent paths through a program or a CFG [12]. It can be thought of as the number of unique paths that can be taken through the code. It is calculated using the following formula:* $Cyclomatic_complexity(CFG) = edges - nodes + 2$

The cyclomatic complexity will be used during the prediction phase to decide which paths leading to a BB are worth exploring. The cyclomatic complexity of the SESE regions in our example is displayed in Figure 2 (B).

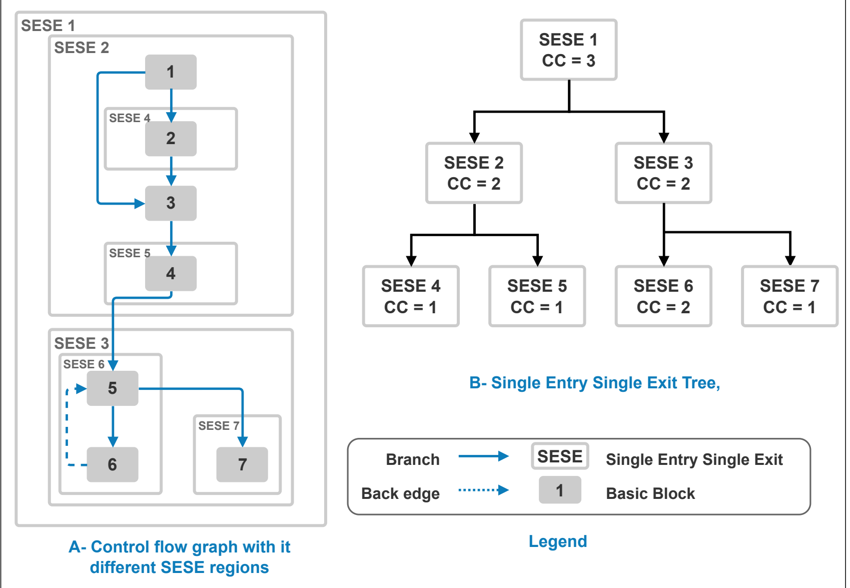


Figure 2 A CFG example transformed into a SESE tree and annotated with cyclomatic complexity.

2.3.2 Context generation

The task of finding all the possible paths in a graph may be computationally expensive. To address this issue, we use a divide-and-conquer strategy based on the SESE tree of the program. In the example of Figure 2, the root SESE region (SESE 1) represents the entire CFG. Each tree level represents a sub-SESE region (e.g., SESE 2 and SESE 3 are the children of SESE 1), with smaller and thus simpler sub-graphs.

To limit the complexity, CAWET performs an exhaustive path exploration only for the SESE regions that are simple enough (based on their Cyclomatic Complexity, CC) to allow a full path exploration. SESE selection is performed using a top-bottom traversal of the SESE tree, and the SESE regions with a value of CC strictly higher than a threshold are filtered out. Path exploration for the selected regions uses Depth-First Search [34] (DFS) to enumerate all possible paths[2]. We ensure, by construction, that the chosen SESE covers the entire input code. i.e., in situations where a SESE node cannot be analyzed due to its high CC value, we analyze all its children. Additionally, basic blocks that do not belong to any region in the tree are included to ensure complete code coverage.

This process is illustrated in Figure 3 step 1 using the CFG and SESE in Figure 2 as an example, with a CC threshold of 2. In this example, the SESE regions 2 and 3 are selected, and their paths are fully explored (step 2 in Figure 3).

Management of loops

As explained above, the enumeration of paths in SESE regions ignores the back edges of loops. Therefore, all paths in a given loop are explored only for one iteration. Obtaining the execution context of any BB to be executed after a loop requires considering several loop

[2] DFS traversal ignores loop back-edges. Loop management is described later in this Section.

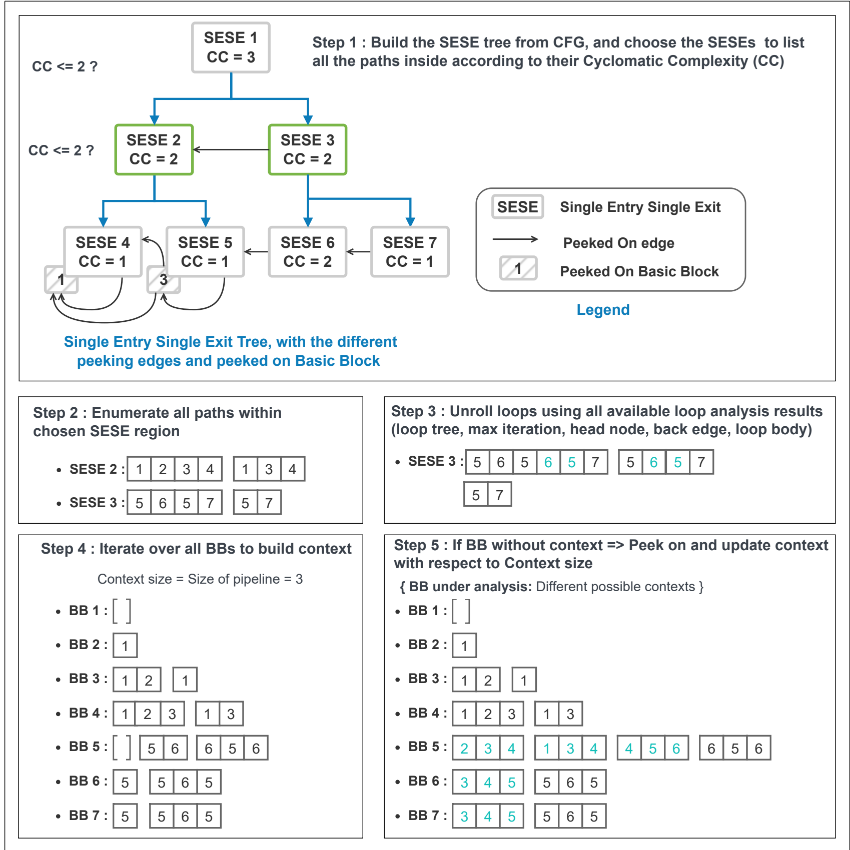


Figure 3 Example of the different steps for context generation, where the cyclomatic complexity limit is set to 2 and the context size is set to 3 BBs.

iterations. This is achieved in CAWET using (virtual) *unrolling*: the context of a loop is composed of several iterations of the loop body (from zero to the loop's maximum number of iterations).

As the path followed may differ across iterations, generating all possible contexts may lead to a combinatorial explosion. This issue is addressed by restricting the number of BBs added by the unrolling process for the loop body to a fixed value, the hyperparameter *context size* of CAWET. In the presence of nested loops, the context of the inner loops is generated first, to be further used to generate the context for outer loops. This is performed using a bottom-up traversal of the *loop nesting tree* of every CFG[3].

The result of the loop unrolling process on our example is given in Figure 3 step 3, for SESE 3. Three contexts are generated, corresponding respectively to 1, 2, and 3 executions of the loop. Note that, at this step, the size of the contexts of SESE regions may be longer than the *context size* hyperparameter.

Per BB context generation

The execution traces for the different SESE regions, after loop unrolling, are used to generate the *context list* of every basic block, as depicted in Figure 3 step 4. The size of each context is limited to the *context size* hyperparameter of CAWET.

[3] A *loop nesting tree* is a tree data structure used to represent nested loops. Each node in the tree represents a loop, and the edges between the nodes represent the nesting relationship between the loops.

In some cases, the initial nodes of some SESE sub-regions are smaller than the *context size* hyperparameter. To address this issue, we look for the preceding SESE region or BB to access the end of its traces. The peeked-on edges are shown in Figure 3; they can easily be found by looking at the end of the traces of all the BB that occur before this trace. The obtained information can then be used as context for the start nodes of the current SESE region, provided we can find a region before the current one.

As an example, Figure 3 step 5 shows that the context of BB 5 can be augmented by peeking at the execution trace of SESE 2.

2.3.3 Basic Block WCET estimation and program WCET calculation

After generating all possible limited-size contexts for each BB, we move on to estimating its WCET. This involves predicting the execution time of the BB under study for all its contexts. In an architecture without a cache, the maximum estimated time is selected as the worst-case scenario. If the target architecture includes a cache, we keep track of the two highest estimated execution values to account for cache effects. The largest value represents the first execution of the basic block within a loop, which is typically long, while the other value represents subsequent executions of the same basic block, which may be shorter[4]. The WCET of BBs is then fed into a static WCET estimation tool to calculate the WCET of the overall program using standard techniques such as IPET [36].

3 Experimental setup

This Section provides a comprehensive description of the experimental setup used to evaluate CAWET on multiple ARM Cortex targets, specifically M4, M7, and A53. The programs used to train CAWET and evaluate the quality of predictions are first described in Section 3.1. The context-agnostic baselines CAWET is compared to are presented in Section 3.2, followed by an introduction to the software and hardware environments in Section 3.3. The setups for the learning and prediction phases of CAWET are presented respectively in Section 3.4 and 3.5.

3.1 Dataset and benchmarks

CAWET training consists of two steps: (self-supervised) pre-training and fine-tuning. We have pre-trained CAWET on a large number of BBs in order for the Transformer to learn the assembly language under study, using CodeNet [28]. CodeNet is a collection of solutions submitted by the public to competitive programming websites. It contains approximately 900,000 C programs, which we cross-compile to the target architecture and disassemble using GNU binary utilities using *objdump*. The textual format produced by *objdump*, after some basic parsing (e.g., extraction of addresses, separation of BBs) allows the creation of a large pre-training set. This pre-training set is used to build a vocabulary model with *sentence piece* [23]. Once the model (sentence piece model) has been trained, it is then used to tokenize any binary programs written with the target instruction set. To fine-tune CAWET on basic blocks with their context, we have used a diverse and publicly available set of programs:

[4] Since the context size is limited, the predicted timing values may be too optimistic. We, therefore, analyze in Section 4.2 and 4.4 a technique that applies static cache analysis, and we add the overhead obtained by this analysis to the timing values produced by CAWET.

The Algorithms[5], *MiBench* [15] and *Polybench* [37]. Table 1 gives a short description of each benchmark suite, the number of programs it contains, and the total number of BBs encountered when executing the programs.

Table 1 The benchmarks used for training CAWET.

Dataset name	Description	Nb. of programs	Nb. of BB
The Algorithms	Collection of open-source implementations of a variety of algorithms implemented in C	200	12123
PolyBench	A collection of benchmarks containing static control parts. The purpose is to uniformize the execution and monitoring of kernels	30	11224
MiBench	A free, commercially representative embedded benchmark suite	14	8324
Total		244	31671

Table 2 Selected TacleBench codes used to evaluate the quality of the predictions.

Name	Description
bs	Binary search in an array
bsort	Bubble sort algorithm
countnegative	Basic counting on arrays
crc	Cyclic redundancy codes
expint	Exponential integral function
fdct	Fast discrete cosine transform.
fir	Finite impulse response filter
h264 dec	H.264 block decoding functions
insertsort	Insertion sort
jfdctint	Discrete-cosine transformation
matrix1	Generic matrix multiplication
ns	Search in 4-dimension array
petrinet	Petri net simulation

To validate the quality of the WCET predictions provided by CAWET, we use a subset of the codes from the TacleBench benchmark suite [13] whose characteristics are given in Table 2. We chose these codes because: (i) the programs are analyzable by static WCET estimation tools, and in particular, they contain loop-bound annotations; (ii) they come with input data known to trigger the worst-case execution paths; (iii) they are used in our closest competitor WE-HML [2], allowing us to compare CAWET with this work. Note that the selected TacleBench programs were not used during any of the two steps of the training phase.

3.2 Context-agnostic baselines

CAWET is evaluated by comparing it to two context-agnostic WCET predictors. The first one is a Multi-Layer Perceptron regressor (loosely called a neural network (NN)). Although not a naive approach, the neural network is a feed-forward architecture that does not incorporate sequential information and requires a fixed-size input. Our implementation of the NN employs a total of 233 static features of the basic blocks as input, including the proportion of different machine instruction types (e.g., MOV, ADD, LDR). We used a greedy search algorithm to determine optimal hyperparameters for the NN, including the number of

[5] Available here: `https://github.com/TheAlgorithms/C`

hidden layers, optimizer, learning rate, and loss function. Based on the validation dataset, the ideal parameters were determined to be hidden layer sizes=(512, 256, 128), learning rate='adaptive', learning rate init=0.001, solver='adam'. The other baseline CAWET is compared with is WE-HML, a hybrid ML-based WCET estimation technique presented in [2]. The best performing ML algorithm of [2] (Neural Network trained to account for cache effects) is used. CAWET is compared to WE-HML for the Cortex A53 processor only, a processor for which the results of WE-HML were available.

3.3 Hardware and software setups

Accurate timing values must be employed whenever possible when training and validating CAWET, and the method used to obtain the timing values should not interfere with the execution of the code, a phenomenon commonly known as the *probe effect*. CAWET either uses a hardware-based approach or a software solution when the hardware-based solution is not accessible. The hardware solution leverages the Joint Test Action Group (JTAG) interface. The J-Trace Pro trace solution from Segger [31] is used to connect to the JTAG interface of the target processor (in our case Cortex-M4 and Cortex-M7), in conjunction with Ozone [14], a cross-platform debugger and performance analyzer. Ozone generates execution traces that provide the value of the cycle counter, the instruction's address, opcode, and operands, as well as the corresponding assembly code for each instruction. The software solution involves adding code instrumentation to measure the execution time of individual basic blocks (BB) in a program. To provide context and assembly code for the timed BB, we retrieve the execution trace using GDB (the GNU Debugger). The software solution is only used when no JTAG interface is available since it is prone to probe effects and requires significant human effort to implement.

Our experiments are performed on various Arm processors, whose characteristics are summarized in Table 3. We initially focus on the Cortex-M4 processor, which has a simple in-order pipeline with three stages and no cache. This processor allows us to validate our method on a deterministic processor with precise timing measurements through the JTAG interface. Then, we evaluate our approach to the more advanced Cortex-M7 processor. This processor features a 6-stage in-order pipeline, data and instruction caches, and a branch predictor. Finally, we use a more complex processor, Cortex-A53, which is hosted in a Raspberry Pi 3. This superscalar processor has two data and instruction cache levels: an 8-stage in-order pipeline and a branch predictor. The Cortex-A53 has no JTAG interface; the reading of the cycle counter is used for the timing measurements. Using this commercial off-the-shelf (COTS) hardware is part of the experiments in the WE-HML approach [2].

Table 3 Summary of the processors used and their micro-architectural features.

Target	Measurement solution	OS?	Pipeline/#stages	Branch predictor	Cache memory and proprieties
Cortex-M4	Hardware (JTAG)	Baremetal	In-order/3	No	No
Cortex-M7	Hardware (JTAG)	Baremetal	In-order/6	Yes	Yes data and instruction cache, L1, random replacement policy
Cortex-A53 (also used in [2])	Software	Linux	In-order/8	Yes	Yes data and instruction cache, L2, random replacement policy

3.4 Setup for the learning phase

PyTorch was used to implement the learning models, which were then trained on a Tesla V100. Each setting (processor) required two days for CAWET training: 1,5 days for pre-training and 0,5 days to fine-tune the model. To avoid underestimating execution times, we employed the

Root Mean Squared Logarithmic Error (RMSLE) loss function provided in Equation 1, which tends to penalize underestimations more heavily than overestimations. We also incorporated an additional penalty for predictions that underestimated the execution time, according to Equation 2. We artificially modify the target value in the loss when the prediction is too low. When computing the loss, this is done by increasing the target with the predicting error $(target - prediction)$.

$$RMSLE(target, predict) = \sqrt{(\log(target + 1) - \log(predict + 1))^2} \tag{1}$$

$$UsedTarget = \begin{cases} target & if \quad target \leq prediction \\ target + (target - prediction) & if \quad target > prediction \end{cases} \tag{2}$$

3.5 Setup for the prediction phase

The CFG, the SESE tree, and the loop tree is generated by the Heptane WCET estimation tool [16]. These structures are used to construct the list of contexts for each BB. Then, we predict the WCET for each BB using CAWET. Finally, we employ Heptane's IPET to determine the overall WCET of the program.

To create the contexts, we opted for a cyclomatic complexity of 5, as this value has been shown empirically to generate paths within a reasonable amount of time (less than five minutes to generate traces for each basic block in the 13 programs previously described). Since the best context size varies across different architectures, we only considered a fixed number N of consecutive basic blocks, where N corresponds to the number of pipeline stages.

4 Results

The quality of WCET predictions for the Cortex M4 and Cortex M7 architectures is evaluated in Sections 4.1 and 4.2. The effect of the different features of CAWET on the quality of the predictions is studied in Section 4.3. Finally, CAWET is evaluated in Subsection 4.4 on a more complex processor, the Cortex-A53, using a software measurement method and an operating system, allowing us to compare the WCET predictions of CAWET with those of WE-HML [2].

4.1 Quality of WCET predictions for the Cortex M4

Table 4 compares the WCET predictions of the selected TacleBench programs on the deterministic cache-less architecture Cortex M4. WCET predictions of BBs are either obtained by CAWET or by the context-agnostic Neural Network (NN) baseline described in Section 3.2. The table gives for the two techniques both the WCET prediction in cycles and the Relative Percentage Error RPE defined as $RPE = \frac{(Predict - Actual)}{Actual} * 100$. A context size of 3 BB is used.

The results show that CAWET is twice less pessimistic than the NN baseline on average, using the Mean Absolute Error[6] on the RPE (i.e., Error = RPE). This can be explained by the fact that: (i) neural networks do not consider the ordering of instructions in BBs (ii) neural networks are context-agnostic. We also observe that neither CAWET nor the NN baseline underestimates the WCET since all RPE are positive.

[6] Mean Absolute Error: MAE $= \frac{1}{n} * \sum^{n} |Error|$

Table 4 Comparison of WCET predictions for CAWET and a Neural Network (NN) baseline on TacleBench programs for Cortex-M4.

Benchmark	Maximum observed execution time (Cycles)	NN estimations (Cycles)	NN RPE (%)	CAWET estimations (Cycles)	CAWET RPE (%)
bs	140	307	119.2	272	94.3
bsort	317279	414882	30.7	374712	18.1
countnegative	9638	14047	45.7	12858	33.4
crc	78496	102005	29.9	92872	18.3
expint	5683	7758	36.5	5727	0.7
fdct	7308	10557	44.4	8606	17.7
fir	6882	10844	57.5	7490	8.8
h264_dec	573752	661037	15.2	607918	5.9
insertsort	3125	3964	26.8	3898	24.7
jfdctint	7761	11454	47.5	9968	28.4
matrix1	440243	577831	31.2	564921	28.3
ns	28444	45026	58.2	34367	20.8
petrinet	3283	4159	26.7	3592	9.4
Avg. MAE	–	–	43.80	–	**23.8**

Impact of the context size. Table 5 shows the considered context size's impact on the prediction quality. Four values are considered: 0 (no context), 1 BB as context, 3 BBs as context, and 20 BBs as context.

Table 5 Impact of the context size on the Mean Absolute Error (MAE) on TacleBench programs for Cortex-M4.

Benchmark	Context 0	Context 1 BB	Context Pipeline size (3)	Context 20 BB
bs	104,2%	97,6%	94,3%	117,9%
bsort	22,4%	27,6%	18,1%	34,2%
countnegative	47,3%	38,9%	33,4%	46,2%
crc	19,6%	11,1%	18,3%	19,3%
expint	21%	15,9%	0,7%	21,6%
fdct	39,2%	28,4%	17,7%	38,2%
fir	34,5%	31,6%	8,8%	39%
h264_dec	30,2%	22,1%	5,9%	30,9%
insertsort	15,5%	25,6%	24,7%	27,4%
jfdctint	34,6%	31,9%	28,4%	41,9%
matrix1	36,1%	33,3%	28,3%	53,4%
ns	45,7%	33,8%	20,8%	41,3%
petrinet	11%	17,2%	9,4%	16%
Avg. MAE	35,5%	31,9%	**23,8**%	40,6%

The results show that, on average, the error is minimal when the context size is 3 BBs. Accounting for the execution context of BBs is beneficial to the quality of the predictions up to a context size of 3. Taking into account larger context sizes results in much higher error values. One possible explanation for these higher error values is that the context vector is being disrupted by extensive information that cannot be processed efficiently with the current TXL architecture. In future works, we plan to examine this phenomenon more closely, which will require substantial computing resources.

4.2 Quality of WCET predictions for the Cortex M7

The Cortex M7 processor is more complex than the Cortex M4. It features a 6-stage in-order pipeline, data, and instruction caches with random cache replacement and a branch predictor. Table 6 evaluates WCET predictions produced by CAWET and the baseline NN for the Cortex M7, using a context size of 6 for CAWET.

Table 6 Comparison of WCET predictions for CAWET (vanilla) and a Neural Network (NN) baseline for Cortex-M7.

Benchmark	Maximum observed execution time (Cycles)	NN estimations (Cycles)	NN RPE (%)	CAWET estimations (Cycles)	CAWET RPE (%)
bs	140	307	119.3	280	100.0
bsort	191406	464616	142.7	376784	96.9
countnegative	6956	15874	128.2	13904	99.9
crc	47476	98473	107.4	88668	86.8
expint	3592	8260	130.0	7140	98.8
fdct	4957	12044	143.0	9341	88.4
fir	4625	10856	134.7	9132	97.4
h264_dec	362349	779905	115.2	706162	94.9
insertsort	1760	4188	138.0	3414	94.0
jfdctint	4011	11877	196.1	10215	154.7
matrix1	301866	660739	118.9	644668	113.6
ns	21253	46004	116.5	41167	93.7
petrinet	1595	3741	134.5	3342	109.5
Avg. MAE	–	–	132.7	–	**102.2**

The results show that even with no explicit support for caches, CAWET never underestimates compared to the Maximum observed execution time (the max of 1000 executions) and is again more precise than the NN baseline. It should also be noted that the average MAE, both for CAWET and NN, is, as one would expect, higher for the more complex Cortex M7 than for the very simple Cortex M4, showing that the tight timing analysis of complex processors is harder to achieve than the analysis of simpler ones.

Since the context size in CAWET is limited, the reuse of code/data (with instruction/data caches) may not be fully taken into account by the model. We thus modified CAWET to add a cache miss penalty to the WCET of a BB when the static cache analysis of Heptane cannot guarantee a cache hit. The same procedure is applied to the NN baseline, and the results are reported in Table 7.

The integration of cache analysis results into CAWET and NN leads to more pessimistic WCETs for both techniques. Two factors explain this additional pessimism: (i) the static cache analysis for random cache replacement is inherently pessimistic; (ii) CAWET already captures parts of the cache behavior due to its use of the execution contexts for BBs. Thus the impact of some cache misses may be counted twice.

4.3 Impact of CAWET features (Cortex M4 and M7)

In this section, we analyze the effect of different features of CAWET on the Relative Percentage Error (RPE): context accounting, peek-on mechanism, loop management, and using Heptane's cache analysis. Our study involves a comparison of the impact of each feature, starting with context accounting (A), followed by the peek-on mechanism (B), loop

Table 7 Comparison of WCET predictions for CAWET and a Neural Network (NN) baseline for Cortex-M7 when accounting for the static cache analysis results.

Benchmark	Maximum observed execution time (Cycles)	NN estimations (Cycles)	NN RPE (%)	CAWET estimations (Cycles)	CAWET RPE (%)
bs	140	537	283.6	516	268.6
bsort	191406	840959	339.4	699961	265.7
countnegative	6956	33552	382.3	26997	288.1
crc	47476	184152	287.9	166025	249.7
expint	3592	14528	304.5	12764	255.3
fdct	4957	34861	603.3	20076	305.0
fir	4625	18088	291.1	16554	257.9
h264_dec	362349	1281479	253.7	1403042	287.2
insertsort	1760	6040	243.2	7105	303.7
jfdctint	4011	34044	748.8	19663	390.2
matrix1	301866	2021791	569.8	1249975	314.1
ns	21253	97870	360.5	76205	258.6
petrinet	1595	5813	264.5	6372	299.5
Avg. MAE	–	–	398.1	–	**289.6**

unrolling (C), and finally, applying cache analysis (D). The results in Table 8 show that incorporating the context (A) provides the most significant improvement to CAWET, while the effects of peeking (B) and loop enrolling (C) are less substantial. Additionally, we can see that adding the cache analysis (D) in Cortex M7 has a considerable impact on the predictions, with a significant increase in pessimism.

Table 8 RPE measures of CAWET predictions for Cortex-M4 and Cortex-M7 when adding different features of CAWET: context accounting (A), peek-on mechanism (B), loop unrolling (C), and cache analysis (D).

Feature(s) \Optimization	Cortex-M4 RPE (%)	Cortex-M7 RPE (%)
None	35.5	142.5
A	25.2	130.2
A+B	24.9	126.1
A+B+C	**23.8**	**102.2**
A+B+C+D	NA	288.0

4.4 Quality of WCET predictions for the Cortex A53

The objectives of these experiments are twofold: (i) evaluate the WCET predictions produced by CAWET for a more complex processor than the Cortex M7; (ii) be able to compare CAWET to WE-HML [2], the related work closest to CAWET, that targets this architecture. We re-use the very same experimental conditions as in WE-HML: software measurements of execution times, and execution on top of an operating system. The maximum measured BB execution time is used alongside its context to train CAWET. We have collected 1000 measurements for each studied benchmark and kept the maximum execution time observed as a reference value to calculate the RPE. On the thousand measurements collected, we have also applied the probabilistic WCET technique as described in [30], where we set the probability to 10^{-3} to provide another reference point than the MOET.

Table 9 Comparison of WCET predictions on Cortex A53 for: CAWET, a probabilistic WCET solution, WE-HML, CAWET (vanilla), and a modified CAWET to account for static cache analysis results.

Benchmark	MOET (Cycles)	pWCET10^{-3} RPE (%)	WE-HML RPE (%)	Vanilla CAWET RPE (%)	CAWET with cache analysis RPE (%)
bs	2568	43.8	177.1	97.0	122.8
bsort	358380	60.4	838.3	18.6	21.3
countnegative	29720	6.3	168.5	70.2	169.6
crc	66867	64.2	315.2	53.8	86.5
expint	6122	1.0	352.5	29.0	80.3
fdct	8877	1.2	195.0	25.5	52.2
fir	7646	-13.6	391.4	31.1	114.9
h264_dec	426327	120.4	590.0	76.5	88.4
insertsort	3042	75.8	297.6	29.6	40.2
jfdctint	8070	51.1	296.1	44.4	57.5
matrixl	21380	5.8	207.1	223.9	236.6
ns	22018	-0.3	731.1	108.6	119.5
petrinet	3920	30.7	1865.3	2.3	30.8
Avg. MAE	–	36.5	494.2	62.4	93

Table 9 shows the Maximum Observed Execution Times (MOET) and Relative Percentage Error (RPE) for all considered techniques: probabilistic WCET estimation, WE-HML, Vanilla CAWET, and CAWET modified with the results of static cache analysis. On all benchmarks but one (matrix1), CAWET is much less pessimistic than WE-HML (even for the modified CAWET). This is due to the significant pessimism introduced by WE-HML to account for caches (WE-HML evaluates cache effects by generating the worst possible cache pollution in loops regardless of the actual accesses performed in the loop).

Compared to the probabilistic technique, we observe that the pWCET is sometimes unsafe. This may come from rare outliers (due, for example, to the presence of an operating system) that are considered as WCET and that pWCET (smartly) ignores because they are sufficiently rare. It may also happen when pWCET is less pessimistic than CAWET. However, in general, pWCET techniques may miss the worst-case execution path in programs, whereas CAWET, a hybrid technique, will not.

5 Related works

The challenge of accurately estimating the WCET of programs has led to the development of various hybrid timing analysis techniques that are compared with CAWET below. These techniques can be broadly categorized into two types: those that use measurements to estimate the WCET of individual basic blocks and those that incorporate machine learning to learn the BB's timing patterns.

1. **Hybrid WCET estimation techniques using measurements**
   AbsInt [11, 20, 21] and Rapita [4, 32] have developed hybrid WCET estimation solutions, namely *Timeweaver* and *Rapitime*, which rely on hardware-assisted measurements (e.g., JTAG) and manual annotations (way/trace points and interest points, respectively) to measure the WCET on code snippets, and then estimate the WCET of the program with

their static tool. Kirner et al. propose in [22] to perform measurements on code segments larger than a basic block and propose techniques to enforce coverage of the measured segments. In contrast to these research works, CAWET does not use measurements to estimate the WCET of code snippets. Instead, it utilizes a timing model learned through Machine Learning (ML) techniques.

2. **Hybrid WCET estimation techniques using ML**
   Several methods for estimating Worst-Case Execution Time (WCET) using Machine Learning (ML) have been proposed [5, 18, 17, 2, 24, 25]. Bonenfant et al. [5] use worst-case event counts for training a neural network, that will be subsequently used to calculate the WCET of a program at an early stage. Similarly, the approaches proposed by Kumar [24, 25] estimate WCET using features extracted from the source code. These approaches disregard valuable information about the code flow and hide the compilation effects by operating at the source code or intermediate code level, which can bias the timing prediction. The research works presented in [17, 2], similarly to CAWET, propose to extract features from the binary code and to use ML techniques to predict the WCET of individual basic blocks. However, contrary to [18, 17, 2], CAWET takes a more fine-grained approach, considering the context surrounding each basic block, and the dependencies between instructions within it to better consider hardware components such as the pipeline. [2] accounts for data caches by simulating the worst possible data access pattern for basic blocks within loops, whereas CAWET relies on static analysis through the Heptane tool [16] to obtain less pessimistic estimations of data cache behavior. [26] propose a technique similar to linear regression to estimate the WCET from a set of end-to-end measurements. Unlike CAWET, this approach uses static features and is thus not able to accurately predict pipeline effects.
   All approaches described in this section oversimplify the code characterization, either by using high-level abstractions of the source code or by relying on static features of basic blocks at the binary code level. In contrast, *CAWET* operates on the flow of instructions using state-of-the-art ML techniques (Transformers-XL).
3. **Machine Learning for contention prediction and throughput prediction**
   Brando et al. [6] use neural networks to estimate the worst contention factor of programs using hardware event counters. Similarly, Courtaud et al. [7] introduce a profiling tool that produces high-resolution profiles of the memory behavior of applications. They train a regressor using microbenchmarks to finally calculate contention. Even though these two studies rely heavily on ML, they focus on contention prediction on multi-core targets and not on WCET prediction for single-cores like CAWET.
   Deep PM [33] and Ithemal [27] employ transformers and LSTMs, respectively, to predict the throughput of isolated basic blocks. However, CAWET takes a different approach by incorporating the execution context to predict WCETs. Similarly, CATREEN [1] uses stacked LSTMs to forecast the average execution time of basic blocks in a contextualized manner, but it differs from CAWET in its focus on average execution time rather than worst-case execution time.

6 Conclusion

In this paper, we presented CAWET: a hybrid approach that estimates the worst-case program timing for individual basic blocks in a program. Our approach uses static techniques to identify the longest execution path and an advanced machine learning architecture called transformer-XL to predict the worst-case execution time of each basic block. By considering

the execution context formed by previously executed basic blocks, CAWET is able to account for the micro-architecture of the processor pipeline without explicit modeling. The technique is demonstrated to be empirically reliable and less pessimistic than its competitors in experiments on the TacleBench benchmarks for different target processors. While there are still challenges to be addressed, such as the need for more accurate context for less pessimistic predictions, CAWET offers a promising solution for predicting worst-case execution times for Commercial off-the-shelf processors. In future work, the technique will be further explored for processors with out-of-order pipelines, such as Cortex A9 or A72.

References

1 Abderaouf N. Amalou, Elisa Fromont, and Isabelle Puaut. Catreen: Context-aware code timing estimation with stacked recurrent networks. In *34rd IEEE International Conference on Tools with Artificial Intelligence, (ICTAI)*. IEEE, 2022.

2 Abderaouf N Amalou, Isabelle Puaut, and Gilles Muller. We-hml: hybrid wcet estimation using machine learning for architectures with caches. In *2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 31–40. IEEE, 2021.

3 Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Otawa: An open toolbox for adaptive wcet analysis. In *Software Technologies for Embedded and Ubiquitous Systems: 8th IFIP WG 10.2 International Workshop, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Proceedings 8*, pages 35–46. Springer, 2010.

4 Adam Betts, Nicholas Merriam, and Guillem Bernat. Hybrid measurement-based wcet analysis at the source level using object-level traces. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2010.

5 Armelle Bonenfant, Denis Claraz, Marianne De Michiel, and Pascal Sotin. Early wcet prediction using machine learning. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, pages 5–1. OASICs, Dagstuhl Publishing, 2017.

6 Axel Brando, Isabel Serra, Enrico Mezzetti, Jaume Abella, and Francisco J Cazorla. Using quantile regression in neural networks for contention prediction in multicore processors. In *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

7 Cédric Courtaud, Julien Sopena, Gilles Muller, and Daniel Gracia Pérez. Improving prediction accuracy of memory interferences for multicore platforms. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 246–259. IEEE, 2019.

8 Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint*, 2019. `arXiv:1901.02860`.

9 Jean-François Deverge and Isabelle Puaut. Safe measurement-based wcet estimation. In *5th International Workshop on Worst-Case Execution Time Analysis (WCET'05)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2007.

10 Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT*, pages 4171–4186. Association for Computational Linguistics, 2019.

11 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)*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2016.

12 Christof Ebert, James Cain, Giuliano Antoniol, Steve Counsell, and Phillip Laplante. Cyclomatic complexity. *IEEE software*, 33(6):27–29, 2016.

13 Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. Taclebench: A benchmark collection to support worst-case execution time research. In *16th International Workshop on Worst-Case Execution Time Analysis*, 2016.

14 SEGGER Microcontroller GmbH. Ozone User Guide & Reference Manual. URL: `https://www.segger.com/`.

15 Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *4th IEEE international workshop on workload characterization*, 2001.

16 Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. The heptane static worst-case execution time estimation tool. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2017.

17 Thomas Huybrechts, Thomas Cassimon, Siegfried Mercelis, and Peter Hellinckx. Introduction of deep neural network in hybrid wcet analysis. In *Advances on P2P, Parallel, Grid, Cloud and Internet Computing: Proceedings of the 13th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC-2018)*, pages 415–425. Springer, 2019.

18 Thomas Huybrechts, Siegfried Mercelis, and Peter Hellinckx. A new hybrid approach on wcet analysis for real-time systems using machine learning. In *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2018.

19 Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In Vivek Sarkar, Barbara G. Ryder, and Mary Lou Soffa, editors, *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*, pages 171–185. ACM, 1994. `doi:10.1145/178243.178258`.

20 Daniel Kästner, Markus Pister, Simon Wegener, and Christian Ferdinand. Obtaining worst-case execution time bounds on modern microprocessors. In *Embedded World Conference*, 2018.

21 Daniel Kästner, Markus Pister, Simon Wegener, and Christian Ferdinand. Timeweaver: A tool for hybrid worst-case execution time analysis. In *19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019)*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2019.

22 Raimund Kirner, Ingomar Wenzel, Bernhard Rieder, and Peter Puschner. Using measurements as a complement to static worst-case execution time analysis. *Intelligent Systems at the Service of Mankind*, 2(8):20, 2005.

23 Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint*, 2018. `arXiv:1808.06226`.

24 Vikash Kumar. Deep neural network approach to estimate early worst-case execution time. In *2021 IEEE/AIAA 40th Digital Avionics Systems Conference (DASC)*, pages 1–8. IEEE, 2021.

25 Vikash Kumar. Estimation of an early wcet using different machine learning approaches. In *Advances on P2P, Parallel, Grid, Cloud and Internet Computing: Proceedings of the 17th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC-2022)*, pages 297–307. Springer, 2022.

26 Björn Lisper and Marcelo Santos. Model identification for wcet analysis. In *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 55–64. IEEE, 2009.

27 Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on machine learning*, pages 4505–4515. PMLR, 2019.

28 Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint*, 2021. `arXiv:2105.12655`.

29 Federico Reghenzani, Giuseppe Massari, William Fornaciari, and Andrea Galimberti. Probabilistic-wcet reliability: On the experimental validation of evt hypotheses. In *Proceedings of the International Conference on Omni-Layer Intelligent Systems*, pages 229–234, 2019.

30 Federico Reghenzani, Luca Santinelli, and William Fornaciari. Dealing with uncertainty in pwcet estimations. *ACM Transactions on Embedded Computing Systems (TECS)*, 19(5):1–23, 2020.

31 Segger. J-Trace PRO – The Leading Trace Solution. URL: `https://www.segger.com/products/debug-probes/j-trace/`.

32 Hardik Shah, Andrew Coombes, Andreas Raabe, Kai Huang, and Alois Knoll. Measurement based wcet analysis for multi-core architectures. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, pages 257–266, 2014.

33 Jun S Shim, Bogyeong Han, Yeseong Kim, and Jihong Kim. Deeppm: transformer-based power and performance prediction for energy-aware software. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1491–1496. IEEE, 2022.

34 Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

35 Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

36 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

37 Tomofumi Yuki. Understanding polybench/c 3.2 kernels. In *International workshop on polyhedral compilation techniques (IMPACT)*, pages 1–5, 2014.

A Appendix

Table 10 Hyperparameters used during transformer XL pretraining and finetuning.

Hyperparameters	Pretraining phase	Finetuning phase
Number of layer	4	4
Number of attention heads	3	3
Dimension of head	22	22
Dimension of inner head	128	128
Dimension of hidden layers	512	512
Optimizer	“adam”	“adam”
Target length	512	512
Memory length	1024	1024
Linear layer (fine tuning)	–	{512, 256, 128, 1}
Learning rate	0.00025	0.0001

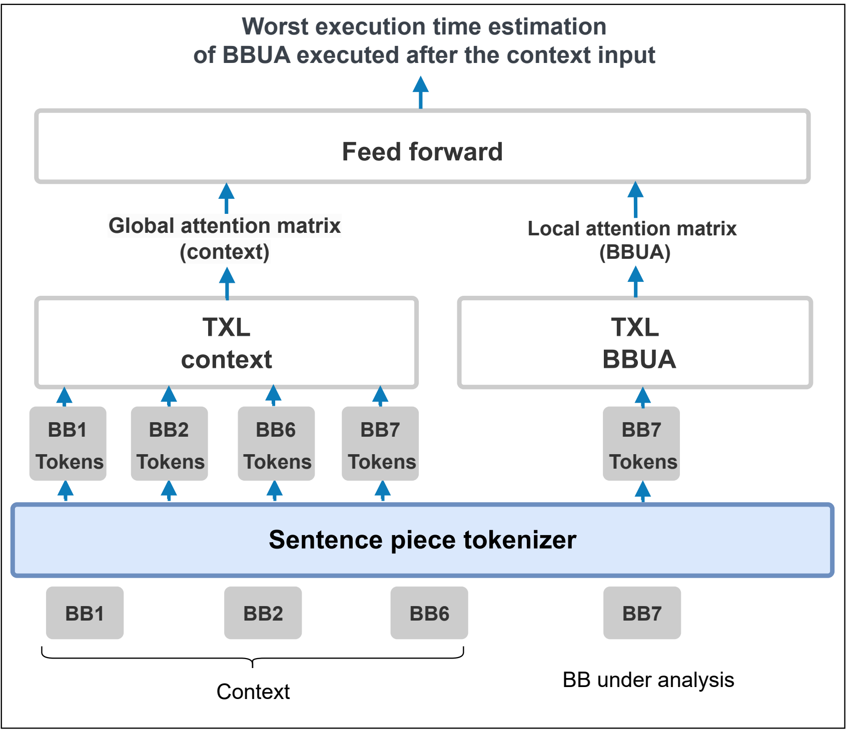


Figure 4 CAWET Transformer XL system architecture.

Precise Scheduling of DAG Tasks with Dynamic Power Management

Ashikahmed Bhuiyan
Department of Computer Science, West Chester University, PA, USA

Mohammad Pivezhandi
Department of Computer Science, Wayne State University, Detroit, MI, USA

Zhishan Guo
Department of Computer Science, North Carolina State University, Raleigh, NC, USA

Jing Li
Department of Computer Science, New Jersey Institute of Technology, Newark, NJ, USA

Venkata Prashant Modekurthy
Department of Computer Science, University of Nevada, Las Vegas, NV, USA

Abusayeed Saifullah
Department of Computer Science, Wayne State University, Detroit, MI,USA

Abstract

The rigid timing requirement of real-time applications biases the analysis to focus on the worst-case performances. Such a focus cannot provide enough information to optimize the system's typical resource and energy consumption. In this work, we study the real-time scheduling of parallel tasks on a multi-speed heterogeneous platform while minimizing their typical-case CPU energy consumption. Dynamic power management (DPM) policy is integrated to determine the minimum number of cores required for each task while guaranteeing worst-case execution requirements (under all circumstances). A Hungarian Algorithm-based task partitioning technique is proposed for clustered multi-core platforms, where all cores within the same cluster run at the same speed at any time, while different clusters may run at different speeds. To our knowledge, this is the first work aiming to minimize typical-case CPU energy consumption (while ensuring the worst-case timing correctness for all tasks under any execution condition) through DPM for parallel tasks in a clustered platform. We demonstrate the effectiveness of the proposed approach with existing power management techniques using experimental results and simulations. The experimental results conducted on the Intel Xeon 2680 v3 12-core platform show around 7%-30% additional energy savings.

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

Keywords and phrases Parallel task, mixed-criticality scheduling, energy minimization, dynamic power management, cluster-based platform

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.8

Funding *Ashikahmed Bhuiyan*: Provost's Research Grant, West Cheser University
Zhishan Guo: NSF CCF-2028481, Startup Grant at NC State University
Jing Li: NSF CNS-1948457
Venkata Prashant Modekurthy: UNLV Troesh Center, CNS-2211640
Abusayeed Saifullah: CNS-2301757, CAREER-2306486, CNS-2306745

© Ashikahmed Bhuiyan, Mohammad Pivezhandi, Zhishan Guo, Jing Li, Venkata Prashant Modekurthy, and Abusayeed Saifullah;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

Multi-core processors are increasingly appearing as an enabling platform for embedded systems (e.g., mobile phones, tablets, drones, computerized numerical controls, etc.). The parallel task model can exploit the multi-core platform's capability as they support *intra-task parallelism*, where a task can execute on multiple cores simultaneously. Many computation-intensive systems (e.g., self-driving cars) that demand stringent timing requirements often evolve in the form of parallel tasks. Many recent studies on real-time scheduling and analysis have focused on the directed acyclic graph (DAG) model of parallel tasks [8, 9, 16, 37, 38, 53, 55]. The DAG model is a general workload model for representing intra-task parallelism, where nodes represent threads of execution and edges represent their dependencies. Several real-world applications use the DAG model [29].

Energy efficiency is essential for embedded systems, as they rely on a time-limited energy sources (i.e., batteries, energy harvesting devices). Modern generation processors minimize power consumption through *dynamic voltage and frequency scaling* (DVFS) that adjusts the voltage and frequency at runtime. To date, some works considered the energy-aware real-time scheduling of parallel tasks [10, 11, 54]. These works adopted the federated scheduling [38] or task decomposition framework [55] with the DVFS policy for minimizing system energy consumption in the per-core or per-node (of a DAG task) speed modulation settings. Such speed tuning is inefficient as it increases the hardware cost [31]. Also, there is an ongoing trend of considering the cluster-based platform (e.g., big.LITTLE [48]), which groups processors into multiple islands, each execute at the same speed. Such a cluster-based platform balances energy efficiency and cost [43]. To date, few efforts have been made to study the energy-aware real-time scheduling of parallel tasks in a clustered platform [11, 25].

All these works assumed that hard real-time constraints must be satisfied to guarantee the system's correctness. For a hard real-time task, missing a deadline is considered a system failure and may result in catastrophic consequences. Hence, most schedulability analyses considered that a task could execute up to its worst-case execution time (WCET). WCET-based schedulability test is often very pessimistic [42], and the task execution pattern varies significantly across different job releases, rarely executing up to its WCET [58]. Thus, designing a system that relies primarily on WCET may lead to resource over-provisioning in typical cases [46]. Moreover, modern embedded systems pose strict energy constraints and demand leveraging richer system models to optimize energy consumption under typical-case instead of worst-case. To address this issue, this paper requires system designers to provide at least two execution time estimates for a task: a WCET and a typical-case execution budget (no more than the WCET). The first will give worst-case guarantees, while the latter is used for energy minimization. Although a dual-execution-estimation setting may appear similar to the mixed-criticality (MC) framework [59], this work focuses on a different problem. In an MC setup, a common platform integrates different tasks with varying levels of criticality. A criticality level is assigned to each task with multiple execution time thresholds. Under such settings, existing works studied energy minimizing [12, 13, 34, 47, 60]. However, these works assume that all the tasks execute up to their WCET at the respective criticality levels. Meanwhile, our approach proposes optimizing energy consumption under the typical-case execution time instead of WCET. Our approach also ensures that all tasks must receive full-service guarantees under all circumstances.

The energy-aware scheduling of real-time tasks is challenging due to the complicated dependencies among frequency, energy consumption, and execution time [24]. Existing works focused mostly on the DVFS policy [10, 11, 25, 26, 54] with a significant limitation: it is

not effective in reducing static power consumption, which may elevate to 50% or more of the overall power consumption [33]. Besides, existing energy minimization approaches are applied to WCET and thus will only lead to better power/energy behaviors in the worst case (rare event) instead of the typical/average scenarios. Given both *typical* and *worst-case* execution time estimates, we propose an energy-aware technique that minimizes the *typical* energy consumption while guaranteeing (worst-case) timing correctness for all tasks.

Challenges. Handling the dual execution estimation is challenging for the following reasons. *First*, schedulers are unaware of each task's exact behavior before run-time. Such non-clairvoyance of execution length typically leads to NP-Hard problems. *Second*, all tasks receive full-service assurances under any circumstances. *Third*, some recent works have studied the energy-aware scheduling of the MC task model [12, 13, 34, 47], but thy did not consider intra-task parallelism.

Motivated by these facts, we study the real-time scheduling of DAG tasks in a clustered platform to minimize their CPU energy consumption, one of the significant contributors to the overall system power consumption. The scheduling problem aims to achieve both worst-case real-time guarantees and typical-case energy efficiency. In a clustered platform, all cores in the same cluster execute at the same speed. However, different clusters can operate at different speeds [48]. We adopt the DPM policy to reduce static power consumption. DPM policy reduces static power consumption by utilizing idle intervals. If the idle interval is at least equal to a certain threshold (known as the *break-even time* [18]), the processor is switched to a low-power sleep mode, thus reducing its static power consumption. Our approach finds the minimum but a sufficient number of low-speed cores for each task, leaving many high-speed cores idle for a long duration. If the low-speed cores are insufficient to schedule all the tasks, we assign additional high-speed cores. Therefore, the proposed approach can lead to high energy savings resulting from the power-down of CPU components such as cores, caches, and translation look-aside buffer.

The key objective of the proposed method is to conserve energy during the actual execution of a DAG task when its nodes do not execute until their WCET. Several factors, including pipelines, branch predictors, and caches, impact the WCET estimation of the nodes in the DAG task, thereby impacting the task's makespan estimation. In addition, hardware features often introduce pessimism to the WCET estimation, implying the difference between WCET estimation and actual execution time increases considerably. In such cases, the proposed approach will significantly increase energy savings by allocating resources only when needed. Specifically, we make the following key contributions:

- We utilize DPM to propose an energy-aware federated scheduling strategy of parallel DAG tasks on dual-speed platforms. Given both typical and worst-case execution time estimations, our energy-aware approach determines the required (minimum) number of low-speed processors to minimize the typical CPU energy consumption while guaranteeing worst-case timing correctness for all tasks.
- Under multi-speed cluster-based settings, we propose an energy-efficient task-cluster partitioning technique without violating the schedulability guarantees.
- We perform the experimental study under randomly generated task sets. We report the *schedulability ratio* of our approach, and demonstrate a minimum of 29.23% less power consumption against state-of-the-art approach [39].
- We present onboard experiments conducted on Intel Xeon 2680 v3 multi-core (12-core) platform and report up to 30% energy savings compared to the existing approach.

2 Related Work

There have been works studying the energy-efficient real-time scheduling of sequential tasks in both uni- and multi-processor platforms (few to mention [15,17,19,35,47,50,51]; refer to [4] for a comprehensive survey). However, all these works considered the sequential task model. In contrast, a parallel task can make use of multiple cores simultaneously and complete the same amount of work in a shorter time via exploiting the internal parallelism. Hence, the scheduling strategy and analysis of parallel task is significantly different from the sequential task. The state-of-the-art parallel real-time scheduling primarily emphasizes scheduling analysis and does not account for energy awareness [1,5,8,9,23,55,56].

To date, a few works studied the energy-aware scheduling of parallel tasks. Li et al. [40] studied a non-recurrent task model with a fixed number of parallel threads. Paolillo et al. [52] studied the energy-aware scheduling of the gang task model. Zhu et al. [61,62] proposed a slack stealing based scheduling approach considering the inter-dependent sequential tasks. Energy-aware scheduling of DAG tasks was proposed by Bhuiyan et al. [10] and Guo et al. [26]. Both of them have considered a simplified model (i.e., the number of cores cannot be pre-fixed). They have considered table-driven scheduling. Hence, the entire schedule until the hyper-period needed to be created in advance. Some recent efforts have been made to study the energy-efficient scheduling of parallel (and sequential) tasks in a clustered platform [11,20,25,41,45]. However, our work's focus differs from existing ones. We study the energy-efficient scheduling of parallel tasks on a clustered platform while minimizing their typical-case CPU energy consumption while guaranteeing worst-case execution requirements.

Meanwhile, extensive research has investigated the real-time scheduling of the MC task model considering both the sequential and parallel task model (e.g., [1,5,7,12–14,22,27,39]). Regarding the task model, the work in [1] is most near to us. However, our paper's contributions differ significantly from [1] regarding problem statement, challenges, solution techniques, and evaluation. For MC DAG tasks, [1] did not consider energy-aware scheduling. In contrast, our paper adopts the DPM policy to minimize power consumption. Incorporating the DPM policy into the existing analysis is not trivial because (i) DPM policy utilizes the processor's idle slot that is unknown apriori; (ii) We have considered the clustered platform. Hence, we cannot turn off some processors in a cluster while others are running.

3 System Model and Background Concepts

In this work, we consider a set of sporadic parallel DAG tasks denoted by $\tau = \{\tau_1, \tau_2, \cdots, \tau_n\}$. Each DAG contains a set of nodes, where a node represents some sequential computation. The precedence constraint between two nodes is represented by a directed edge. A node can only execute if all of its predecessors have finished execution. For each task $\tau_i \in \tau$, we consider the following two parameters: (i) *total work*, which is the sum of the number of clock cycles (i.e., total computation work) performed by all nodes in τ_i; and (2) the *critical-path length*, which is defined as the number of clock cycles of the longest directed path of the DAG (i.e., the path with the largest computation work).

Each task $\tau_i \in \tau$ is characterized by a 5-tuple $(C_i^N, C_i^O, L_i^N, L_i^O, T_i)$. Here, C_i^N denotes the typical-case execution time of the task (referred to as LO-criticality execution time[1]), while C_i^O denotes the overload execution time (referred to as HI-criticality execution time).

[1] Although the terms LO- and HI-criticality may lead to the assumption that we are considering tasks with different criticality levels [59]. In this work, the term criticality is used to distinguish whether a job exceeds its typical workload and whether a job's execution modes are still exhibiting the typical (or overload) workload estimates.

Similarly, $L_i^N(L_i^O)$ denotes the typical (overload) critical-path length estimates, referred to as LO(HI)-criticality critical-path length estimates. Note that the typical workload estimates (i.e., C_i^N and L_i^N) are obtained using a less pessimistic yet practical tool and are expected to occur during regular operations. In contrast, a more pessimistic tool (by considering all possible scenarios, including the worst-case ones) is used to obtain C_i^O and L_i^O. Thus, the overload workload estimates may exceed the typical ones in several orders of magnitudes. However, executing up to its overload workload is rare for a task. We assume that under any condition, a task's total work and critical-path length will never exceed C_i^O and L_i^O, respectively. The period of task τ_i is denoted by T_i. We assume that each task τ_i has an *implicit* deadline, i.e., the relative deadline D_i is equal to T_i. For τ_i to be scheduled during an overload scenario, the condition $T_i \geq L_i^O$ must be satisfied, where L_i^O denotes the minimum time required to complete task τ_i even when an infinite number of cores are available. The typical (overload) utilization of task τ_i is defined as the ratio of its typical (overload) total work and its period. Let, $u_i^N = \frac{C_i^N}{T_i}$ and $u_i^O = \frac{C_i^O}{T_i}$ denote the typical and overload utilization, respectively.

▶ **Example 1.** Consider two DAG tasks τ_1 and τ_2, where $\tau_1 = (12, 24, 4.08, 8.16, 21.6)$ and $\tau_2 = (12, 36, 3.36, 12.24, 33.12)$. The typical total work of τ_1 is 12 and the overload total work is 24. The typical and overload critical-path lengths of τ_1 are 4.08 and 8.16, respectively. Because τ_1 has a period of 21.6, its typical utilization $u_1^N = \frac{12}{21.6} = 0.56$ and its overload utilization $u_1^O = \frac{24}{21.6} = 1.11$. Similarly, the typical utilization u_2^N and overload utilization u_2^O of task u_2^N are $u_2^N = \frac{12}{33.12} = 0.36$ and $u_2^O = \frac{36}{33.12} = 1.087$, respectively.

System behavior. In this work, we consider a task's two different execution (typical and overload) requirement. At runtime, the scheduler does not know the exact behavior of each job of a task, e.g., the exact workload or critical-path length of the job. Thus, it is expected that the job of a task starts execution using its typical execution budget. If a job's total work or the critical-path length exceed this task's typical total work or critical-path length, this job execute according to its HI-criticality execution budget and critical-path length estimate. Note that the precise timing of when a task may require the use of its overload execution budget is unpredictable. However, once a job (of any task) finishes executing its overload execution budget, the next job (of the same task) starts running according to its typical execution requirement.

Power/Energy model. In this work, we consider the following power model to represent the CPU power consumption by a processor [10, 26, 34, 47, 50, 51]. Let s denotes the main frequency (speed) of a processor, and the power consumption $P(s)$ can be expressed as:

$$P(s) = P_{sta} + P_{dyn}(s) = \beta + \alpha s^\gamma \tag{1}$$

Here, P_{sta} and $P_{dyn}(s)$ denote the static/leakage consumption and dynamic power consumption, respectively. If a processor is not entirely turned off, P_{sta} (represented as β) is introduced in the system due to leakage current, and the frequency-dependent switching activities introduce $P_{dyn}(s)$ (represented as αs^γ). For modeling the dynamic power consumption $P_{dyn}(s)$, α depends on the effective switching capacitance and $\alpha > 0$ [50]; γ is a fixed parameter determined by the hardware and it ranges between [2,3]. Pagani et al. [50] have shown that this model is highly realistic by comparing the power consumption (estimated by this model) with the actual power consumption results from [32]. Table 1 illustrates the

Table 1 Comparison of the CPU power consumption P_{equ} considering the power model in Eq (1) and experimental data P_{exp} from [32]. The *error rate* is defined as $100 \times (P_{\text{equ}} - P_{\text{exp}})/P_{\text{exp}}$.

Frequency	P_{exp}	P_{equ}	Error rate
0.24 GHz	0.54 W	0.52 W	-3.7%
0.46 GHz	0.70 W	0.67 W	-4.28%
0.68 GHz	1.04 W	1.05 W	0.96%
0.84 GHz	1.5 W	1.54 W	2.67%
0.92 GHz	1.96 W	1.88 W	-4.07%
1.02 GHz	2.30 W	2.35 W	2.17%

detailed comparison. Given a fixed amount of workload C executing on a speed-s processor, we can calculate the total energy consumption $E(C, s)$ as the integral of power throughout C/s, where $E(C, s) = (\beta + \alpha s^{\gamma}) \times \frac{C}{s} = \frac{\beta C}{s} + \alpha C s^{\gamma-1}$.

We aim to reduce static energy consumption by employing the DPM approach, which utilizes the processor's idle time. If the idle interval in a processor is greater or equal to a certain threshold (i.e., break-even time [18]), the processor enters a low-power sleep mode. In this work, we try to allocate low-speed cores to all tasks, leaving the high-speed cores idle. The high-speed cores are used when some jobs enter HI-criticality mode due to exceeding their typical workload estimates. In this case, additional resources (i.e., some high-speed cores) are needed to complete the overload workload. Because tasks rarely exceed their typical workload estimates, the high-speed cores typically idle for a long duration. Thus, these idle cores can enter the low-power sleep mode, reducing static energy consumption.

Platform model. We are examining a homogeneous multi-core architecture called the Intel Xeon 2680 V3, where each core can have a designated clock frequency and a set of customized fine-tuned cores. In contrast to the initially clustered platforms such as the Odroid XU4 ARM's big.LITTLE architecture [48], which forces a fixed number of cores in each cluster to be synchronized at the same speed. This feature of the Xeon Processor facilitates the identification of the appropriate number of cores in each cluster. It gives a final general solution where clusters are initialized with a constant speed that can be fixed to a corresponding speed optimized through DAG Task features. In this case study, we halve the platforms cores referring to ξ clusters (each with n cores), and all cores in the same cluster execute at the same speed. However, different clusters can operate at different speeds. The maximum speed is s^L (s^H) for the LO(HI)speed clusters, where $s^L \leq s^H$. In a clustered architecture, we assume that any core can be put to sleep (i.e., processor clock turned off) and only an entire cluster can be put to deep sleep (processor and L2 clock turned off) [3].

Virtual Deadline. The concept of the virtual deadline was first proposed in [6], considering MC scheduler. High-criticality task is assigned a virtual deadline which is less than the actual deadline (and thus a higher priority). This ensures the HI-criticality jobs get sufficient slack for their overload workload to complete after a mode switch.

Federated Scheduling Algorithm. In real-time systems, multiprocessor algorithms are implemented considering either the *global* or *partitioned* approach. In the partitioned approach, a task to processor mapping is performed before run-time for each task. During run-time, no job migration is allowed, and all the jobs generated by a task execute only on its mapped processor. Considering the parallel DAG task models, Li et al. [37] proposed

Table 2 Major notations used throughout the paper.

	Symbol	Description
Workload	τ_i	The ith task
	$C_i^N(C_i^O)$	typical (overload) execution budget of task τ_i
	$L_i^N(L_i^O)$	typical (overload) critical-path length estimates of task τ_i
	$u_i^N(u_i^O)$	typical (overload) utilization of τ_i
	$T_i(D_i)$	period (relative deadline) of task τ_i
	D_i'	virtual deadline of task τ_i
Platform	$\mathcal{K}_i$	i^{th} cluster
	M_i	number of sub-cluster in i^{th} cluster
	$\mathcal{K}_m^n$	n^{th} sub-cluster inside m^{th} cluster
	$s^L(s^H)$	execution speed of the low (high) speed cores
	S_k	Speed of kth cluster
	$m_i^L(m_i^H)$	number of low(high) speed cores allocated to task τ_i
	$E_i^{\mathcal{K}_m^n}$	energy consumption by a sub-cluster $\mathcal{K}_m^n$ when executing a task τ_i

the *federated scheduling* approach, which is considered a reasonable extension of partitioned scheduling for parallel tasks. A federated scheduling algorithm classifies a task as a heavy task if its utilization is greater than or equal to 1; otherwise, the task is classified as a light task. Each heavy task receives a set of cores dedicated to this task. All the remaining cores (i.e., the cores left after each heavy task receives its portion) are given to all light tasks. A multiprocessor scheduling algorithm (e.g., partitioned earliest deadline first [44] or rate monotonic schedulers [2]) is used to schedule all these light tasks sequentially. In contrast to heavy tasks, light tasks can share cores.

Given a task set τ, a federated scheduler works as follows. The task set τ is divided into disjoint sets, i.e., τ_{heavy} and τ_{light}. Here, τ_{heavy} contains all the heavy tasks (utilization is at least 1), and τ_{light} contains all the light tasks (utilization is less than 1). A heavy task $\tau_i \in \tau_{heavy}$, receives m_i cores. Here, $m_i = \left\lceil \frac{C_i - L_i}{T_i - L_i} \right\rceil$ (refer to [37]), where, C_i is the WCET of τ_i, L_i is the critical path length, and $D_i(= T_i)$ the deadline. Then, the remaining m_{light} cores, where $m_{light} = m - \sum_{\tau_i \in \tau_{heavy}} m_i$, can be used by all the light tasks. The light tasks are forced to execute sequentially and scheduled by a multiprocessor scheduling algorithm. After a valid task to core allocation, runtime scheduling is performed as follows:

- For a high-utilization task $\tau_i \in \tau_{heavy}$, any greedy or work-conserving parallel scheduler is used to schedule τ_i on m_i cores.
- Any multiprocessor scheduling algorithm is used to schedule all light tasks on the remaining m_{light} cores if the algorithm's schedulability test is passed.

4 Energy-aware Federated Scheduling for the Dual-Speed Platform

This section discusses our energy-aware federated scheduling strategy for platforms with dual-speed cores. Considering that the platform supports cores with two speeds, i.e., low- and high-speeds, we aim to minimize the CPU energy consumption under typical scenarios while guaranteeing that all tasks receive enough execution budget even under overload scenarios. Our approach relies on the DPM approach to reduce energy consumption and tries to allocate only the low-speed cores to all tasks, leaving the high-speed cores idle in most cases, leading to reduced dynamic power consumption. Besides, all these high-speed idle cores can enter the low-power sleep mode, which further minimizes the static energy consumption.

Towards this goal, Subsection 4.1 determines the required minimum number of low-speed cores m_i^L for each task $\tau_i \in \tau$ to complete the overload workload. If there are not enough low-speed cores to serve all the tasks under overload scenarios, we compensate for this shortage by assigning additional high-speed cores to some of the tasks when their jobs exceed typical workloads; on the other hand, the numbers of low-speed cores allocated to these tasks are reduced under typical scenarios. Because tasks rarely exhibit overload behavior, the high-speed cores typically are not used, which is beneficial for reducing energy consumption. Subsection 4.2 presents a greedy approach (Algorithm 1) that checks if there are enough low-speed cores to serve all the tasks under overload scenarios. If not, Algorithm 1 allocates additional high-speed cores under overload scenarios to some tasks that provide the maximum relative core saving ratio under typical scenarios. Note that, in this approach, we assume that the processor speeds are given.

4.1 Determining the Number of Cores for Each Task

In this subsection, we determine the number of cores required for each task $\tau_i \in \tau$ to meet its deadline. Our analysis considers the following assumption.

Assumption. This section assumes that the platform consists of two types of cores, i.e., low-speed cores and high-speed cores. The execution speed of any task τ_i on the low-speed and high-speed cores are respectively denoted by s^L and s^H.[2] The execution speed of a core denotes the (minimum) amount of computation that can be completed per time unit. This section assumes that the total workload and the processor speed have a linear relationship [13]. Hence, for any task τ_i, we can translate the total workload to the execution time on s-speed cores as C_i^{χ}/s, where $\chi = \{N, O\}$. Similarly, for any task τ_i, we can translate the critical path to the execution time on s-speed cores as L_i^{χ}/s. Now, we classify a task $\tau_i \in \tau$ into the following three categories:

Category 0: $C_i^O \leq s^L D_i$. Task τ_i in this category with WCET C_i^O/s^L and deadline D_i is a light (or low utilization) task even on low-speed cores. We enforce these tasks to execute sequentially and use any traditional multiprocessor scheduling (e.g., partitioned EDF) approach to schedule them on the low-speed cores.

Category 1: $C_i^O > s^L D_i$ and $C_i^N/s^L > C_i^O/s^H - L_i^O/s^H$. For tasks in this category, we allocate m_i^L low-speed cores for both LO- and HI–criticality modes. According to Lemma 1 in [1], any task τ_i (where τ_i belongs to Category 1) has a maximum makespan of $(C_i^O - L_i^O)/m + L_i^O$, where m is the number of unit-speed cores allocated to τ_i. Therefore, to meet deadline D_i on cores with speed s^L, the lower bound on m_i^L can be calculated as follows:

$$\frac{\frac{C_i^O}{s^L} - \frac{L_i^O}{s^L}}{m_i^L} + \frac{L_i^O}{s^L} \leq D_i \Longrightarrow \frac{C_i^O - L_i^O}{s^L \times m_i^L} \leq \frac{s^L \times D_i - L_i^O}{s^L}$$
$$\Longrightarrow \frac{C_i^O - L_i^O}{s^L \times D_i - L_i^O} \leq m_i^L$$

Hence, we allocate m_i^L low-speed cores to τ_i belonging to Category 1, where $m_i^L = \left\lceil \frac{C_i^O - L_i^O}{s^L \times D_i - L_i^O} \right\rceil$. Here, we assume that $m_i^L \geq 0$, meaning that the critical-path length of each DAG on the low-speed cores is shorter than its corresponding deadline, and it is

[2] This section assumes that all the low (or high) speed cores execute at the same speed, which is independent of the executing tasks. Hence, for any task $\tau_i, \tau_j \in \tau$, their execution speeds are the same, if they are allocated in the same cluster. Such a restriction appears commonly in existing systems [48], where all processors within the same cluster/island execute at the same speed during run time.

feasible to assign each task τ_i to a low-speed core. One could relax this constraint and include tasks that are assigned to high-speed cores; however, the energy-saving techniques proposed in this paper are no longer applicable in this situation. In this work, we focus on minimizing energy consumption while ensuring that all tasks meet their deadlines using low-speed cores. Hence, handling any infeasible task to schedule on low-speed cores falls beyond the scope of this paper.

Category 2: $C_i^O/(s^L \times D_i) > 1$ and $C_i^N/s^L \le C_i^O/s^H - L_i^O/s^H$. For any task τ_i that belongs to Category 2, we try to allocate $\left\lceil \frac{C_i^O - L_i^O}{s^L \times D_i - L_i^O} \right\rceil$ low-speed cores to it in both modes, if there is sufficient number of low-speed cores available. Otherwise, we allocate fewer low-speed cores and set a virtual deadline D_i' for the LO-criticality mode. The virtual deadline D_i' is set as $D_i' = \frac{C_i^N}{m_i^L \times s^L}$, so that task τ_i can finish its nominal workload by D_i' if no core idles. In summary, we set m_i^L as follows:

$$m_i^L = \begin{cases} \left\lceil \dfrac{C_i^O - L_i^O}{s^L \times D_i - L_i^O} \right\rceil ; \text{If available LO-speed cores} \\ \text{are sufficient.} \\ \left\lceil \dfrac{C_i^N/s^L}{(D_i - \frac{L_i^O}{s^H} - \frac{C_i^O - C_i^N - L_i^O}{m_i^H \times s^H})} \right\rceil ; \qquad \text{Otherwise.} \end{cases} \tag{2}$$

Here, m_i^H and s^H denote the number of high-speed cores (allocated to task τ_i) and their speed, respectively, for the case where not enough low-speed cores available. See Theorem 1 for the derivation of m_i^L.

For task τ_i with fewer low-speed cores available and assigned to it, we assign it m_i^H high-speed cores and consider the actual deadline D_i during the HI-criticality mode. In order to meet the deadline, the computing power in the high-criticality mode must be equal to or greater than the computing power in the low-criticality mode, meaning that, meaning that, so $m_i^H \times s^H \ge m_i^L \times s^L$. Additionally, task τ_i needs to finish the remaining work and critical-path length within $D_i - D_i'$ time units. The worst case scenario happens when there is no progress on the critical-path length L_i^O, which leaves a remaining work of $C_i^O - m_i^L \times D_i' \times s^L$. This is the worst-case scenario because more processor times are idling due to the critical-path length given the relation between m_i^H and m_i^L, where $m_i^H \ge m_i^L \times s^L/s^H$. Therefore, to meet the deadline, m_i^H can be calculated as follows:

$$\begin{aligned} m_i^H &= \max\left\{ \frac{m_i^L \times s^L}{s^H}, \left\lceil \frac{\frac{(C_i^O - m_i^L \times D_i' \times s^L - L_i^O)}{s^H}}{(D_i - D_i') - \frac{L_i^O}{s^H}} \right\rceil \right\} \\ &= \max\left\{ \frac{m_i^L \times s^L}{s^H}, \left\lceil \frac{C_i^O - m_i^L \times D_i' \times s^L - L_i^O}{(D_i - D_i') \times s^H - L_i^O} \right\rceil \right\} \end{aligned}$$

Refer to [1] for the formal proof.

▶ **Theorem 1.** *If there is not enough low-speed cores for task τ_i, m_i^L can be reduced to $m_i^L = \left\lceil \frac{C_i^N/s^L}{(D_i - \frac{L_i^O}{s^H} - \frac{C_i^O - C_i^N - L_i^O}{m_i^H \times s^H})} \right\rceil$ by assigning m_i^H high-speed cores to τ_i during the HI-criticality mode, where $m_i^H \ge \frac{C_i^O - m_i^L \times D_i' \times s^L - L_i^O}{(D_i - D_i') \times s^H - L_i^O}$.*

Proof. For any task τ_i, when there are not enough low-speed cores, we assign additional high-speed cores to τ_i to update m_i^L. Since $D_i' = \frac{C_i^N}{m_i^L \times s^L}$, we have

$$
\begin{aligned}
m_i^H &\geq \frac{C_i^O - m_i^L \times D_i' \times s^L - L_i^O}{(D_i - D_i') \times s^H - L_i^O} \\
\Longrightarrow &(D_i - D_i') \times s^H - L_i^O \geq \frac{C_i^O - m_i^L \times D_i' \times s^L - L_i^O}{m_i^H} \\
\Longrightarrow &D_i \times s^H - \frac{C_i^N \times s^H}{m_i^L \times s^L} - L_i^O \geq \frac{C_i^O - C_i^N - L_i^O}{m_i^H} \\
\Longrightarrow &m_i^L \geq \frac{C_i^N \times s^H}{(D_i \times s^H - L_i^O - \frac{C_i^O - C_i^N - L_i^O}{m_i^H}) \times s^L} = \frac{C_i^N / s^L}{(D_i - \frac{L_i^O}{s^H} - \frac{C_i^O - C_i^N - L_i^O}{m_i^H \times s^H})}
\end{aligned}
$$

which holds since $m_i^L = \left\lceil \frac{C_i^N / s^L}{(D_i - \frac{L_i^O}{s^H} - \frac{C_i^O - C_i^N - L_i^O}{m_i^H \times s^H})} \right\rceil$. ◀

4.2 When Low-Speed Cores Are Not Sufficient

Section 4.1 determines the number of low-speed cores (m_i^L) required for each task, $\tau_i \in \tau$, to meet its deadline. If there are a finite number of low-speed cores, some tasks may not receive enough low-speed cores. In this section, we present Algorithm 1 that first tries to allocate the desired number (m_i^L) of low-speed cores to each task τ_i. If there are not enough low-speed cores, Algorithm 1 assigns additional high-speed cores to compensate for this shortage. In this approach, we assume that the processor speeds are given.

Algorithm 1 starts by checking whether a task τ_i is a Category-0 task (i.e., $C_i^O/(s^L \times D_i) \leq 1$). If yes, we use traditional multiprocessor scheduling (e.g., partitioned EDF) for sequential tasks with WCET C_i^O/s^L and deadline D_i on the minimum number of low-speed cores (Lines 3-4). If a task τ_i belongs to Category-1, i.e., $C_i^N/s^L > C_i^O/s^H - L_i^O/s^H$, we allocate $m_i^L = \left\lceil \frac{C_i^O - L_i^O}{D_i \times s^L - L_i^O} \right\rceil$ low-speed cores and meet its deadline D_i (Lines 5-6). Let, $\tilde{\tau}$ denotes the set of unscheduled tasks, which is updated continuously (Line 4 and Line 6). After allocating the required number of low-speed cores to all Category-0 and Category-1 tasks, we calculate the remaining low-speed cores, $\tilde{m}^L$ (Line 9).

For any task $\tau_i \in \tilde{\tau}$, we set m_i^L as $\left\lceil \frac{C_i^O - L_i^O}{D_i \times s^L - L_i^O} \right\rceil$ and m_i^H (i.e., number of high-speed cores allocated to τ_i) to 0 (Lines 9-11). If all the tasks in $\tilde{\tau}$ receive the required number of low-speed cores, then the algorithm terminates (Lines 12-13). Else, we update the required number of high-speed cores allocated to task τ_i and denote it as m_i^H (Line 16), and also update m_i^L (Line 17). We calculate the relative core saving ratio (Line 15) and pick the task (say τ_i) that has the highest relative core saving ratio (Line 16). If there are sufficient cores for τ_i, we remove τ_i from $\tilde{\tau}$, and update $\tilde{m}^L$ (Line 18). If $\tilde{\tau}$ is empty, then the algorithm terminates successfully (Line 20). Else, repeat the same process to reduce the number of low-speed cores allocated to a task τ_i (Line 22). At any point, if cores are unavailable for task τ_i, then the task set is not schedulable and the algorithm returns failure. Upon successful completion, this algorithm greedily reduces the number of allocated low-speed cores by assigning (available) additional high-speed cores. If there are total K tasks in τ, the time complexity to calculate core allocation is $O(K)$.

▶ **Example 2.** Let us consider a platform with four low-speed cores of speed 0.75, and four high-speed cores of speed 1.0, two DAG tasks τ_1 and τ_2, where $\tau_1 = (12, 24, 4.08, 8.16, 20)$ and $\tau_2 = (12, 36, 3.36, 12.24, 33.12)$. Here, the low-speed and high-speed are normalized w.r.t. to

Algorithm 1 *greedyAlloc(τ).*

1 **Input:** The set of DAG tasks τ.
2 **Output:** Allocation of low(high)-speed cores to each task.
3 Unscheduled tasks, $\tilde{\tau} = \tau$; $\tilde{m}^L$ = *Available* LO*-speed cores*
4 **for** *(each $\tau_i \in \tau$)* **do**
5   **if** *($C_i^O/(s^L \times D_i) \leq 1$)* **then**
6   use traditional multiprocessor scheduling for sequential tasks; $\tilde{\tau} = \tilde{\tau} - \tau_i$;
7   **else if** *($C_i^N/s^L > C_i^O/s^H - L_i^O/s^H$)* **then**
8     $m_i^L = \left\lceil \frac{C_i^O - L_i^O}{D_i \times s^L - L_i^O} \right\rceil$; $\tilde{\tau} = \tau - \tau_i$;
9 $\tilde{m}^L = \tilde{m}^L - \sum_{\tau_i \in (\tau - \tilde{\tau})} m_i^L$, and $\forall_i m_i^H = 0$;
10 **for** *(each $\tau_i \in \tilde{\tau}$)* **do**
11   $m_i^L = \left\lceil \frac{C_i^O - L_i^O}{D_i \times s^L - L_i^O} \right\rceil$, and $\tilde{m}^L = \tilde{m}^L - m_i^L$;
12 **if** $\tilde{m}^L \geq 0$ **then**
13   $\forall \tau_i \in \tau$ allocate m_i^L and m_i^H cores and *RETURN SUCCESS*;
14 **else**
15   **for** *(each $\tau_i \in \tilde{\tau}$)* **do**
16     $m_i^H == 0 ? m_i^H = \lceil m_i^L \times s^L/s^H \rceil : (m_i^H = m_i^H + 1)$;
17   Update m_i^L as $\bar{m}_i^L$, where $\bar{m}_i^L = \left\lceil (C_i^N/s^L)/(D_i - \frac{L_i^O}{s^H} - \frac{C_i^O - C_i^N - L_i^O}{\bar{m}_i^H \times s^H}) \right\rceil$;
    $coreSaving_i = (m_i^L - \bar{m}_i^L)/(\bar{m}_i^H - m_i^H)$; ▷ *Relative core saving ratio of τ_i*
18   $maxCoreSaving = \forall_{\tau_i \in \tilde{\tau}} max(coreSaving_i)$;
19   **if** *(There are enough cores for τ_i)* **then**
20     $\tilde{\tau} = \tilde{\tau} - \tau_i$ and $\tilde{m}^L = \tilde{m}^L + (m_i^L - \bar{m}_i^L)$ ▷ *Update $\tilde{\tau}$ and $\tilde{m}^L$*
21   **if** $\tilde{\tau} = NIL$ **then**
22   *RETURN SUCCESS*;
23   **else**
24   Go to Line-9;
25   **else**
26   *RETURN FAILURE*;

maximum speed supported by this platform. Here, τ_1 is a category-1 task ($\frac{12}{0.75} > \frac{24}{1.0} - \frac{8.16}{1.0}$), and τ_2 is a category-2 task ($\frac{12}{0.75} < \frac{36}{1.0} - \frac{12.24}{1.0}$). For task τ_1, we calculate the required number of low-speed cores m_1^L as $m_1^L = \left\lceil \frac{C_i^O - L_i^O}{D_i \times s^L - L_i^O} \right\rceil = \left\lceil \frac{24 - 8.16}{20 \times 0.75 - 8.16} \right\rceil = 3$ (Line 9). Task τ_2 belongs to $\tilde{\tau}$, and we calculate the required number of low-speed cores m_2^L as $m_2^L = \left\lceil \frac{C_i^O - L_i^O}{D_i \times s^L - L_i^O} \right\rceil = \left\lceil \frac{36 - 12.24}{33.12 \times 0.75 - 12.24} \right\rceil = 2$ (Line 15). As there are not enough low-speed cores available for τ_2, two additional high-speed cores are allocated to τ_2 (Line 21). Now, τ_2 is removed from $\tilde{\tau}$ (Line 27). Both τ_1 and τ_2 receive the required number of cores and hence the algorithm terminates (Line 29).

5 Energy-aware Federated Scheduling for Multi-Speed Clustered Platform

We now describe how to allocate the low (or high) speed cores to each task. We extend the analysis presented in Sec. 4.1 to fit a multi-speed clustered platform, where different clusters offer different speeds. Hence, the energy consumption by different clusters (while

executing the same task) may vary significantly. For such a multi-speed clustered platform, we propose a task to cluster Hungarian assignment algorithm [36] to minimize the CPU energy consumption while satisfying the real-time schedulability guarantee.

5.1 Task to Cluster Assignment Approach

In this subsection, we discuss our approach to allocate all Category-1 (tasks with $C_i^N/s^L > C_i^O/s^H - L_i^O/s^H$) and Category-2 (tasks with $C_i^N/s^L < C_i^O/s^H - L_i^O/s^H$) DAG tasks into clusters, such that CPU energy consumption is reduced without violating the real-time guarantee. We assume that a task can not be allocated to multiple clusters.

Let, $\mathcal{Z}$ denotes the total number of available low-speed clusters, where each cluster is denoted as $\{\mathcal{K}_1, \mathcal{K}_2, \cdots \mathcal{K}_{\mathcal{Z}}\}$. We assume that each of these processors in the Kth cluster (where $1 \leq K \leq \mathcal{Z}$) execute at speed S_k. This assumption is motivated by Theorem 4 in [26], which asserted that executing a task with a consistent speed reduces the energy consumption significantly. In Section 4.1, we have shown the steps to determine the minimum number of processors to τ_i such that τ_i finishes execution within its deadline. That analysis assumes only two speed settings, i.e., low and high speed. Note that, different cluster offers different speed settings. Hence, for the same task, the required number of exclusively allocated processors may vary in different clusters. From now on, we assume these exclusively allocated processors form a *sub-cluster* inside the cluster. Let us assume that we know the number of available sub-clusters inside each cluster and each sub-cluster's size [3]. Let us denote a sub-cluster as $\mathcal{K}_m^n$ which denotes the nth sub-cluster inside m^{th} cluster, and $\mathcal{K}_m^n \in \{\mathcal{K}_1^1, \mathcal{K}_1^2, \cdots, \mathcal{K}_1^x, \cdots, \mathcal{K}_{\mathcal{Z}}^1, \cdots, \mathcal{K}_{\mathcal{Z}}^y\}$.

Algorithm 2 *createTable*$(\tau_{heavy}, \mathcal{K})$.

```
1  Input: The set of heavy DAGs τ_heavy and the set of sub-clusters
     K = {K_1^1, ..., K_1^x, ..., K_Z^1, ..., K_Z^y}.
2  Output: A table E storing the energy consumption value.
3  E[size(τ_heavy)][size(K)];  /*Store energy consumption*/;
4  for x = 1 to size(τ_heavy) do
5      for y = 1 to size(K) do
6          if the considered sub-cluster satisfy the minimum allocation requirement then
7              Calculate E_x^y;     /* Energy consumed by task x in sub-cluster y */
8              E[x][y] = E_x^y;
9          else
10             E[x][y] = ∞; /* Set to a very large value */;
```

Let a task τ_i is allocated to the Kth cluster, and it needs m_i^K cores. We conclude that, there exists an $\mathcal{K}_K^n \in \mathcal{K}_K$ such that m_i^K fits to $\mathcal{K}_K^n$. Now, we calculate the energy consumed by task τ_i (when executing in a sub-cluster $\mathcal{K}_K^n$), which is denoted as $E_i^{\mathcal{K}_K^n}$. We repeat this step for all task $\tau_i \in \tau$, and for all sub-cluster $\in \{\mathcal{K}_1^1, \mathcal{K}_1^2, \cdots, \mathcal{K}_1^x, \cdots, \mathcal{K}_{\mathcal{Z}}^1, \cdots, \mathcal{K}_{\mathcal{Z}}^y\}$. Refer to Section 3 for details regarding energy consumption.

Algorithm 2 starts by creating a table $\mathbb{E}$, which stores the energy consumption by a DAG task when allocated to a sub-cluster (Line 3). Then, it traverses each heavy DAG task τ_i and each sub-cluster $\in \{\mathcal{K}_1^1, \mathcal{K}_1^2, \cdots, \mathcal{K}_1^x, \cdots, \mathcal{K}_{\mathcal{Z}}^1, \cdots, \mathcal{K}_{\mathcal{Z}}^y\}$ (Lines 4-5). Then it checks whether

[3] The number and size of sub-clusters (inside any $\mathcal{K}_K$) depend on which task is allocated to $\mathcal{K}_K$. We handle the task-cluster allocation in Section 5.1.

the DAG task can be allocated to a sub-cluster (Line 7), i.e., the number of cores available in this sub-cluster satisfies the minimum processor required (refer to Subsection 4.1 and 4.2) for this DAG task. If it satisfies the constraints, we store the energy consumed by this DAG task (when allocated to this sub-cluster) at Table $\mathbb{E}$ (Line 8). Else, we put an arbitrarily large value to $\mathbb{E}$ (Line 10). We do this to ensure that the scheduler will never assign a DAG task τ_i to any sub-cluster that does not have enough cores to execute τ_i.

Determining the number of sub-cluster and number of cores inside each sub-cluster. So far, we have discussed how to create the energy consumption table and find the task to sub-cluster allocation using the information presented in this table. However, we did not mention the total number of available sub-cluster inside each cluster and the number of cores in each sub-cluster. Recall that the cluster speed influences the minimum number of cores required (i.e., the sub-cluster size) for any task τ_i. As we do not know the task-cluster mapping, we are unaware of the available sub-clusters inside any cluster. As a preliminary approach, we divide any cluster $\mathcal{K}_K$ into M_K sub-cluster, where:

$$M_K = \begin{cases} \left\lfloor \dfrac{M}{m^K} \right\rfloor + 1, & \text{if } M(mod\ m_K) \neq 0 \\ \dfrac{M}{m^K}, & \text{Otherwise} \end{cases} \tag{3}$$

Here, M is the number of cores inside any cluster $\mathcal{K}_K \in \mathcal{K}$. We calculate m^K as $m^K = max\{m_i^K\}$, for all tasks $\tau_i \in \tau$. Here, m_i^K is calculated using the analysis provided in Subsection 4.1. Each of these M_K sub-clusters contains m^K cores, if $M(mod\ m^K) = 0$. Else, the first $M_K - 1$ sub-clusters contain m^K cores, and the remaining sub-cluster contains $M - (M_K - 1) \times m^K$ cores. Note that partitioning a cluster with respect to the task that needs the *maximum* number of cores (in this cluster) may seem pessimistic. This is because any other task that is also allocated in $\mathcal{K}_K$ may not need m^K cores. To tackle this pessimism, we will update the sub-cluster number and their size (in Algorithm 3) until all the tasks are scheduled, or the algorithm returns failure.

Task to Cluster Assignment. Now we know the energy consumption at all possible combinations of the DAG task to the sub-cluster mapping. We use this information to determine the processor allocation that provides the minimum energy consumption. At each sub-cluster, we assign a task that is not allocated to any other sub-cluster previously – we can pick a single entry from each row and column in the energy consumption table. The pseudo-code for this approach is presented in Algorithm 3.

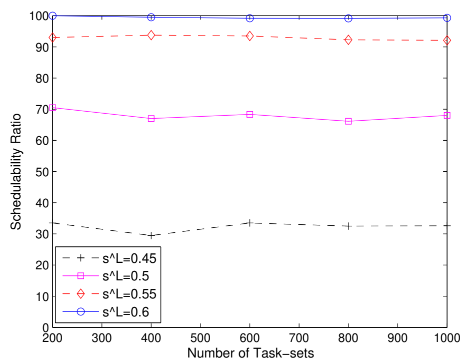
We determine the optimum assignment that minimizes the total energy consumption using the *Hungarian algorithm* [36] (Line 10). The algorithm takes the energy consumption table as input and returns an ordered collection of a task to sub-cluster allocation. The allocation provides the lowest combined energy consumption. The Hungarian algorithm has two significant advantages: it produces an optimal solution if the elements are non-negative (as in our case), and it has a polynomial complexity (i.e., affordable even for a large number of tasks). Note that the Hungarian algorithm works only when the input is an $N \times N$ square matrix. In our case, the energy consumption table may not be square in size. Hence, Algorithm 3 adds extra dummy rows in the table (to make it square in size) if the number of tasks is less than the number of sub-clusters, and fills them with arbitrary large values (Lines 6–9). Recall that we partition a cluster with respect to the task that needs the maximum number of cores (in this cluster), and it minimizes the number of sub-clusters in each cluster. Hence, some tasks may not get any sub-cluster, while some sub-clusters may

Algorithm 3 *taskToClusterAllocation*($\tau_{heavy}, \mathcal{K}$).

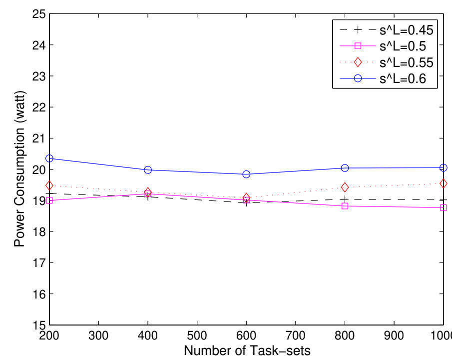
```
1  Input: The set of Category-1 heavy DAGs τ_heavy and the set of sub-clusters
     K = {K_1^1, ··· , K_1^x, ··· , K_Z^1, ··· , K_Z^y}.
2  Output: Processor to task allocation.
3  Set ∞ to 10^6    /* An arbitrary large value */;
4  createTable(τ_heavy, K)    /* The number of sub-clusters (in any cluster) is calculated using
     Equation (3) */
5  j = size(τ_heavy);  k = size(K);
6  if j < k then
7      for x = j + 1 to k do
8          for y = 1 to k do
9              E[x][y] = ∞;  /*Dummy row makes E square */
10 Solve E using the Hungarian algorithm [36].
11 for i = 1 to n do
12     if τ_i is allocated in any sub-cluster in K_x^y ∈ K then
13         τ_heavy = τ_heavy − τ_i    /* Update task set */;
14         if (size(K_x^y) − m_i^x) > 0 then
15             /* y^th sub-cluster of K_x has idle cores */;
16             size(K_x^y) = size(K_x^y) − m_i^x;
17 if 0 < size(τ_heavy) < j then
18     Repeat taskToClusterAllocation(τ, K);
19 else if size(τ_heavy) == 0 then
20     return the optimal processor to task allocation and allocate the remaining light DAGs to
         remaining cores of each sub-cluster;
21 else
22     Return FAILURE;
```

remain underutilized. To tackle this issue, we continuously check for the tasks that get an allocation and remove them from the task set τ (Line 13). If any sub-cluster is underutilized (i.e., a task receives more cores than required), we update the sub-cluster size (Lines 14–16). We repeat Algorithm 3 if some tasks are removed from the task set, i.e., some update in the task set takes place, but some tasks are still unassigned to any cluster (Lines 17–18). When all tasks are allocated to some sub-cluster (i.e., $size(\tau)$ becomes 0). Algorithm 3 concludes by returning the task to sub-cluster allocation that results in minimum energy consumption (Lines 19–20). Else, i.e., no task receives any cores as there are not sufficient cores, the algorithm stops and returns failure (Line 22).

Algorithm 3 performs a task to cluster allocation considering the Category-1 DAG tasks and low-speed clusters. We use a slightly modified version of Algorithm 3 to allocate Category-2 tasks to the remaining low-speed cores. When all Category-1 tasks receive the required number of low-speed cores, we use Algorithm 3 again to allocate the remaining low-speed cores to the Category-2 DAG tasks. This time the input to Algorithm 3 is the set of Category-2 DAG tasks and the set of sub-clusters $\mathcal{K} = \{\mathcal{K}_1^1, \cdots, \mathcal{K}_1^x, \cdots, \mathcal{K}_{\mathcal{Z}}^1, \cdots, \mathcal{K}_{\mathcal{Z}}^y\}$ which has some low-speed cores unused. If the available low-speed cores are insufficient to accommodate all the Category-2 tasks, we call Algorithm 3 again with a modified input parameter (i.e., set of Category-2 DAG tasks that do not receive enough low-speed cores and set of sub-clusters containing high-speed cores) to allocate additional high-speed cores.



(a) Schedulability ratio for different s^L values.



(b) Power consumption for different s^L values.

Figure 1 Schedulability ratio and power consumption for different s^L values.

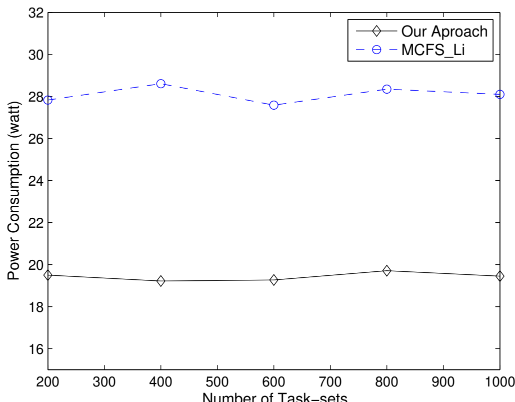


Figure 2 Power consumption comparison between our approach and MCFS_Li [39]. In this simulation, we set the value of s^L to 0.55.

6 Evaluation

This section demonstrates the algorithm's performance through evaluation conducted on a randomly generated task set. We report the *schedulability ratio* and the power consumption of our approach for different speed settings of a low-speed cluster. In this setup, the low-speed is normalized w.r.t. to the maximum speed supported by this platform, ranging from 0.45 to 0.6. We use the following parameters to generate the random task-set:

- ζ: number of task set, ranged between [200-1000].
- ζ_G: number of tasks per task set, ranged between [5-10].
- $D_i := xC_i^O$: relative deadline of a task, $x = [0.9 - 1.0]$.
- $[Z_{down}, Z_{up}]$: the range of the ratio of normal and overload execution budget. We set $1 \leq \frac{Z_{up}}{Z_{down}} \leq 8$.
- s^L: speed of the low-speed cluster, ranging [0.45-0.6].

The reference approach. To date, no work has investigated the same problem studied in this paper, i.e., minimize CPU power consumption for the DAG tasks by adopting the DPM policy in a clustered platform. Hence, we do not have a direct baseline to compare.

We consider a reference approach (for performance comparison) based on the DAG tasks scheduling [39], denoted as *MCFS_Li*. Similar to us, the reference approach has characterized a DAG task using its typical (and overload) execution requirement and the critical path length. The approach in [39] also proposed a core assignment to each DAGs. However, unlike us, [39] did not consider a multi-speed clustered platform. Hence, we assume that all the cores execute at the maximum speed (i.e., s^H) possible for the reference approach.

In Figure 1a, we vary the s^L values for a different size task set and report the schedulability ratio. As expected, the schedulability ratio is directly proportional to s^L, but not strongly correlated with with task-set size. Figure 1b shows the system power consumption for different s^L values over the different sizes of task sets. We see that energy consumption increases with a higher s^L value. In Figure 2, we compare our approach with an existing approach [39], denoted as MCFS_Li. We set the value of s^L to 0.55, and our approach leads to a power-saving of at least 29.23% compared to MCFS_Li (Figure 2). While guaranteeing real-time correctness, our approach utilizes the low-speed core as much as possible (Subsection 4.1 and 4.2), which leads to an energy-saving.

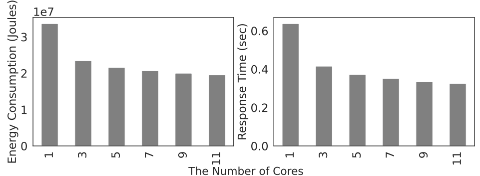
All these experiments in this work involve varying the number of randomly generated task sets. The aim is to investigate whether our proposed method's results are sensitive to changes in the number of tasks. However, our findings show no significant correlation between power consumption, schedulability ratio, and the number of tasks. This observation concludes that the proposed method is robust to different task set sizes.

7 Proof-of-Concept on Real-Time Platform Experiments

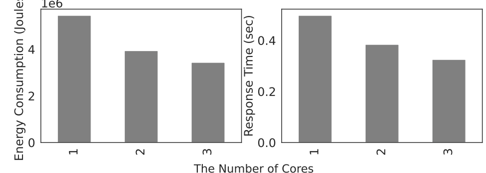
This section evaluates the proposed approach on a 12-core Intel Xeon 2680 v3 platform. The proposed method applies to ARM big.LITTLE architectures and modern Intel processors. However, the choice of the Xeon 2680 Processor stems from its well-studied power and energy consumption behavior [28], the ease of in-kernel status monitoring while having the per-core speed adjustment, per-core sleep, energy monitoring, and tracking the per-core temperature. The energy consumption behavior of modern Intel processors shows a significant deviation from energy models obtained on older Intel platforms due to latencies in changing a core's energy state, uncore frequencies, and out-of-order throttling at lower frequencies [57] – furthermore, documentation for turning off an entire cluster in ARM big.LITTLE architecture is sparse, and ARM does not provide public libraries for energy management.

On the platform, 11 of the 12 cores are isolated from kernel processes, user processes, and interrupts using `isolcpus` option in the kernel bios. Among these eleven cores, six cores represent Low-speed cores, and five cores represent High-speed cores. For a High-speed core, the minimum and maximum frequencies are normalized in the frequency range between 1.2 GHz and 2.5 GHz, respectively, with a minimum transition time of $20\mu s$. The minimum frequency of a Low-speed core is 1.2 GHz while the maximum frequency is a parameter of the evaluation. Additionally, each core can be independently turned off using DPM.

We conducted experiments on an Ubuntu 20.04 operating system, utilizing the default Completely Fair scheduler (CFS) introduced in Linux kernel version 2.6.23 [49]. The CFS scheduler is designed to allocate CPU times fairly among all runnable tasks on the system, making it ideal for our experiments. As a non-real-time scheduler, it provides a fair CPU time allocation among all runnable tasks on the system. We utilized the CFS scheduler by not specifying a scheduler type through the `sudo cset set` command. Our implementation of the proposed task-to-cluster allocation algorithm was written in Python. It periodically executed each benchmark task on the isolated 11 cores using the `cset` command-line option



(a) Correlation between response time and energy consumption of an OpenMP Benchmark Task on Xeon processor.



(b) Correlation between response time and energy consumption of an OpenMP Benchmark Task on Core i7 processor.

Figure 3 Change in energy consumption dependency with response time.

with a nice default value. In addition, our program modified kernel parameters, turned off appropriate clusters and monitored power consumption information from the allocated cluster. We used several average-case power management governors from the Linux Kernel to ensure a fair comparison, including schedutil, performance, powersave, conservative, and ondemand as baselines. Hyper-threading was turned off on all cores by adjusting the `scaling_max_freq` parameter in the Linux kernel. We fixed the frequency of each core using the `cpufrequtils` tool. Finally, we turned off Turbo mode to avoid unwanted frequency adjustments in each core.

The Barcelona OpenMP tasksuit (BOTS) [21] benchmark is used to evaluate the energy consumption of the proposed approach. Each of the 43 tasks within the BOTS benchmark follows the DAG task model discussed in Section 3. Nodes within a benchmark task are created using a task directive, while edges between nodes are generated using depend or taskwait directives. Nodes within a DAG task are scheduled using a greedy algorithm, as proposed in [37]. While intel p-state and c-state configurations transfer resource and power control to the hardware, we turned off this configuration for our application. We used advanced configuration and power interface (`ACPI`), which gives software access to touch voltage and frequency for speed adjustment and provides the baselines and `userspace` configuration. The `ACPI-Freq` is portable to other platforms.

To extract performance results, the `perf` Linux profiler and a hardware energy counter tool called reduced average power limit (`RAPL`) are used to monitor energy consumption during the execution of each DAG task [30]. Since the Linux profiler does not provide tool with the capability to profile multiple DAGs simultaneously, this evaluation focuses on evaluating single DAG tasks. This simplification is due to the limitation on profiling ring buffers and a socket hardware profiling register.

The paper proposes an approach for energy minimization while ensuring the schedulability of DAG tasks. Although other factors may affect the energy consumption in DAG benchmarks, including context switches, branch misses, and the number of instructions, these effects are out of the scope of this research. The correlation between response time and energy consumption is shown in Fig. 3, as we used the different numbers of cores. The results on Intel Xeon 2680 and core i7 show as we increased the number of cores, the energy consumption decreased on average on all the targeted benchmarks due to increasing the level of parallelism. When allocating only one core, the time to process increases, and the number of inactive cores adds much more waste on energy consumption.

This paper aims to determine the schedulability condition at runtime. To achieve this, we estimate DAG tasks' workload execution and critical path when executed at normal speed and allocate the number of cores accordingly. We assume the critical path can be determined when all cores are assigned to a DAG benchmark. Due to the absence of execution time

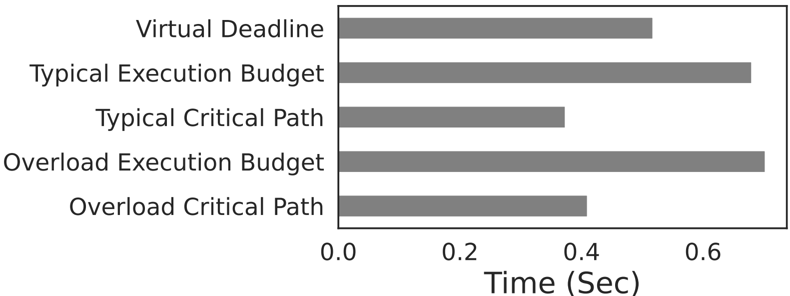


Figure 4 The response time of one DAG makespan is based on the critical path definition, workload execution time, and the virtual deadline.

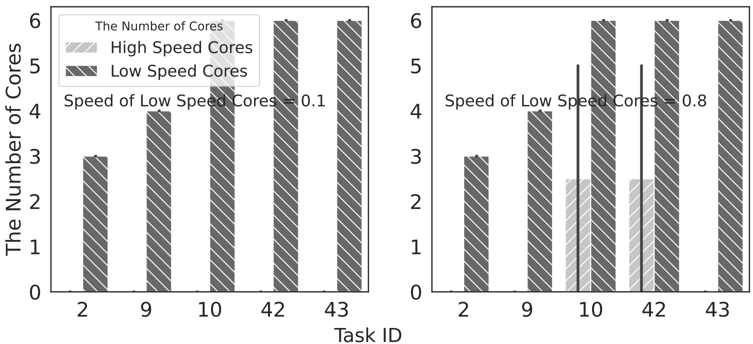


Figure 5 The light dash color refers to an increase in the number of high-speed cores due to the satisfaction of Category 2 in resource allocation.

tools for DAG tasks, the execution budget is obtained by running the DAG on a single core. From 20 corresponding iterations, the average value represents the typical condition, and the maximum value represents the overload condition. The results of this configuration for one task are shown in Fig. 4, and we obtain deadline constraints using Graham's bound. The virtual deadline is estimated using the following expression $VD < L + (C - L)/m$.

This paper defines different categories and allocates high-speed and low-speed cores based on core speeds and virtual deadlines. As explained in Section 4, there are three categories, and Category 2 involves meeting the condition $C_i^N/s^L < C_i^O/s^H - L_i^O/s^H$. This means that we may need to increase the speed of low-speed cores, decrease the speed of high-speed cores, or add high-speed cores to meet the required deadline. In theory, speed refers to the amount of computation in a given unit of time, while in experiments, it refers to the frequency of operation that determines the amount of computation that can occur in a given unit of time. Experiments utilize DVFS-enabled processors to control frequency within a predetermined range of frequencies. While the speed can theoretically range from 0 to 1, in practice, it is limited to the frequency range supported by the platform, as shown in Figure 5.

Based on Figure 5, the DAG tasks are assigned two low-speed and one high-speed core set. Six cores were assigned to low speed and five to high speed. In the experiments conducted for this paper, multiple clusters were used to test different scenarios where the number of low-speed and high-speed cores exceeded the schedulability requirements. Figure 5 shows an example where task 10 is allocated six low-speed cores with a speed of 0.1 and high-speed cores with a speed of 0.7 while maintaining schedulability. Similarly, this schedulability

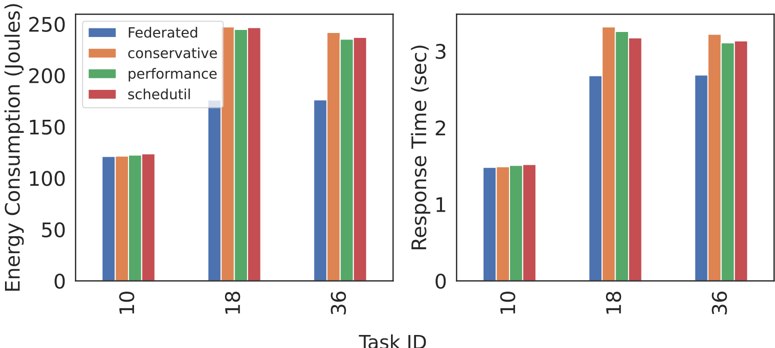
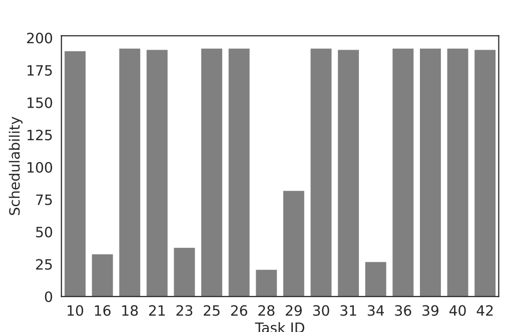


Figure 6 The federated energy-aware scheduling algorithm gives better results in terms of energy consumption and execution time compared to all the available Linux governors. Here task 18 is 30% better energy-efficient compared with the Linux governors.



(a) The schedulability of each DAG task for all the elements of clusters and sub-clusters in Algorithms 3. Having at least one schedulable task result means full schedulability in our results.

(b) Aggregated energy consumption of all the studied tasks versus the available linux governors shows 5% improvement over executing all the tasks.

Figure 7 Evaluation on schedulability and aggregated energy consumption.

can be achieved by increasing the number of high-speed cores or by increasing both the number and speed. To test energy efficiency under these conditions, we developed a clustering algorithm. Thus, cluster allocation is needed to get the optimal spot in energy minimization. We changed the speed of low-speed and high-speed cores to test the proposed method. We would fix the speed of high-speed cores to 0.7 of maximum speed in the Intel Xeon processor and observe the extra allocation of high-speed cores to heavy tasks in Fig. 5 as the speed of low-speed cores increases.

After setting the high and low-speed cores for each task and running the tasks in the directed acyclic graph (DAG) to create an energy matrix, we can accurately estimate the required low and high-speed cores. The results obtained with the proposed Federated scheduler were compared to state-of-the-art Linux governors, and it was found that the energy consumption was 30% less than the best result obtained from the governors, while the overall results were not worse than state-of-the-art. The advantage of this algorithm is that it takes schedulability conditions into account, unlike the Linux governors, which cannot do so.

The task to cluster allocation happens according to Algorithm 3, i.e., an energy value would be assigned to each table element according to the defined number of low(high)-speed cores and core's speed. We will allocate the low-speed cores based on the specified task categories. We would evaluate the conditions above requirements to fill the table. The element in the search table showing the optimal energy value would get the configuration needed for each task. The schedulability results in Fig. 7a for the table with 192 elements show we have at least one element in the search table, which follows the virtual deadline and schedulability conditions which means 100% schedulability all the time. While in Fig 6, we see some times 30% improvement in energy consumption on some tasks, the results show similar values in most of the evaluated tasks regarding all the times schedulability is met. The proposed algorithm represents results in Fig. 7b indicate an approximately 7% improvement in energy consumption when aggregating over all the tasks.

8 Conclusion and Future Work

The traditional workload model for real-time embedded systems focuses on worst-case behaviors to provide worst-case guarantees. However, modern embedded systems possess more energy constraints and require a richer system model to optimize energy consumption under typical scenarios instead of worst-case scenarios. In this work, we propose the energy-aware scheduling framework for DAGs in a clustered platform to minimize the typical-case energy while guaranteeing worst-case temporal correctness. Specifically, we propose determining the minimum number of low-speed cores required to schedule each DAG task under a dual-speed platform. If there are not enough low-speed cores, our algorithm assigns additional high-speed cores to a DAG, providing maximum energy-saving benefits. For multi-speed platform, we further propose a task to cluster partitioning approach to reduce the typical energy consumption without violating the worst-case real-time scheduling guarantee. We evaluate our algorithm via extensive simulations on randomly generated task set and report the energy consumption and the schedulability ratio. We also have implemented our algorithm on an Intel Xeon 2680 v3 platform and report that our approach reduces energy consumption by up to 30% w.r.t. the compared baseline.

Our current workload characterization assumes two thresholds: a typical execution length and a WCET. It would be interesting to study the situation when more detailed information can be provided, e.g., in the form of multiple thresholds, even with probability information. We also plan to investigate the impact of other components, e.g., cache misses, context switches, bus accesses, I/O usage, on the total power consumption. In the future, we plan to extend the evaluation to ARM big.LITTLE architecture and 12th generation Intel Core i7 clustered mobile platform with four High-speed and eight Low-speed cores.

References

1 Kunal Agrawal, Sanjoy Baruah, Pontus Ekberg, and Jing Li. Optimal scheduling of measurement-based parallel real-time tasks. *Real-Time Systems*, pages 1–7, 2020.

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

3 Arm a15 technical reference manual, 2013. https://developer.arm.com/documentation/ddi0438/i/functional-description/power-management/dynamic-power-management?lang=en.

4 Mario Bambagini, Mauro Marinoni, Hakan Aydin, and Giorgio Buttazzo. Energy-aware scheduling for real-time systems: A survey. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(1):7, 2016.

5 Sanjoy Baruah. The federated scheduling of systems of mixed-criticality sporadic DAG tasks. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 227–236. IEEE, 2016.

6 Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo DAngelo, 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 *2012 24th Euromicro Conference on Real-Time Systems*, pages 145–154. IEEE, 2012.

7 Sanjoy 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. *Journal of the ACM (JACM)*, 62(2):1–33, 2015.

8 Sanjoy Baruah, Vincenzo Bonifaci, and Alberto Marchetti-Spaccamela. The global EDF scheduling of systems of conditional sporadic DAG tasks. In *ECRTS*, 2015.

9 Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. A generalized parallel task model for recurrent real-time processes. In *RTSS*. IEEE, 2012.

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

11 Ashikahmed Bhuiyan, Di Liu, Aamir Khan, Abusayeed Saifullah, Nan Guan, and Zhishan Guo. Energy-efficient parallel real-time scheduling on clustered multi-core. *IEEE Transactions on Parallel and Distributed Systems*, 31(9):2097–2111, 2020.

12 Ashikahmed Bhuiyan, Federico Reghenzani, William Fornaciari, and Zhishan Guo. Optimizing energy in non-preemptive mixed-criticality scheduling by exploiting probabilistic information. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3906–3917, 2020.

13 Ashikahmed Bhuiyan, Sai Sruti, Zhishan Guo, and Kecheng Yang. Precise scheduling of mixed-criticality tasks by varying processor speed. In *Proceedings of the 27th International Conference on Real-Time Networks and Systems*, pages 123–132, 2019.

14 Ashikahmed Bhuiyan, Kecheng Yang, Samsil Arefin, Abusayeed Saifullah, Nan Guan, and Zhishan Guo. Mixed-criticality multicore scheduling of real-time gang task systems. In *RTSS*. IEEE, 2019.

15 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.

16 Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Sebastian Stiller, and Andreas Wiese. Feasibility analysis in the sporadic DAG task model. In *ECRTS*, 2013.

17 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.

18 Hui Cheng and Steve Goddard. Online energy-aware I/O device scheduling for hard real-time systems. In *Proceedings of the conference on Design, automation and test in Europe:*. European Design and Automation Association, 2006.

19 Alexei Colin, Arvind Kandhalu, and Ragunathan Rajkumar. Energy-efficient allocation of real-time applications onto heterogeneous processors. In *RTCSA*, 2014.

20 Alberto Corpas, Luis Costero, Guillermo Botella, Francisco D Igual, Carlos García, and Manuel Rodríguez. Acceleration and energy consumption optimization in cascading classifiers for face detection on low-cost arm big. little asymmetric architectures. *International Journal of Circuit Theory and Applications*, 46(9):1756–1776, 2018.

21 Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *2009 international conference on parallel processing*, pages 124–131. IEEE, 2009.

22 Pontus Ekberg and Wang Yi. Bounding and shaping the demand of generalized mixed-criticality sporadic task systems. *Real-time systems*, 50(1):48–86, 2014.

23 David Ferry, Jing Li, Mahesh Mahadevan, Kunal Agrawal, Christopher Gill, and Chenyang Lu. A real-time scheduling service for parallel tasks. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 261–272. IEEE, 2013.

24 Ana Guasque, Patricia Balbastre, Alfons Crespo, and Gerhard Fohler. Energy characterization of real-time partitioned systems. In *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 118–124. IEEE, 2018.

25 Zhishan Guo, Ashikahmed Bhuiyan, Di Liu, Aamir Khan, Abusayeed Saifullah, and Nan Guan. Energy-efficient real-time scheduling of DAGs on clustered multi-core platforms. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 156–168. IEEE, 2019.

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

27 Zhishan Guo, Luca Santinelli, and Kecheng Yang. EDF schedulability analysis on mixed-criticality systems with permitted failure probability. In *RTCSA*. IEEE, 2015.

28 Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. An energy efficiency feature survey of the intel haswell processor. In *2015 IEEE international parallel and distributed processing symposium workshop*, pages 896–904. IEEE, 2015.

29 Tarek Hagras and Jan Janecek. A high performance, low complexity algorithm for compile-time job scheduling in homogeneous computing environments. In *Parallel Processing Workshops*. IEEE, 2003.

30 Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring energy consumption for short code paths using rapl. *ACM SIGMETRICS Performance Evaluation Review*, 40(3):13–17, 2012.

31 Sebastian Herbert and Diana Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *ISLPED*. IEEE, 2007.

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

33 Kai Huang, Luca Santinelli, Jian-Jia Chen, Lothar Thiele, and Giorgio C Buttazzo. Applying real-time interface and calculus for dynamic power management in hard real-time systems. *Real-Time Systems*, 47(2):163–193, 2011.

34 Pengcheng Huang, Pratyush Kumar, Georgia Giannopoulou, and Lothar Thiele. Energy efficient DVFS scheduling for mixed-criticality systems. In *EMSOFT*. IEEE, 2014.

35 Connor Imes, David HK Kim, Martina Maggio, and Henry Hoffmann. Poet: a portable approach to minimizing energy under soft real-time constraints. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 75–86. IEEE, 2015.

36 Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

37 Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Analysis of global EDF for parallel tasks. In *ECRTS*. IEEE, 2013.

38 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 *ECRTS*. IEEE, 2014.

39 Jing Li, David Ferry, Shaurya Ahuja, Kunal Agrawal, Christopher Gill, and Chenyang Lu. Mixed-criticality federated scheduling for parallel real-time tasks. *Real-time systems*, 53(5):760–811, 2017.

40 Keqin Li. Energy efficient scheduling of parallel tasks on multiprocessor computers. *The Journal of Supercomputing*, 2012.

41 Xinmei Li, Lei Mo, Angeliki Kritikakou, and Olivier Sentieys. Approximation-aware task deployment on heterogeneous multi-core platforms with dvfs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

42 Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pages 88–98, 1995.

43 Di Liu, Jelena Spasic, Gang Chen, and Todor Stefanov. Energy-efficient mapping of real-time streaming applications on cluster heterogeneous mpsocs. In *ESTIMedia*. IEEE, 2015.

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

45 Agostino Mascitti and Tommaso Cucinotta. Dynamic partitioned scheduling of real-time dag tasks on arm big. little architectures. In *29th International Conference on Real-Time Networks and Systems*, pages 1–11, 2021.

46 Alexander Maxiaguine, Simon Kunzli, and Lothar Thiele. Workload characterization model for tasks with variable execution demand. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 2, pages 1040–1045. IEEE, 2004.

47 Sujay Narayana, Pengcheng Huang, Georgia Giannopoulou, Lothar Thiele, and R Venkatesha Prasad. Exploring energy saving for mixed-criticality systems on multi-cores. In *RTAS*. IEEE, 2016.

48 Odroid xu-3, 2017. `http://www.hardkernel.com/`.

49 Chandandeep Singh Pabla. Completely fair scheduler. *Linux Journal*, 2009(184):4, 2009.

50 Santiago Pagani and Jian-Jia Chen. Energy efficient task partitioning based on the single frequency approximation scheme. In *RTSS*. IEEE, 2013.

51 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.

52 Antonio Paolillo, Joël Goossens, Pradeep M Hettiarachchi, and Nathan Fisher. Power minimization for parallel real-time systems with malleable jobs and homogeneous frequencies. In *RTCSA*. IEEE, 2014.

53 Manar Qamhieh, Frédéric Fauberteau, Laurent George, and Serge Midonnet. Global EDF scheduling of directed acyclic graphs on multiprocessor systems. In *RTNS*. ACM, 2013.

54 Abusayeed Saifullah, Sezana Fahmida, Venkata P Modekurthy, Nathan Fisher, and Zhishan Guo. CPU energy-aware parallel real-time scheduling. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

55 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.

56 Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, 2013.

57 Robert Schöne, Thomas Ilsche, Mario Bielert, Andreas Gocht, and Daniel Hackenberg. Energy efficiency features of the intel skylake-sp processor and their impact on performance. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pages 399–406. IEEE, 2019.

58 Youngsoo Shin and Kiyoung Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Proceedings 1999 Design Automation Conference*, pages 134–139. IEEE, 1999.

59 Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *RTSS*. IEEE, 2007.

60 Kecheng Yang, Ashikahmed Bhuiyan, and Zhishan Guo. F2vd: Fluid rates to virtual deadlines for precise mixed-criticality scheduling on a varying-speed processor. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020.

61 Dakai Zhu, Nevine AbouGhazaleh, Daniel Mossé, and Rami Melhem. Power aware scheduling for and/or graphs in multiprocessor real-time systems. In *ICPP*. IEEE, 2002.

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

Bounding the Data-Delivery Latency of DDS Messages in Real-Time Applications

Gerlando Sciangula ✉
TeCIP Institute, Scuola Superiore Sant'Anna, Pisa, Italy
Huawei Research Center, Pisa, Italy

Daniel Casini ✉
TeCIP Institute, Scuola Superiore Sant'Anna, Pisa, Italy
Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Pisa, Italy

Alessandro Biondi ✉
TeCIP Institute, Scuola Superiore Sant'Anna, Pisa, Italy
Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Pisa, Italy

Claudio Scordino ✉
Huawei Research Center, Pisa, Italy

Marco Di Natale ✉
TeCIP Institute, Scuola Superiore Sant'Anna, Pisa, Italy
Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Pisa, Italy

Abstract

Many modern applications need to run on massively interconnected sets of heterogeneous nodes, ranging from IoT devices to edge nodes up to the Cloud. In this scenario, communication is often implemented using the publish-subscribe paradigm. The Data Distribution Service (DDS) is a popular middleware specification adopting such a paradigm. The DDS is becoming a key enabler for massively distributed real-time applications, with popular frameworks such as ROS 2 and AUTOSAR Adaptive building on it. However, no formal modeling and analysis of the timing properties of DDS has been provided to date. This paper fills this gap by providing an abstract model for DDS systems that can be generalized to any implementation compliant with the specification. A concrete instance of the generic DDS model is provided for the case of eProsima's FastDDS, which is eventually used to provide a real-time analysis that bounds the data-delivery latency of DDS messages. Finally, this paper reports on an evaluation based on a representative automotive application from the WATERS 2019 challenge by Bosch.

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

Keywords and phrases DDS, real-time systems, response-time analysis, end-to-end latency, CPA

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.9

Funding This work has been supported by EIT Urban Mobility, an initiative of the European Institute of Innovation and Technology (EIT), a body of the European Union. The work has also been partially supported by the project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU and the European Union's Horizon Europe Framework Programme project NANCY under the grant agreement No. 101096456.

1 Introduction

The Data Distribution Service (DDS) is a standard specification by the Object Management Group (OMG) describing a transfer protocol based on a data-centric publish-subscribe pattern (DCPS) [48]. With the advent of massively distributed applications, such as autonomous driving [9, 26, 31, 34], smart cities, Industry 4.0 [61], and more, the DDS gained a renewed in-

© Gerlando Sciangula, Daniel Casini, Alessandro Biondi, Claudio Scordino, and Marco Di Natale;
licensed under Creative Commons License CC-BY 4.0
35th Euromicro Conference on Real-Time Systems (ECRTS 2023).
Editor: Alessandro V. Papadopoulos; Article No. 9; pp. 9:1–9:26
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

terest in allowing communication among a vastly heterogeneous set of computing devices [42], such as those involved in the so-called IoT-to-Edge-to-Cloud compute continuum [5]. Furthermore, other popular frameworks, such as ROS 2 [12, 17, 21] and Autoware [34], build on top of the DDS to implement the publish-subscribe communication. In the automotive field, the AUTOSAR consortium has recently integrated DDS in its Adaptive Platform software standard [4]. Moreover, DDS support is being integrated in the next release of AUTOSAR Classic Platform [56]. In many of these applications, it is important to provide real-time guarantees on the delivery latency of messages passing through the DDS. However, the DDS is implemented as a complex multi-threaded middleware with threads that must be properly scheduled to achieve the desired real-time performance. These threads serve many purposes, from message dispatching, listening and liveliness monitoring, to garbage collection. Furthermore, some DDS threads implement custom, implementation-specific message queuing policies that can severely affect the message response times.

In this complex scenario, designers of real-time edge applications are called to provide proper values for several critical parameters, such as periods, application and DDS threads priorities, queue sizes, and others. Without fine-grained modeling and analysis of the system, designers can only rely on trial-and-error approaches, deploying system configurations and empirically assessing their performance, which is heavily time-consuming and error-prone.

Contribution. This paper provides a detailed modeling of DDS-enabled real-time systems. First, it provides a general model based on the DDS specification. Then, it shows how to instantiate it for the case of the eProsima's *FastDDS* [27], one of the most popular and efficient [57, 69] DDS implementation, leveraging an extensive exploration of the source code, documentation, and a set of experiments to validate the behavior inferred from the source code. Building on the model, we devise a response-time analysis for messages in a DDS-based distributed real-time system. The analysis can be used as an essential building block for future tools for design-space exploration of the system parameters, which can significantly help designers in configuring complex DDS-based systems. Finally, we evaluate our approach using the *WATERS 2019 Industrial Challenge* by Bosch [30], which represents a complex and real case of autonomous driving application, and we compare the analysis results with the latency values observed by running a simple FastDDS-enabled use-case application on a real platform.

2 Background

In this section, we review the DDS standard. Then, we highlight the peculiarities of FastDDS, i.e., the DDS implementation considered in this work, and we review the Compositional Performance Analysis (CPA) scheme, adopted in the paper.

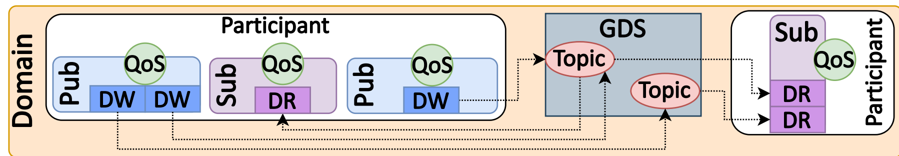


Figure 1 Example of connections between DDS participants in a domain.

2.1 The DDS Standard

The DDS standard specifies a data transfer protocol based on a Data-Centric Publisher-Subscriber (DCPS) pattern [48]. The DCPS model leverages the concept of a *Global Data Space (GDS)*, accessible to all the interested applications. Applications that provide information to the GDS declare their intent to become *Publishers*, whereas applications that want to access portions of the data space are identified as *Subscribers*. The DDS provides mechanisms for the exchange of data between these applications. Whenever a publisher publishes new data into the GDS, the middleware broadcasts this data to all interested subscribers. Moreover, the information flow is regulated by Quality of Service (QoS) policies at various levels of the communication stack [45]. According to the DDS specification, the transfer of any information happens in a logical area called *Domain*, which can be seen as a set of abstract links that connect all the communicating distributed applications. In any domain, there are several *Participants* and *Topics*. Topics are unambiguous identifiers that associate a name, unique within the Domain, to a data type and a set of attached data-specific QoS policies. Topics can be seen as channels for exchanging data. Participants are entities that can send and receive information from any topic in one Domain. A participant can include one or more publishers and/or subscribers. A publisher can send information over multiple different topics through `DataWriter (DW)` objects, and, similarly, a subscriber can receive data from different topics through `DataReader (DR)` objects. Each `DW` or `DR` object is linked to a single topic. Figure 1 shows an example of connections between DDS participants in a domain. The DDS leverages a lower-level protocol, the Real-Time Publish-Subscribe Protocol [46]. RTPS provides both best-effort and reliable publish-subscribe communications over unreliable transports, such as UDP, in both unicast and multicast settings. The OMG has standardized RTPS as the interoperability protocol for all the DDS implementations. Despite its name, RTPS does not define any real-time specific feature. The DDS operates in three main phases: **1)** *Discovery phase*, when the DDS participants find each other in the network, **2)** *Matching phase*, when the discovered participants determine if they should engage in a publish-subscribe relationship, and **3)** *Data Distribution phase*, when data is disseminated from the publishers to the matching subscribers.

2.2 The FastDDS Implementation

FastDDS is a C++ implementation of the DDS with a complex multi-threaded architecture, analyzed by means of code inspection. FastDDS threads are usually scheduled with the `SCHED_OTHER` (i.e., CFS) standard scheduler of Linux.

Publisher application. A publisher application consists of: a *publisher thread*, an *event thread*, and *meta-traffic listener threads*. The *publisher thread* is a user-level thread that manages a single publisher object. It is responsible for preparing and publishing application data on topics. The publishing of data can be **1)** *synchronous*, when it is performed by the publisher thread and **2)** *asynchronous*, when data is sent through the network on behalf of the publisher thread by a *flow-controller thread*, i.e., FastDDS internal middleware-level thread. If the publishing mode is asynchronous, the publisher thread inserts the new message into a queue of pending messages shared with the flow-controller thread. The queue contains messages related to different topics. The queue can be ordered according to three policies, i.e., `FIFO`, `RR` (round robin), `HIGH_PRIORITY` (fixed priority). The flow-controller thread is responsible for extracting data from the queue and sending it over the network. A publisher

thread can refer to multiple flow-controller threads if it publishes to multiple topics. The middleware-level *event thread* processes periodic and time-triggered events (mainly related to discovery/matching and QoS-checking services). The middleware-level *meta-traffic listener threads* manage the reception of discovery information.

Subscriber application. A subscriber application consists of: i) a *subscriber thread*, an *event thread*, a *user-traffic listener thread*, and *meta-traffic listener threads*. The *subscriber thread* is a user-level thread managing a single subscriber object, which is responsible for reading and interpreting data from topics. As for the publisher application, the subscriber application includes the middleware-level *event thread*. The middleware-level *user-traffic listener thread* manages the reception of user data (i.e., application data). The middleware-level *meta-traffic listener threads* manage incoming meta-traffic information.

Communication threads. FastDDS allows multiple publisher threads to publish data over the same topic. Similarly, multiple subscriber threads can subscribe to a specific topic. In this way, *many-to-many* communications between participants are supported. In FastDDS, the transport layer provides communication services between DDS entities, being in charge of sending and receiving messages over a physical transport [27]. Note that a listener thread is spawned for each reception channel, where the definition of channel depends on the adopted transport layer (UDP, TCP, or shared memory transport port).

2.3 Compositional Performance Analysis

CPA [32] is a framework for analyzing the timing behavior of complex heterogeneous and distributed real-time systems. CPA is built around two main concepts: *workloads* and computational *resources*. Workloads consist of tasks with precedence constraints. Applications are modeled as a direct acyclic graph (DAG) of communicating tasks. Groups of tasks execute on a *resource*, which provides the supply time and determines the resource-specific scheduling policy. In CPA, the source task of a chain is triggered according to an *externally provided event arrival curve* $\eta(\Delta)$, denoting an upper bound on the number of release events in any interval $[t, t+\Delta)$. Non-source tasks are triggered by *derived arrival curves*. Derived curves are obtained from arrival curves by accounting for the activation delay given by the completion times of predecessor tasks. The typical approach consists in accounting for a release jitter in non-source tasks that depends on predecessors' response times. The basic CPA analysis uses the sum of individual response-time bounds of each task to bound the end-to-end latency of a processing chain, while extensions have been designed to improve the precision in specific conditions [55].

3 Compositional DDS Model

Next, we model a DDS-based system in two steps. First, we provide a general and compositional model based on the DDS specification only that can be instantiated on any DDS implementation. Then, we show how to instantiate the model for the specific case of FastDDS.

We leverage a compositional approach to model the DDS middleware in an implementation-independent manner by mapping DDS operations to compositional *Logic Functional Blocks (LFBs)*. Each block describes the basic DDS operations. LFBs are divided into two categories: **(i)** *principal* blocks, which are directly involved in the data exchange from the publisher to the subscriber, and **(ii)** *auxiliary* blocks providing support (middleware) features such as discovery/matching, QoS-enforcement, or other implementation-specific services. Examples

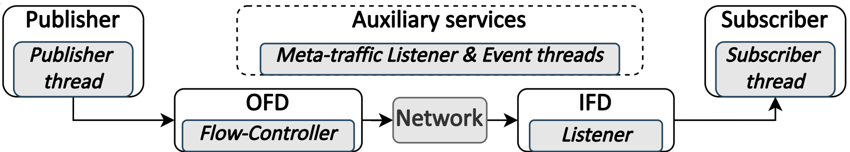


Figure 2 Instantiation of FastDDS threads on DDS model.

of DDS auxiliary implementation-specific services are represented by FastDDS's *Timed-Event handling* and Eclipse CycloneDDS's *Garbage Collector* and *Liveliness monitoring* [68]. *Publisher*, *Subscriber*, *Outgoing Flows Dispatching (OFD)*, *Network*, and *Incoming Flows Dispatching (IFD)* are principal LFBs. The Publisher and Subscriber blocks implement the fundamental publishing and subscribing operations, respectively, performed by user-level application-specific threads. The OFD block receives data from the Publisher block and controls the process of publishing it over the Network block. Note that the DDS standard does not define how data dispatching over the network should be implemented. The IFD block manages the procedure of processing messages received by the Network block. Furthermore, it is responsible to deliver messages to the Subscriber block. Finally, the Network block maps the functionalities of a network protocol and it is in charge of transmitting data over a communication link, from a source node to a destination node.

FastDDS instance of the model. Figure 2 shows the FastDDS implementation-specific instance of the abstract compositional DDS model. Each FastDDS thread we identified is mapped in its corresponding LFB. Meta-traffic listener and event threads have been mapped to the *auxiliary services block*. The publisher and subscriber threads have been mapped respectively to the *Publisher block* and *Subscriber block*. The flow-controller thread, discussed in Section 2.2, has been instantiated upon the *OFD block* when asynchronous-send mode is enabled. Similarly, the user-traffic listener (from now on, we refer to it simply as listener) thread has been mapped to the *IFD block*. Finally, the functionalities of a transport protocol and the network have been mapped onto the *Network block*.

4 FastDDS-based System Model and Problem Definition

The considered FastDDS-based system comprises a set C of cores, where each core $c_k \in C$ is possibly distributed onto multiple nodes in a distributed system.

Thread model. Four classes are used to identify the system threads: *publisher*, *flow-controller*, *listener*, and *subscriber*, contained in the sets Γ_p, Γ_f, Γ_l, and Γ_s, respectively. Threads are scheduled using a partitioned fixed-priority scheduler (each thread is statically allocated to a core). An arbitrary i-th thread belonging to each category is denoted as $\tau_i^\mathrm{p} \in \Gamma_\mathrm{p}$, $\tau_i^\mathrm{f} \in \Gamma_\mathrm{f}$, $\tau_i^\mathrm{l} \in \Gamma_\mathrm{l}$, or $\tau_i^\mathrm{s} \in \Gamma_\mathrm{s}$, respectively. The set $\Gamma_\mathrm{all} = \{\Gamma_\mathrm{p} \cup \Gamma_\mathrm{f} \cup \Gamma_\mathrm{s} \cup \Gamma_\mathrm{l}\}$ represents all the threads in the system. When the type of a thread is not relevant or clear from the context, the thread is simply denoted with τ_i. The set of *middleware-level* threads includes flow-controller and listener threads and is denoted with $\Gamma_\mathrm{mw} = \Gamma_\mathrm{f} \cup \Gamma_\mathrm{l}$. The set Γ_all^k includes all threads of any type running on core $c_k \in C$. $\Gamma_\mathrm{mw}^k \subseteq \Gamma_\mathrm{all}^k$ is the subset of the middleware threads on c_k. Each thread $\tau_i \in \Gamma_\mathrm{all}$ is associated with a unique fixed priority. Finally, we use the notation $\mathtt{hp}_{oth}^k(\tau_i) \subseteq \Gamma_\mathrm{all}^k \setminus \Gamma_\mathrm{mw}^k$ and $\mathtt{hp}_\mathrm{mw}^k(\tau_i) \subseteq \Gamma_\mathrm{mw}^k$ to indicate the set of non-middleware- and middleware-level threads, respectively, that run on core c_k and have priority higher than $\tau_i \in \Gamma_\mathrm{all}^k$.

Topic and message model. To define the logical communication channels between publisher and subscriber applications, we define a set of topics Θ. Each topic $\theta_j \in \Theta$ has a unique priority within the whole system, *which is independent of the priorities of the corresponding threads*, discussed in the *Thread model*. The topic priority is then inherited by each instance of any message $m_z(\tau_i^{\mathrm{p}}, \theta_j)$ published by the publisher thread τ_i^{p} over the topic θ_j. We simply use the symbol m_z whenever it is not needed to identify the publisher thread and the topic. An instance of a message m_z is said to be *pending* in a middleware-level thread $\tau_i \in \Gamma_{\mathrm{mw}}$ from when it is released in the thread to when its processing completes in the middleware-level thread. The set of messages associated with a topic θ_j is denoted with $\mathcal{M}(\theta_j)$. Each instance of the publisher thread τ_i^{p} can send up to w_i^j messages to topic θ_j. We define $\Theta(\tau_j^{\mathrm{s}}) \subseteq \Theta$ as the subset of the topics from which a subscriber thread τ_j^{s} can receive messages. $N_{\mathrm{sub}}(m_z)$ denotes the number of subscribers interested in message m_z.

Association among threads. A publisher thread τ_i^{p} can be associated to multiple flow-controller threads $\tau_i^{\mathrm{f}} \in \Gamma_{\mathrm{f}}$ if it publishes to multiple topics. A subscriber thread τ_j^{s} is associated to a unique listener thread τ_j^{l}, which can handle messages from different topics. A pair (publisher thread, topic) $(\tau_i^{\mathrm{p}}, \theta_j)$, and therefore a message $m_z(\tau_i^{\mathrm{p}}, \theta_j)$, is associated with a single flow-controller thread and a single listener thread. The association of a message to a middleware thread is denoted by $m_z \in \tau_i^{\mathrm{t}}$, with $t \in \{\mathrm{f}, \mathrm{l}\}$.

Execution times and activations. Non-middleware-level threads $\tau_j \in \Gamma_{\mathrm{all}} \setminus \Gamma_{\mathrm{mw}}$ are characterized by a worst-case execution time e_j. This paper considers a discrete-time model, i.e., all time parameters are integer multiples of a basic time unit (e.g., a processor cycle), defined as $\epsilon \triangleq 1$. Each publisher thread τ_i^{p} is characterized by an *externally-provided* event arrival curve $\eta_i^p(\Delta)$. Subscriber thread instances are triggered in a data-driven fashion. Therefore, each subscriber thread τ_j^{s} is associated with a *derived* arrival curve $\eta_j^s(\Delta)$, which depends on the response times of the message triggering the computation. We show later in Section 5.2 how to derive such curves. Differently, the worst-case execution time and activation patterns of flow-controller and listener threads are determined by the arrival patterns and message-processing delays of the messages. We denote with $\eta_{z,i}^{\mathrm{f}}(\Delta)$ and $\eta_{z,j}^{\mathrm{l}}(\Delta)$ the derived arrival curve of each message in their flow-controller thread τ_i^{f} and listener thread τ_j^{l}, respectively. Whenever it is not relevant whether τ_i is a flow-controller or a listener thread, we simply denote the arrival curve of a message with $\eta_{z,i}(\Delta)$. The parameters $\delta^{\mathrm{f}}(m_z)$ and $\delta^{\mathrm{l}}(m_z)$ denote the worst-case time required to process a message m_z in its flow-controller and listener threads, respectively, without the interference of any other message and thread. In the flow-controller, this time is required to execute a single system send call, while in the listener involves the deserialization of a single message and delivery of the message to the subscriber object. In both cases, the message size affects the parameter. The network propagation delay of a message m_z is denoted as $\delta^{\mathrm{net}}(m_z)$. It can be either pragmatically estimated or analytically bounded, depending on the underlying network [23, 35, 67].

Flow-controller scheduling policies. We define the available scheduling policies of flow-controller threads as `HP` and `F`, denoting the `HIGH_PRIORITY` and `FIFO` policies as defined by FastDDS, respectively. The analysis of the RR policy is left as future work. Within the same flow-controller, messages that have the same priority (related to the same topic) are processed in FIFO order. When using the `HP` policy, given an arbitrary message m_z and a flow-controller thread τ_i, the symbols $hp_i(m_z)$, $ep_i(m_z)$, and $lp_i(m_z)$ denote the set of all the messages with higher, equal, and lower priority than m_z in τ_i, respectively.

Table 1 Table of main symbols.

Sym.	Description	Sym.	Description
c_j	j-th physical core	τ_i^{t}	i-th thread of type $t \in \{\mathtt{p}, \mathtt{f}, \mathtt{l}, \mathtt{s}\}$
Θ	set of topics	$m_z(\tau_i^{\mathrm{P}}, \theta_j)$	z-th *msg* published by τ_i^{P} over θ_j
θ_j	j-th topic	w_i^j	max. num. *msgs* to θ_j for each τ_i^{P} instance
$\mathcal{M}(\theta_j)$	*msgs* for a topic θ_j	$\Theta(\tau_j^{\mathrm{s}})$	topics from which τ_j^{s} receives *msgs*
Γ_{t}	threads of type t	$N_{\mathrm{sub}}(m_z)$	num. of *subscribers* subscribed to m_z
Γ_{mw}	middleware threads	$\mathtt{hp}_{\mathrm{mw}}^k(\tau_i)$	mw-thrds with pr. higher than τ_i on c_k
Γ_{all}	all threads	$\mathtt{hp}_{oth}^k(\tau_i)$	non-mw-thrds with pr. higher than τ_i on c_k
Γ_{t}^k	threads of type t on c_k	$rbf_i(\Delta)$	request-bound function of τ_i
e_j	WCET of τ_j^{t}, $t \in \{\mathtt{p}, \mathtt{s}\}$	$sbf_k(\Delta)$	supply-bound function of c_k
$\eta_i^t(\Delta)$	τ_i^{t} arr. curve, $t \in \{\mathtt{p}, \mathtt{s}\}$	$\eta_{z,i}^{\mathrm{t}}(\Delta)$	arrival curve of m_z in τ_i^{t}, $t \in \{\mathtt{f}, \mathtt{l}\}$

Static discovery. In this paper, we consider a static network of publishers and subscribers, meaning that no new participant join at run-time. Under this assumption, the overhead due to the discovery mechanism becomes negligible by leveraging the FastDDS *Static Discovery* [27]. This configuration implies that, after static discovery is over, the network of entities is fixed, and no other discovery messages are exchanged among them. Thus, delay due to meta-traffic listener and event threads becomes negligible (auxiliary services block in Fig. 2), since they are responsible for processing discovery periodic events (e.g., sending of heartbeat messages for remote node liveliness) and QoS-checking services.

Listener threads. Each listener thread handles one network socket through which the thread receives data related to different topics, possibly sent by different publishers. Incoming messages are processed in `FIFO` order.

Queues. When using the `FIFO` policy, each flow-controller (or listener) thread $\tau_j^{\mathrm{t}} \in \Gamma_{\mathrm{mw}}$, with $t \in \{f, l\}$, manages one queue of pending messages that can contain at most $M_j^{\mathtt{F}}$ messages. Differently, using the `HP` policy, each priority-level i corresponds to a queue of size $M_j^{\mathtt{HP},i}$. Note that `HP` is only used for flow-controller threads. Buffers should be large enough so that no messages are dropped at both the sender and receiver sides.

Supply-bound function. In this work, analysis and results rely on the existence of a supply-bound function $sbf_k(\Delta)$ that denotes the minimum time of processor service provided by a core $c_k \in C$ in any time window of length Δ, [16, 40, 60]. This abstraction is useful to make the analysis extensible with reservation-based scheduling mechanisms [1], such as those implemented by the `SCHED_DEADLINE` scheduling class of Linux [39] or by the QNX Adaptive Partitioning Scheduler [22], and naturally generalizes to the case without reservation (if a core is fully available, $sbf_k(\Delta) = \Delta$).

Table of symbols. Table 1 summarizes the main symbols introduced in this paper.

4.1 Problem Statement

The metric of interest for the analysis in this paper is:

▶ **Definition 1** (Data Delivery Latency). *The Data Delivery Latency (DDL) L_z experienced by a message m_z sent by a publisher thread to a matching subscriber thread is the longest time span elapsed between the time instant in which m_z is sent by the publisher and the time instant when the corresponding instance of the subscriber thread is released.*

Note that, as described next in Section 4 (Execution times and activations), for each subscriber thread, *exactly* one subscriber thread instance is triggered for each received message instance. Moreover, the DDL only accounts for the time spent by a message in the middleware-level threads, and does not include the time in which the target instance of the subscriber is waiting for being scheduled. Our goal is to leverage the model to devise a real-time analysis capable of bounding the worst-case *data delivery latency* of any DDS message.

4.2 Thread behavioral rules

Next, we formalize the behavior of the FastDDS middleware implementation through a set of rules, considering the interactions between the modeled threads.
R1 – Pub-to-Flow: When a publisher thread needs to send data over a topic, it performs a `write` operation and notifies the corresponding flow-controller thread.
R2 – Flow-to-Net: If the queue of pending messages is not empty, the flow-controller thread extracts a message from the head of the queue, arranges the RTPS packet (serialization), and performs a system network send operation for each interested subscriber. The number of send operations corresponds to the number of message copies to be sent to each subscriber, expressed by the parameter $N_{\text{sub}}(m_z)$. When the queue is empty, the flow-controller thread blocks until its associated publisher thread notifies it with new data to send.
R3 – Net-to-List: A listener thread performs a blocking system network `receive` operation on a socket. Whenever a message is received and written to a socket buffer by the system network functionalities, the listener thread is woken up and becomes ready to process incoming messages.
R4 – List-to-Sub: The listener thread takes a message from the socket buffer, following a `FIFO` pattern. Then, it deserializes the message. The message is then delivered to the subscriber thread, which is notified of the new message presence.
R5 – Non-preemptiveness: The send operation of a message is *non-preemptive*, meaning that, if a message has been extracted from the queue, it and all of its copies to different subscribers are sent over the network, even if a higher priority message has arrived in the meantime.
R6 – `HIGH_PRIORITY (HP)` policy: Under this policy, each message is assigned to a priority inherited from the corresponding topic. Messages are handled in the flow-controller thread in priority order, from the highest to the lowest.
R7 – `FIFO (F)` policy: Under this policy, the flow-controller thread handles each message in a first-in-first-out fashion.
R8 – Work-conservation: Flow-controller and listener threads never become idle if there are messages to be served.

4.3 Model Validation

The model and the above behavioral rules have been derived with a deep inspection of the FastDDS documentation and source code (*GitHub* repository [28]). To corroborate our findings with empirical evidence, we performed several experiments to figure out interactions between threads by focusing on shared data structures and condition variables. To this end, an application constituted of three publishers and one subscriber exchanging data over three topics was executed on two desktop machines running Ubuntu 20.04 and interconnected through a point-to-point Ethernet link using UDP communication. Furthermore, we designed ad-hoc experiments to corroborate the behavior of the two scheduling policies of the flow controller. In each experiment, the subscriber is subscribed to three topics (θ_1, θ_2, θ_3) on

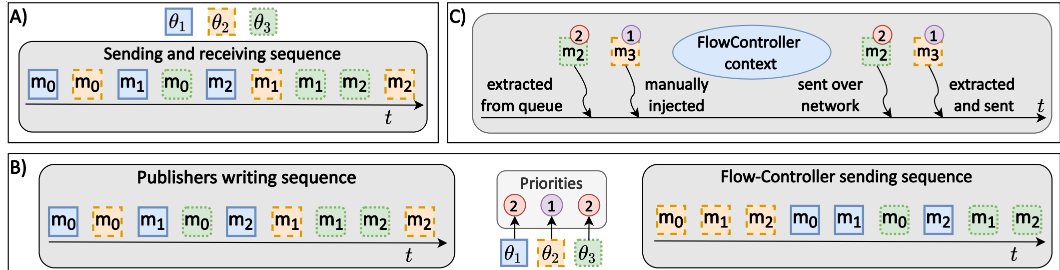


Figure 3 Validation experiments for R5 (C), R6 (B), and R7 (A) rules.

which each publisher publishes three messages over one of the topics. Each message payload contains the timestamp of the moment when it is sent on the network. Subscribers and publishers run on different machines.

`FIFO` policy. First, the flow-controller was configured to work with the `FIFO` policy. To corroborate rule **R7**, we checked that at the subscriber listener side, messages were received from the oldest timestamp to the most recent one. Figure 3 (A) shows the result of this experiment: the sequence of messages sent over the network by the flow controller are in FIFO order as the order observed by the sender and the receiver corresponds[1].

`HIGH_PRIORITY` policy. A similar experiment was performed to check the behavior of the `HIGH_PRIORITY` policy (rule **R6**). In this experiment, each topic is assigned to a priority. Topic θ_2 is assigned to priority 1, which is the highest of this configuration. θ_1 and θ_3 are both assigned to priority 2. Figure 3 (B) shows the results of this experiment. As expected, messages related to topic θ_2 (with the highest priority) were sent first on the network, while the messages related to the topics with the same priority were handled in FIFO order.

Non-preemptiveness. We checked the non-preemptiveness (rule **R5**) of the sending operation, modifying the previous experiment. Referring to Figure 3 (B), when the last message related to topic θ_3 has already been extracted from the pending message queue, we manually injected, by modifying the source code, a new higher-priority message (i.e., m_3 related to topic θ_2) in the queue, as shown in Figure 3 (C). Even if the new message should be processed first according to the priority order, the flow-controller thread waits until the current message was sent, before processing the highest-priority one.

Making FastDDS more predictable. We introduced two features to improve FastDDS predictability, considering a Linux-based system, and to show that it is possible to make FastDDS fully compliant with our model. We leveraged these changes in the comparison of analysis-driven and empirical latencies in Section 6.2. First, FastDDS does not provide any mechanism to set scheduling properties for its internal threads, such as thread priorities and usage of the Linux's fixed-priority scheduler to schedule such threads. Therefore, we modified FastDDS to introduce a new scheduling service able to initialize these parameters at the system start time. Second, message queues (e.g., in the flow-controller) are unbounded

[1] In principle, it would have been possible to observe out-of-order delivery due to data streams following multiple paths through the network and the lack of flow control mechanisms in UDP protocol [70]. Our experiment leveraged a point-to-point connection to mitigate the issue.

Figure 4 Source-Destination data path and arrival curve propagation.

by default, which may lead to unbounded growth of memory if the system is flooded with a considerably large amount of messages (e.g., due to a distributed denial-of-service attack). Hence, we modified the source code to comply with a limited size. Methods to derive a suitable size for such queues will be derived in future work.

5 Data-Delivery Latency Analysis

Figure 4 leverages the FastDDS instance of the compositional model to summarize the message path from the publisher to the subscriber. Once the publisher thread prepares new data to transmit, such data is inserted in a queue of pending messages managed by the flow controller thread, which sends messages through the network. When the subscriber listener thread receives the message, it is processed and delivered to the user-level subscriber thread. To bound the DDL of an arbitrary message m_z we provide bounds for the worst-case response-time experienced by each message in each middleware thread of interest, namely, the flow-controller and listener threads. The worst-case response time of a message m_z in a middleware (either the flow-controller or the listener) thread τ_i^{t}, with $t \in \{f, l\}$, is the longest time span from the release of the message instance in the thread to when the message instance processing completes. We denote with the symbols $R_x^{\text{f}}(m_z)$ and $R_y^{\text{l}}(m_z)$ a *response-time bound* for message m_z in the associated flow-controller thread τ_x^{f} and listener thread τ_y^{l}, respectively. Whenever specifying the involved thread is not needed or clear from the context, we simply write $R(m_z)$. Note that all the threads involved in the communication can be allocated to arbitrary cores. The following analysis leverages the knowledge of arrival curves of messages at the flow-controller and listener threads: we show later in Section 5.2 how to derive them. Following CPA [32] and by rules **R1-R4**, the DDL L_z of an arbitrary message m_z can be bounded as the sum of the individual worst-case delays experienced in the network and flow-controller and listener threads, i.e.,

$$L_z = R_x^{\text{f}}(m_z) + R_y^{\text{l}}(m_z) + \delta^{\text{net}}(m_z). \tag{1}$$

For each thread $\tau_i \in \Gamma_{\text{all}} \setminus \Gamma_{\text{mw}}$, the symbol $rbf_i(\Delta)$ denotes its *request-bound function* (RBF). The RBF returns the maximum processor time needed by the thread instances of τ_i in any interval of length Δ, i.e., $rbf_i(\Delta) = \eta_i^t(\Delta) \cdot e_i$, with $t \in \{p, s\}$ [15, 17]. The sum of request-bound functions of an arbitrary set of threads Γ' is referred to as $RBF(\Gamma', \Delta) = \sum_{\tau_j \in \Gamma'} rbf_j(\Delta)$.

5.1 Response-Time Analysis for a Fast-DDS message

Definitions. To bound the worst-case response time of a message m_z while being processed by middleware thread τ_i^{t}, with $t \in \{f, l\}$, or simply τ_i if the type is not needed, we start defining the sources of interference that can delay m_z, and the corresponding bounds. We start from the *thread-level* interference, which depends on higher-priority non-middleware threads running on the same core.

▶ **Definition 2** (Thread-level Interference). *The thread-level interference $I_{i,z}^{thread}(\Delta)$ is an upper bound on the delay suffered by an arbitrary instance of m_z while pending in middleware thread $\tau_i \in \Gamma_{mw}^k$, in any time interval of length Δ, due to non-middleware threads $\tau_j \in \Gamma_{all}^k \setminus \Gamma_{mw}^k$ allocated on the same core c_k.*

Other sources of interference are due to messages. This interference can be either due to: **(i)** messages handled in other middleware threads with higher priority running on the same core, or **(ii)** messages handled in the same middleware thread τ_i under analysis. We call (i) *inter-thread message interference*, and (ii) *intra-thread message interference*.

▶ **Definition 3** (Inter-Thread Message Interference). *The inter-thread message interference $I_{i,z}^{inter}(\Delta)$ is an upper bound on the delay suffered by an arbitrary instance of m_z while being pending in middleware thread $\tau_i \in \Gamma_{mw}^k$, in any time interval of length Δ, due to the processing of other messages by high-priority middleware-level threads $\tau_j \in \boldsymbol{hp}_{mw}^k(\tau_i)$ on the same core c_k.*

▶ **Definition 4** (Intra-Thread Message Interference). *The intra-thread message interference $I_{i,z}^{intra}(\Delta)$ is an upper bound on the delay suffered by an arbitrary instance of m_z, in any time interval of length Δ, due to messages processed by the same middleware-level thread $\tau_i \in \Gamma_{mw}^k$ where m_z is pending.*

Note that Definition 4 includes both interference due to instances of other messages $m_r \neq m_z$, and from other instances (previously released) of the same message under analysis. We call the latter *self-interference*, and the corresponding instances *self-interfering instances*.

Policy-independent bounds. Now, we instantiate the previously defined interference bounds, and we finally derive a generic response-time bound for a message in a middleware-level thread, which can be used for both flow-controller and listener threads. We start presenting bounds for the thread-level and inter-thread message interference, which are independent of the scheduling policy adopted in the middleware-level thread. To this end, Lemma 5 bounds the number of pending message instances in a middleware-level thread.

▶ **Lemma 5.** *Let $\overline{R}(m_z)$ be a response-time bound for m_z in an arbitrary middleware-level thread τ_i. In any interval of length Δ, there are at most $\eta_{z,i}(\Delta + \overline{R}(m_z) - \epsilon)$ pending instances of m_z in τ_i.*

Proof. Consider an arbitrary interval $[\bar{t}, \bar{t} + \Delta)$, with $\Delta > 0$. First, note that instances of m_z released at or after $\bar{t} + \Delta$ are not pending in $[\bar{t}, \bar{t} + \Delta)$. Note that $\eta_{z,i}(\Delta + \overline{R}(m_z) - \epsilon)$ counts all the instances released in $(\bar{t} - \overline{R}(m_z), \bar{t} + \Delta)$, which has length $\Delta + \overline{R}(m_z) - \epsilon$. By contradiction, assume there are more than $\eta_{z,i}(\Delta + \overline{R}(m_z) - \epsilon)$ pending instances in τ_i. Then it means there exists an instance of m_z released at or before $\bar{t} - \overline{R}(m_z)$ that is still pending in $[\bar{t}, \bar{t} + \Delta)$. This leads to a contradiction because $\overline{R}(m_z)$ is a response-time bound for m_z. ◀

While Lemma 5 is presented as a mean to derive a response-time bound for m_z, it requires, in turn, a pre-existing response-time bound $\overline{R}(m_z)$, hence introducing a circular dependency. The same notation for pre-existing bounds is used also in the presentation of the following results: the dependency can be solved by using standard real-time analysis techniques [32] that provide an outer response-time analysis loop and initially set $\overline{R}(m_z) = 0$, deriving a response-time estimate at every iteration, and updating $\overline{R}(m_z)$ until a global fixed-point is achieved. The procedure is guaranteed to converge since response-time estimates never decrease [32]. Further details are provided next in Section 5.3.

Next, Lemma 6 bounds the thread-level interference.

▶ **Lemma 6.** *Let τ_i be a thread handling an instance of message m_z (either as flow-controller or listener thread) running on c_k. In any interval of length Δ, the corresponding thread-level interference is bounded by*

$$I_{i,z}^{thread}(\Delta) \triangleq RBF(\boldsymbol{hp}_{oth}^{k}(\tau_i), \Delta). \tag{2}$$

Proof. By definition, the thread-level interference involves all non-middleware threads with a higher priority than τ_i. These threads are contained into the set $\mathtt{hp}_{oth}^{k}(\tau_i)$. The lemma follows by noting that $RBF(\mathtt{hp}_{oth}^{k}(\tau_i), \Delta)$ sums all terms $rbf_h(\Delta)$ due to each $\tau_h \in \mathtt{hp}_{oth}^{k}(\tau_i)$, where each term $rbf_h(\Delta)$ bounds the individual demand due to τ_h , in any interval of length $\Delta > 0$. ◀

Next, we consider the interference due to messages. Before starting, we derive a bound on the delay due to each message processed by a flow-controller or listener thread.

▶ **Lemma 7.** *The delay due to a single instance of an interfering message m_z in a middleware-level thread $\tau_i^t \in \Gamma_{mw}$ is bounded by*

$$\delta_i^t(m_z) \triangleq \begin{cases} \delta^f(m_z) \cdot N_{sub}(m_z) & \text{if } t = f, \\ \delta^l(m_z) & \text{if } t = l. \end{cases} \tag{3}$$

Proof. Recall that the delay due to an instance of message m_z is equal to $\delta^{\mathrm{f}}(m_z)$ for a flow-controller thread and $\delta^{\mathrm{l}}(m_z)$ for a listener thread. By rule **R2**, for each instance of a message m_z sent by a publisher, the flow controller sends $N_{\mathrm{sub}}(m_z)$ copies towards subscribers. This leads to a delay of $\delta^{\mathrm{f}}(m_z) \cdot N_{\mathrm{sub}}(m_z)$, proving the first branch of Equation (3). The second branch follows by noting that, due to rule **R4**, for each message instance processed by the listener only one message instance at a time is forwarded to the subscriber. ◀

With the previous result in place, Lemma 8 bounds the inter-thread message interference experienced by an arbitrary message m_z under analysis.

▶ **Lemma 8.** *Consider a message m_z in a middleware-level thread $\tau_i^t \in \Gamma_k^{mw}$. Let $\overline{R}_j^t(m_r)$ be a response-time bound for m_r in an arbitrary middleware-level thread $\tau_j^t \in \boldsymbol{hp}_{mw}^{k}(\tau_i^t)$. In any window of length $\Delta > 0$, the inter-thread message interference of an instance of m_z is bounded by:*

$$I_{i,z}^{inter}(\Delta) \triangleq \sum_{\tau_j^t \in \boldsymbol{hp}_{mw}^{k}(\tau_i^t)} \sum_{m_r \in \tau_j^t} \eta_{r,j}(\Delta + \overline{R}_j^t(m_r) - \epsilon) \cdot \delta_j^t(m_r), \text{ with } t \in \{f, l\} \tag{4}$$

Proof. By definition, $I_{i,z}^{\mathrm{inter}}(\Delta)$ includes all the interference due to messages in other middleware level threads $\tau_j^{\mathrm{t}} \in \mathtt{hp}_{\mathrm{mw}}^{k}(\tau_i^{\mathrm{t}})$ on the same core c_k. The first summation sums over all such threads, and the second over all messages handled by each thread. The lemma follows by recalling that, by Lemmas 5 and 7, each of such messages contributes with at most $\eta_{r,j}(\Delta + \overline{R}_j^t(m_r) - \epsilon)$ instances, each one with a delay of at most $\delta_j^t(m_r)$. ◀

Policy-dependent bounds. Next, we present the bounds on the intra-thread message interference, which depends on the policy used in the middleware-level thread. Before proceeding, we bound the number of self-interfering instances in Lemma 9.

▶ **Lemma 9.** *Let $\overline{R}(m_z)$ be a response-time bound for m_z in an arbitrary middleware-level thread. In any interval of length Δ, the number of self-interfering instances to an arbitrary instance of a message m_z in a middleware-level thread $\tau_i \in \Gamma_{mw}$ is bounded by*

$$si_i(m_z, \Delta) \triangleq \max(0, \eta_{z,i}(\Delta + \overline{R}(m_z) - \epsilon) - 1). \tag{5}$$

Proof. The lemma follows from Lemma 5 by noting that only pending instances of m_z can cause self-interference in the interval $[\bar{t}, \bar{t} + \Delta)$, with $\Delta > 0$, excluding the message instance under analysis, and noting that the number of self-interfering instances cannot be negative. ◀

The precision of the self-interference bound can be further tightened at the cost of testing a search space of multiple message release times in a busy window [17], thus complicating the analysis and requiring additional running time. To keep the analysis simple, this potential improvement is left as future work.

`HIGH_PRIORITY` policy. Under this policy, each instance of m_z under analysis can be delayed by: **(i)** low-priority messages causing delay due to non-preemptive message handling (rule **R5**), **(ii)** equal-priority messages enqueued before and thus being prioritized by the FIFO tie-break, and **(iii)** higher-priority messages. Let $I^{\mathrm{lp}}_{i,z}(\Delta)$, $I^{\mathrm{ep}}_{i,z}(\Delta)$, and $I^{\mathrm{hp}}_{i,z}(\Delta)$ be the interference bounds for (i), (ii), and (iii), respectively, so that $I^{\mathrm{intra}}_{i,z}(\Delta) \triangleq I^{\mathrm{lp}}_{i,z}(\Delta) + I^{\mathrm{ep}}_{i,z}(\Delta) + I^{\mathrm{hp}}_{i,z}(\Delta)$. We begin by considering equal-priority messages in Lemma 10.

▶ **Lemma 10.** *All the delays that may contribute to the intra-thread message interference of an instance of m_z that is pending in a middleware-level thread $\tau^t_i \in \Gamma_{mw}$, during any interval of length Δ and due to messages with same priority, are contained into the multiset*[2]

$$\mathcal{D}^{ep}_i(\Delta) = \biguplus_{m_r \in ep_i(m_z)} \{\delta^t_i(m_r)\} \otimes \overline{\eta}_{r,i}(\Delta), \text{ with } t \in \{f, l\} \tag{6}$$

where

$$\overline{\eta}_{r,i}(\Delta) \triangleq \begin{cases} si_i(m_z, \Delta) & \text{if } z = r, \\ \eta_{r,i}(\Delta + \overline{R}^t_i(m_r) - \epsilon) & \text{otherwise,} \end{cases} \tag{7}$$

where $\delta^t_i(m_r)$ is given by Lemma 7 and $\overline{R}^t_i(m_r)$ is a response-time bound for m_r in τ^t_i.

Proof. First, note that delays due to intra-thread message interference to an instance of message m_z from messages with the same priority in a middleware-level thread τ^{t}_i are due to other messages $m_r \in ep_i(m_z)$ in the queue of the same thread. By Lemma 7, each message contributes with a delay of at most $\delta^t_i(m_r)$. By Lemma 9, m_z can contribute with up to $si_i(m_z, \Delta)$ interfering message instances. The lemma follows by noting that other messages can interfere only if they are pending in the same middleware-level thread, with up to $\eta_{r,i}(\Delta + \overline{R}^t_i(m_r) - \epsilon)$ due to Lemma 5. ◀

Hereafter, the notation $\Sigma(x, M)$ denotes the sum of the x *largest* elements in a multiset M. If M contains less than x elements, all elements in M are summed.

▶ **Lemma 11.** *Let j be the priority of message m_z. Consider an instance of m_z that is pending in a middleware-level thread $\tau^t_i \in \Gamma_{mw}$ that uses the `HIGH_PRIORITY` policy and an arbitrary time window of length Δ. It holds*

$$I^{ep}_{i,z}(\Delta) \triangleq \Sigma(M^{HP,j}_i - 1, \mathcal{D}^{ep}_i(\Delta)). \tag{8}$$

[2] The operator $\uplus$ denotes the union of multisets, e.g., $\{3, 3\} \uplus \{6, 2\} = \{3, 3, 6, 2\}$, and the product operator $\otimes$ multiplies the number of instances of each element in the multiset, e.g., $\{1, 4\} \otimes 2 = \{1, 1, 4, 4\}$.

Proof. By definition, the j-th priority queue of τ_i^{t} has size $M_i^{\mathtt{HP},j}$. Therefore, at most $M_i^{\mathtt{HP},j}-1$ message instances with same priority can interfere with the one under analysis. By Lemma 10, the delays of message instances with same priority that may contribute to the intra-thread interference of m_z are contained into the multiset $\mathcal{D}_i^{\mathrm{ep}}(\Delta)$. Hence $\Sigma(M_i^{\mathtt{HP},j}-1, \mathcal{D}_i^{\mathrm{ep}}(\Delta))$ bounds the intra-thread interference generated by messages with the same priority of m_z. ◀

Next, we provide a bound to the intra-thread interference due to messages with lower priority, which occurs because each message is handled in a non-preemptive manner [23, 43] locally to each thread.

▶ **Lemma 12.** *Consider an instance of a message m_z in a middleware-level thread using the `HIGH_PRIORITY` policy $\tau_i^t \in \Gamma_{mw}$ and a time window of length Δ. It holds*

$$I_{i,z}^{lp}(\Delta) \triangleq \max_{m_r \in lp_i(m_z)} \delta_i^t(m_r), \ \ with\ t = f. \tag{9}$$

Proof. Low-priority messages can contribute with at most *one* instance due to the non-preemptive handling of messages (rule **R5**). The corresponding delay can be at most equal to the longest delay $\delta_i^t(m_r)$, yielding the bound $I_{i,z}^{\mathrm{lp}}(\Delta)$. ◀

Differently from equal-priority messages (Lemma 11), the bound for higher-priority messages cannot rely on message queue sizes. This is because the message under analysis is placed in a different queue: thus, the bound can only leverage the message arrival curves on the middleware-level thread under consideration. A bound on the intra-thread interference due to higher-priority messages is reported in Lemma 13.

▶ **Lemma 13.** *Consider an instance of a message m_z in a middleware-level thread using the `HIGH_PRIORITY` policy $\tau_i^t \in \Gamma_{mw}$ and a time window of length Δ. Let $\overline{R}_i^t(m_r)$ be a response-time bound for a higher priority message m_r in τ_i^t, it holds*

$$I_{i,z}^{hp}(\Delta) \triangleq \sum_{m_r \in hp_i(m_z)} \eta_{r,i}(\Delta + \overline{R}_i^t(m_r) - \epsilon) \cdot \delta_i^t(m_r), \ \ with\ t = f. \tag{10}$$

Proof. Higher-priority messages can interfere with all instances that are pending in an arbitrary interval $[\bar{t}, \bar{t} + \Delta)$. By Lemma 5, the number of such instances is bounded by $\eta_{r,i}(\Delta + \overline{R}_i^t(m_r) - \epsilon)$, each one delaying for up to $\delta_i^t(m_r)$. ◀

With Lemmas 11, 12, and 13 in place, we have all the interference components to instantiate a response-time bound under the `HIGH_PRIORITY` policy, which we present later in Theorem 15.

`FIFO` policy. Under the `FIFO` policy, all messages of a middleware-level thread τ_i^{t} are handled in a single queue of size $M_i^{\mathtt{F}}$. Since the tie-break policy for messages with equal priority under `HIGH_PRIORITY` is FIFO too, intra-thread message interference $I_{i,z}^{\mathrm{intra}}(\Delta)$ can be bounded as $I_{i,z}^{\mathrm{ep}}(\Delta)$ in Lemma 11, but considering $M_i^{\mathtt{F}}$ in place of $M_i^{\mathtt{HP},j}$, and using set τ_i^{t} in the union of Lemma 10 instead of $ep_i(m_z)$.

Response-time bound. We provide the response-time bound by proceeding in two steps. First, we bound the start time of an arbitrary message instance m_z' under analysis, i.e., the time in which the middleware-level thread starts serving non-preemptively m_z', locally to the middleware-level thread under consideration. However, note that m_z' can still suffer thread-level and inter-thread message interference due to higher-priority threads. Later, we bound the response time by leveraging the start-time bound and the fact that once m_z' started being served, it cannot experience intra-thread message interference.

Theorem 14 bounds the start-time of an arbitrary message instance m'_z in a middleware-level thread τ_i.

▶ **Theorem 14.** *Consider an arbitrary instance m'_z of message m_z running in a middleware-level thread τ_i released at a time A. If S^* is the least positive solution (if any) of the following inequality*

$$sbf_k(S^*) \geq \epsilon + I^{intra}_{i,z}(S^*) + I^{thread}_{i,z}(S^*) + I^{inter}_{i,z}(S^*), \tag{11}$$

then m'_z starts being processed in the middleware-level thread no later than time $A + S^*$.

Proof. By Lemmas 6 and 8, $I^{\text{thread}}_{i,z}(S^*)$ and $I^{\text{inter}}_{i,z}(S^*)$ bound the thread-level and inter-thread message interference, respectively. The intra-thread message interference is bounded by $I^{\text{intra}}_{i,z}(S^*)$ due to Lemmas 11, 12 and 13, if the middleware-level thread adopts the `HIGH_PRIORITY` policy, or Lemma 11 (slightly modified as suggested above) if the `FIFO` policy is used. If S^* satisfies Equation (11), then the service time $sbf_k(S^*)$ supplied by core c_k in any interval of length S^* is enough to satisfy the computational demand of the whole interference to m'_z in the same interval. Therefore, being the middleware-level thread work-conserving (rule **R8**), m'_z starts being served in the middleware-level thread no later than time $A + S^*$ and the theorem follows. ◀

Finally, Theorem 15 provides a response-time bound R^*.

▶ **Theorem 15.** *Consider an arbitrary instance m'_z of message m_z processed by a middleware-level thread τ^t_i released at a time A. If S^* is defined as in Theorem 14 and R^* is the least positive solution (if any) of the following inequality*

$$sbf_k(R^*) \geq \epsilon + I^{intra}_{i,z}(S^*) + I^{thread}_{i,z}(R^*) + I^{inter}_{i,z}(R^*) + \delta^t_i(m_z), \tag{12}$$

then m'_z completes no later than $A + R^*$.

Proof. By Theorem 14, m'_z starts being served no later than time $A + S^*$. After that, due to rule **R5**, it starts being processed non-preemptively in the middleware-level thread and it does not suffer intra-thread interference anymore. Hence $I^{\text{intra}}_{i,z}(S^*)$ bounds the overall intra-thread interference suffered by m'_z in $[A, A + R^*)$. Inter-thread and thread-level interference in the same interval are bounded by $I^{\text{inter}}_{i,z}(R^*)$ and $I^{\text{thread}}_{i,z}(R^*)$, respectively. If R^* satisfies Equation (12), then the service time $sbf_k(R^*)$ supplied by core c_k in any interval of length R^* is enough to satisfy the computational demand of the whole interference suffered by m'_z, plus the time $\delta^t_i(m_z)$ to process m'_z itself. Hence, the theorem follows. ◀

Response-time bounds for the flow-controller and listener threads required to compute the DDL (see Equation (1), terms $R^f_i(m_z)$ and $R^l_j(m_z)$) can be computed with the results presented in this section, considering either the `HIGH_PRIORITY` or `FIFO` policy for the flow controller, and the `FIFO` policy for the listener.

5.2 Arrival-curve propagation

The DDL bound derived in the previous section is based on the knowledge of the arrival curves of the various threads in the system. Using the standard arrival curve propagation approach of CPA [32], they can be derived from the externally-provided arrival curves $\eta^p_i(\Delta)$ of publisher threads $\tau^{\text{p}}_i \in \Gamma_{\text{p}}$, response-time bounds, and network propagation delay, if any, in the path from the source to the destination.

As discussed in Section 4, a message $m_z(\tau_i^{\mathrm{p}}, \theta_j)$ is identified by its publisher τ_i^{p} and topic θ_j. Furthermore, the per-message arrival curve depends on the number of messages w_i^j published by each instance of the publisher to θ_j. The arrival curve propagation process is shown in Figure 4. The first step is to compute the arrival curve of a message m_z in the flow-controller thread τ_x^{f} starting from the arrival curve of its publisher and the number of message instances sent in each publisher instance to θ_j. This can be computed as:

$$\eta_{z,x}^{\mathrm{f}}(\Delta) = \eta_i^{p}(\Delta + \overline{R}(\tau_i^{\mathrm{p}}) - \epsilon) \cdot w_i^j, \tag{13}$$

where $\overline{R}(\tau_i^{\mathrm{p}})$ is a response-time bound for the publisher thread, which can be derived with standard methods for response-time analysis under preemptive fixed-priority scheduling [38]. As shown in Figure 4, the message then passes through the network, with a delay $\delta^{\mathrm{net}}(m_z)$, and it is received by the listener thread τ_y^{l}. The arrival curve of the message within the listener thread is hence computed as:

$$\eta_{z,y}^{\mathrm{l}}(\Delta) = \eta_{z,x}^{\mathrm{f}}(\Delta + \overline{R}_x^{\mathrm{f}}(m_z) + \delta^{\mathrm{net}}(m_z) - \epsilon). \tag{14}$$

Finally, the arrival curve of the subscriber thread is obtained with an OR-activation semantics [33, 32] by summing all the activations due to all messages $m_z \in \mathcal{M}(\theta_j)$, from the topics $\theta_j \in \Theta(\tau_q^{\mathrm{s}})$ to which the thread τ_q^{s} subscribes to, i.e.,

$$\eta_q^{s}(\Delta) = \sum_{\theta_j \in \Theta(\tau_q^{\mathrm{s}})} \sum_{m_z \in \mathcal{M}(\theta_j)} \eta_{z,y}^{\mathrm{l}}(\Delta + \overline{R}_y^{\mathrm{l}}(m_z) - \epsilon). \tag{15}$$

5.3 Analysis summary and its applicability

Analysis summary. Algorithm 1 summarizes the analysis proposed in this paper for the purpose of computing the DDL according to Equation (1). The pseudo-code relies on global variables to store response-time bounds and their candidates (line 2). Then, function `COMPUTE_RT_BOUNDS()` needs to be called to populate the global variables $R_x^{\mathrm{f}}(m_z)$ and $R_y^{\mathrm{l}}(m_z)$ with the response-time bounds of each message m_z in the corresponding flow-controller and listener threads, respectively.

The calculation leverages functions `ResponseTimeBound_FlowController()` (line 17) and `ResponseTimeBound_Listener()` (line 27) that properly instantiate $I_{i,z}^{\mathrm{intra}}(\Delta)$ as discussed in Section 5.1, while $I_{i,z}^{\mathrm{inter}}(\Delta)$ and $I_{i,z}^{\mathrm{thread}}(\Delta)$ are defined as in Lemmas 6 and 8 in both the cases. As discussed in Section 5.1, the computation of the response-time bounds $R_x^{\mathrm{f}}(m_z)$, $R_y^{\mathrm{l}}(m_z)$ (functions `ResponseTimeBound_FlowController()` and `ResponseTimeBound_Listener()`) cyclically depends on the existence of pre-existing response-time bounds $\overline{R}_x^{\mathrm{f}}(m_z)$, $\overline{R}_y^{\mathrm{l}}(m_z)$. The dependency is broken by initializing the bounds to zero (line 6) and performing an outer loop (lines 7-15) until a global fixed-point is reached (i.e., $R_x^{\mathrm{f}}(m_z) \neq \overline{R}_x^{\mathrm{f}}(m_z)$ and $R_y^{\mathrm{l}}(m_z) \neq \overline{R}_y^{\mathrm{l}}(m_z)$ for each message m_z) and performing arrival curve propagation (see Section 5.2) inside the loop. Convergence is guaranteed by the fact that response-time estimates (variables $\overline{R}_y^{\mathrm{l}}(m_z)$ and $\overline{R}_x^{\mathrm{f}}(m_z)$) never decreases. Response-time bounds for publisher and subscriber threads also need to be computed (according to standard fixed-priority scheduling, line 13) inside the loop, updating global response time variables (as those initialized in line 6) to be used in the arrival curve propagation process (e.g., Equation (13), not detailed in the pseudo-code for brevity). When `COMPUTE_RT_BOUNDS()` completes, the global variables are configured to the correct response-time bound values, and the function `DDL`(m_z, $\delta^{\mathrm{net}}(m_z)$), at line 33 of the Algorithm 1, can be used to compute the DDL of a given message m_z according to Equation (1) by providing the network delay $\delta^{\mathrm{net}}(m_z)$ as an input parameter.

Algorithm 1 Pseudo-code of the DDL analysis.

1: **global variables** $\forall\, \theta_j \in \Theta, m_z \in \mathcal{M}(\theta_j)$, define ▷ for each message in the system
2: $R^{\mathrm{f}}_x(m_z),\ R^{\mathrm{l}}_y(m_z),\ \overline{R}^{\mathrm{f}}_x(m_z),\ \overline{R}^{\mathrm{l}}_y(m_z)$ to store response-time bounds and the corresponding candidates
3:
4: **function** COMPUTE_RT_BOUNDS( )
5: $\quad\forall\, \theta_j \in \Theta, m_z \in \mathcal{M}(\theta_j)$: ▷ for each message in the system:
6: $\qquad R^{\mathrm{f}}_x(m_z) \leftarrow 0, R^{\mathrm{l}}_y(m_z) \leftarrow 0, \overline{R}^{\mathrm{f}}_x(m_z) \leftarrow 0, \overline{R}^{\mathrm{l}}_y(m_z) \leftarrow 0$
7: $\quad$**do**
8: $\qquad\forall\, \theta_j \in \Theta, m_z \in \mathcal{M}(\theta_j)$: ▷ for each message in the system
9: $\qquad\quad\overline{R}^{\mathrm{f}}_x(m_z) \leftarrow R^{\mathrm{f}}_x(m_z), \overline{R}^{\mathrm{l}}_y(m_z) \leftarrow R^{\mathrm{l}}_y(m_z)$
10: $\qquad\quad$FC_SCHED_POL $\leftarrow$ scheduling policy of the flow-controller thread handling m_z
11: $\qquad\quad R^{\mathrm{f}}_x(m_z) \leftarrow$ RESPONSETIMEBOUND_FLOWCONTROLLER(m_z, FC_SCHED_POL)
12: $\qquad\quad R^{\mathrm{l}}_y(m_z) \leftarrow$ RESPONSETIMEBOUND_LISTENER(m_z)
13: $\qquad\quad$`compute response time bounds for application threads`
14: $\qquad\quad$`perform arrival curve propagation` ▷ see Section 5.2
15: $\quad$**while** `no more response-time bounds updates` $\forall\ m_z$
16:
17: **function** RESPONSETIMEBOUND_FLOWCONTROLLER(m_z, FC_SCHED_POL)
18: $\quad$**switch** FC_SCHED_POL **do**
19: $\qquad$**case** HIGH_PRIORITY:
20: $\qquad\quad$Set $I^{\mathrm{intra}}_{x,z}(\Delta) \leftarrow I^{\mathrm{ep}}_{x,z}(\Delta) + I^{\mathrm{lp}}_{x,z}(\Delta) + I^{\mathrm{hp}}_{x,z}(\Delta)$
21: ▷ where $I^{\mathrm{ep}}_{x,z}(\Delta), I^{\mathrm{lp}}_{x,z}(\Delta), I^{\mathrm{hp}}_{x,z}(\Delta)$ are bounded as in Lemmas 11, 12, and 13
22: $\qquad$**case** FIFO:
23: $\qquad\quad$Set $I^{\mathrm{intra}}_{x,z}(\Delta) = I^{\mathrm{ep}}_{x,z}(\Delta)$ ▷ using Lemma 11
24: $\qquad\quad$▷ with $M^{\mathtt{F}}_x$ in place of $M^{\mathtt{HP},j}_x$, and with τ^{f}_x in the union of Lemma 11 in place of $ep_x(m_z)$
25: $\quad$Compute S^* using Theorem 14, compute R^* using Theorem 15
26:
27: **function** RESPONSETIMEBOUND_LISTENER(m_z)
28: $\quad$Set $I^{\mathrm{intra}}_{y,z}(\Delta) = I^{\mathrm{ep}}_{y,z}(\Delta)$ ▷ using Lemma 11
29: $\quad$▷ with $M^{\mathtt{F}}_y$ in place of $M^{\mathtt{HP},j}_y$, and with τ^{l}_y in the union of Lemma 11 in place of $ep_y(m_z)$
30: $\quad$Compute S^* using Theorem 14, compute R^* using Theorem 15
31: $\quad$**return** R^*
32:
33: **function** DDL(m_z, $\delta^{\mathrm{net}}(m_z)$) ▷ to be called after COMPUTE_RT_BOUNDS()
34: $\quad$**return** $R^f_x(m_z) + \delta^{\mathrm{net}}(m_z) + R^l_y(m_z)$

Applicability. The analysis strategy we proposed makes our method applicable to several practically useful scenarios.

Linux – SCHED_FIFO. A natural fit for our method is to analyze the timing behavior of FastDDS-based applications running on the SCHED_FIFO scheduling class of Linux, which provides a fixed-priority scheduler fulfilling the assumptions of our model. In this case, each core provides the full supply to the scheduled applications, since no reservation-based mechanism is provided. Hence, $sbf_k(\Delta) = \Delta, \forall k$.

Linux – SCHED_DEADLINE and QNX APS. Thanks to the supply-bound function abstraction, our analysis is suitable also for being applied to systems using the SCHED_DEADLINE scheduler of Linux, a reservation-based scheduler. Under SCHED_DEADLINE, each thread can be individually isolated from a temporal perspective by associating it with a budget and period pair [1, 11]. The corresponding definition for $sbf_k(\Delta)$ is available in the literature [10, 15, 40]. Thanks to the temporal isolation, $I^{\mathrm{thread}}_{i,z}(\Delta) = 0$ and $I^{\mathrm{inter}}_{i,z}(\Delta) = 0$. Furthermore, our analysis also generalizes to the QNX APS reservation-based scheduler [6, 22], by considering only threads allocated to the same APS partition of the thread under analysis when deriving $I^{\mathrm{thread}}_{i,z}(\Delta)$ and $I^{\mathrm{inter}}_{i,z}(\Delta)$.

Processing chains and ROS. Thanks to the compositionality of our approach, our analysis can be used to study the end-to-end response-time of data-driven distributed applications [7, 29, 64, 65]. Furthermore, our approach makes few assumptions on how application-level threads are scheduled, making it easily extensible to work with methods to study the response-time of processing chains using ROS 2 [13, 17], which leverages the DDS as a lower-layer middleware and hence it is a practical use case for the DDS. However, none of the analyses for ROS 2 currently available in the literature provides a method to bound the DDS-related delay. For example, the analysis in [17] for ROS 2 models the delay due to the DDS as a single parameter and suggests estimating it empirically. The analysis of this paper (Equation (1)) presents the first analytical solution to provide a theoretically-sound bound on the single-parameter DDS-related delay of [17], which can therefore be used as a complement of previous work on ROS 2 processing chains.

6 Evaluation

First, we evaluated our analysis on a case study based on the WATERS 2019 Challenge by Bosch [30], which consists of a representative autonomous driving application. Then, we report on a comparison between the analysis results and actual measurements on a real platform running FastDDS and a relatively simple application.

6.1 WATERS 2019 Challenge case study

The model for the Challenge provides parameters such as periods, worst-case execution times, and data exchanged among threads (i.e., shared labels) for nine threads. While the challenge model was not designed to work with the DDS, the target application is a good fit to work with a pub/sub paradigm. Figure 5 illustrates how the Challenge application was adapted to work with the DDS. Shared labels were modeled as topics (represented by ellipses) having the same payload size as labels, with the following meaning: when a task writes on a label is *publishing* messages on that label, i.e., topic, and a task reading on a label is *receiving* messages from that topic. Threads performing only reads on labels were considered as subscribers with data-driven activation based on the subscription to the corresponding topics (e.g., see the `DASM` task). For threads that are both reading and writing labels (`LOC`, `EKF`, and `PLAN`), two sub-threads were identified, representing the subscriber and a publisher (denoted in the figure with the prefixes "S_" and "P_" respectively). In this way, the original periods of the Challenge were preserved for the publishers. Given the original WCET e_i of the challenge model, individual WCETs of the corresponding publisher and subscriber thread were derived as $e_i^{\mathrm{pub}} = e_i \cdot \alpha$ and $e_i^{\mathrm{sub}} = e_i \cdot (1 - \alpha)$, with $\alpha \in [0, 1]$. The other threads were left unaltered. The parameter α is introduced to split the WATERS Challenge's threads into publisher-subscriber pairs and can be used to regulate how much computation time to assign to publish and subscribe parts of each thread.

Message delays in the flow-controller and listener. To run the analysis, it is necessary to know the worst-case delays $\delta^{\mathrm{f}}(m_z)$ and $\delta^{\mathrm{l}}(m_z)$ to process a message in the flow-controller and listener thread, respectively. To estimate such parameters, we developed a FastDDS application consisting of a publisher, its flow-controller, a subscriber, and its listener. The two application-level threads communicate through a single topic, using UDP through the loopback interface. The application was executed on an 8-core *Dell Optiplex 7070* machine running Ubuntu 20.04. The flow-controller and listener threads were mapped to two different cores with the highest priority. Each message was processed 50000 times by each middleware

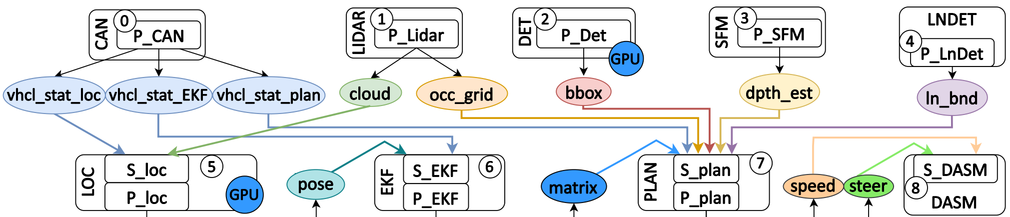


Figure 5 WATERS 2019 Challenge adapted to use a pub/sub paradigm.

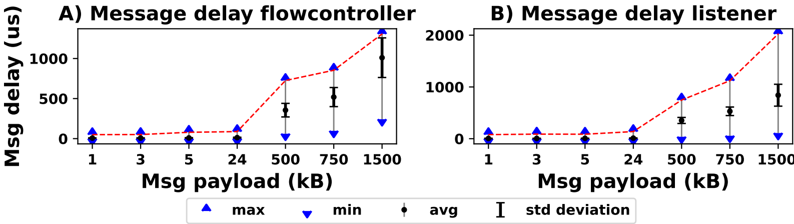


Figure 6 Estimation of the message delay.

thread. Middleware threads were configured to collect data about the execution time of each processed message. Figure 6 illustrates the results. For each payload size (x-axis) used in the WATERS 2019 Challenge, the graph shows the minimum, maximum, average, and standard deviation of message delays (y-axis). Both graphs show the same trend. For payload sizes $\leq$ 24kB, the execution time trend is fairly constant and does not exceed 125μs and 200μs for flow-controller and listener messages, respectively. As payload size exceeds loopback MTU (64kB), the fragmentation and reassembly of UDP packets was found to cause relevant overheads, affecting the processing time of the messages.

Analysis results. Next, we discuss the results of the analysis of this case study, which was implemented using the *pyCPA* framework [25]. The priorities of publishers were set according to rate-monotonic. In this case study, at most one thread publishes on each topic. Therefore, we assign to each topic the same priority of the corresponding publisher (which is then inherited by each message sent through the topic). Whenever a publisher publishes on multiple topics, the topic (i.e. message) associated with a smaller payload size is assigned a higher priority. Subscribers `S_loc`, `S_EKF`, `S_plan` inherited the priority of the corresponding publishers, while `S_DASM` priority was set to the highest priority in the system because of providing the application's output. We set $\delta^{\text{net}}(m) = 0$ since we studied threads running on the same computing node. We evaluated the analysis on a vast range of configurations, where multiple design-level parameters were varied: **(i)** the task-to-core assignment, **(ii)** the priorities of application-level and middleware-level threads, **(iii)** the number of flow-controllers and the topics-to-flow-controller assignments, **(iv)** the message priorities in the flow-controller threads.

Among them, we selected four relevant system configurations (with $\alpha = 0.95$) and we discuss their trade-offs:

(A) A configuration in which each publisher has its own flow-controller, and all the threads are exclusively assigned to a core, and no thread-level interference can occur;

(B) A configuration with *eight* flow-controllers, where flow-controllers and listeners are in the same core of their publishers and subscribers, respectively;

(C) A configuration with *two* flow-controllers, in which one flow-controller manages messages with lower payload ({1, 3, 5, 24} kB), while the other handles messages with higher payload ({500, 750, 1500} kB).

(D) A configuration with *one* flow-controller handling all messages, allocated on a dedicated core.

In (C) and (D), listeners are allocated to the same core of corresponding subscribers. For each configuration, Figure 7 shows the DDL (y-axis) for each message (x-axis).

Configuration A. Figure 7 (A) shows the results of the configuration, where threads execute exclusively on a core. Thus, a message can only suffer interference due to messages within the same flow-controller thread, which occurs for publishers sending multiple different messages. We considered two scenarios for the HP and F policies of the flow-controller. In this configuration, both sending policies found not to have any significant effects on the DDL. For the vchl_stat_plan message, the F policy generates a slightly higher DDL than the HP, due to the fact that, under HP, this message has the highest priority within the flow-controller.

Configuration B. In this configuration, we evaluated the effects of the relative priority between a publisher and its flow-controller thread. To this end, we considered four different cases in Figure 7 (B): **(i)** the HP policy of the flow-controller (settings 1-HP and 2-HP), **(ii)** the FIFO policy of the flow-controller (1-F and 2-F), **(iii)** flow-controllers with higher priorities than publishers (1-HP and 1-F), **(iv)** flow-controllers with lower priorities than publishers (2-HP and 2-F). When flow-controllers have a high priority, results show the same behavior and same values of Figure 7 (A). When we consider lower-priority flow-controllers in the cases 2-HP and 2-F, the DDL increases for each message that can suffer from the publisher execution interference. The longest DDLs ($16.5ms$ for 2-HP and $17ms$ for 2-F) are observed for the occ_grid message, with a 3x latency increment w.r.t. (A).

Configurations C and D. Figures 7 (C) and 7 (D) show the results of the configurations with two and one flow-controller, respectively. Scenarios 1-HP-L>S and 1-F-L>S consider the HP and FIFO policies of the flow-controllers when listeners have higher priorities than their subscribers. Scenarios 2-HP-L<S and 2-F-L<S consider listeners with lower priorities than their subscribers. Considering 1-HP-L>S and 2-HP-L<S, the configuration in inset (D) leads low-priority messages (e.g., ln_bnd, pose, dpth_est) to suffer from a significant interference from the processing of high-priority messages compared to configuration in inset (C). Moreover, due to the non-preemptiveness of the sending operation, for each message, the DDL of the configuration (D) always accounts the time needed to process the highest-payload message (cloud message, with size $1500kB$). Differently, in configuration (C), lower-payload messages just suffer at most from of non-preemptiveness processing delay due to messages with size $24kB$. This is an advantage due to having two flow-controllers that manage different data flows. In both configurations, lower-priority listeners cause larger DDL for all messages due to subscriber-related interference.

6.2 Comparing analysis bounds with measured DDLs

Next, we compare the results of our analysis with empirical measurements collected from a real FastDDS application executed on the previously mentioned *Optiplex 7070* platform, running Ubuntu 20.04. The considered application consists of a publisher τ_1^{p}, with its flow-controller τ_2^{f}, and a subscriber τ_4^{s}, with its listener τ_3^{l}, exchanging data over three different topics (θ_1, θ_2, θ_3). Messages (named $m(\theta_1)$, $m(\theta_2)$, $m(\theta_3)$) have the same payload size ($1kB$). Message delays are set according to the measurements reported in Figure 6 for $1kB$ payloads. Threads τ_1^{p} and τ_2^{f} are allocated to the same core c_1 and assigned to the two

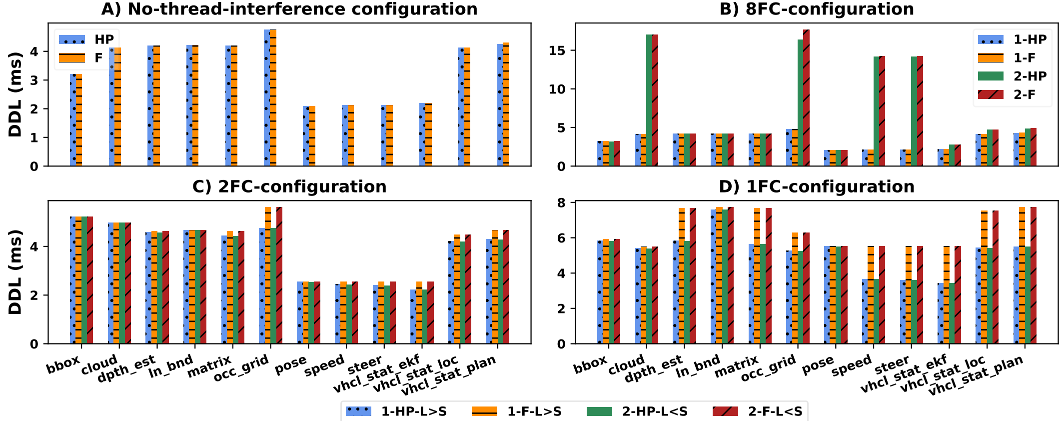


Figure 7 Experimental results under four representative configurations of the case study.

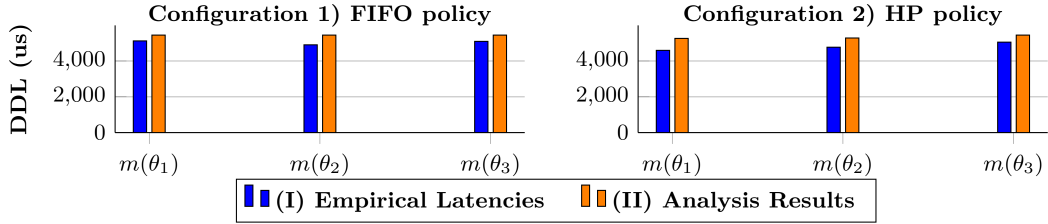


Figure 8 Results from FastDDS app and analysis related to Configurations 1) and 2).

highest priorities, with τ_2^{f} having higher priority than τ_1^{p}. τ_4^{s} and τ_3^{l} are mapped, respectively, to cores c_3 and c_2, and set with the highest priority in their respective core. τ_1^{p} is configured as a periodic thread with period $2ms$. Therefore, it is characterized by an arrival curve $\eta_1^p(\Delta) = \lceil \frac{\Delta}{T} \rceil$. On the Optiplex platform, middleware threads of the FastDDS application are configured to measure the DDL for each message over 50000 samples. We tested two different configurations: **Conf. 1)**, in which messages in the flow-controller are scheduled under `FIFO` policy; **Conf. 2)** in which the flow-controller schedules messages under the `HIGH_PRIORITY` policy and topics are assigned to a unique priority such that lower topic subscript identifiers indicate higher priority values. Figure 8 shows the DDL (y-axis) for each message (x-axis) obtained through measurements and by the analysis for both configurations.

Conf. 1). Under `FIFO` policy, each flow-controller message can interfere with others. Our analysis accordingly computes the same DDL bound ($5446us$) for all of them. Comparing the DDL bound with the measured values on the Optiplex platform we can observe that values do not exceed the DDL bound found by the analysis, corroborating its validity.

Conf. 2). Under `HP` policy, both the analysis and the measurements on the Optiplex platform show DDL values that depend upon the message priority. Moreover, we can observe that for the lowest priority message (i.e., $m(\theta_3)$), the DDL bound of the analysis equals the DDL bound found in the **Conf. 1)**. Also in this case, empirical DDL values do not exceed the DDL bounds found with the analysis.

In Table 2, we reported the relative distance, in percentage, between the DDL bound provided by the analysis and the measured value, showing that the DDL bounds found are tight for the considered application.

Table 2 Table of percentage relative distances: measurements vs. analysis bound.

Message	Configuration (1) FIFO	Configuration (2) HIGH_PRIORITY
$m(\theta_1)$	6.3 %	14.5 %
$m(\theta_2)$	11.1 %	10.7 %
$m(\theta_3)$	6.9 %	7.7 %

7 Related Work

The literature regarding the real-time aspects of DDS is quite limited. To the best of our knowledge, this is the first attempt to model the DDS from a real-time perspective and provide real-time analysis for DDS-based communications.

Most of the previous research on DDS focused on empirical performance measurement. For instance, Bellavista et al. [8] compared the DDS implementation OpenSlice with Connext-DDS by Real-Time Innovations [54]. Krinkin et al. [36] proposed a framework to assess the effectiveness of various DDS implementations in terms of message transport latency and throughput. Other works attempted to suggest potential improvements for DDS implementations. For example, Choi et al. [19] studied a real-time DDS setup over specialized packet-switching ASICs to enable Software Defined Networking (SDN). Peeck et al. [50] presented a UDP-based protocol for effective error correction with integrity guarantees that considers the DDS as the middleware for data-centric embedded systems. Agarwal et al. [2] proposed the integration of a DDS implementation with a TSN protocol for real-time data transfers. Stevanato et al. [62] proposed a reference architecture for implementing virtualized DDS communications in a hypervisor-based multi-domain system. Finally, Scordino et al. [56] implemented in hardware some DDS functionalities. Other works considered other middlewares, e.g., OpenMP [58, 63], ROS 2 [3, 17, 20, 64, 66], the ROS-based framework Apex.OS [51], and RT-Appia [53]. However, none of the works addressing the analysis of ROS 2 provides analytical methods to bound the data-delivery latency of the DDS. Empirical evaluations of ROS 2 over different DDS implementations have been carried out by Maruyama et al. [41] and Kronauer et al. [37], providing guidelines on designing ROS 2 applications to minimize latencies.

8 Conclusion and Future Work

In this paper, we derived a compositional model for studying the timing of the DDS standard and we instantiated it for FastDDS. We inspected the FastDDS documentation and source code to build an accurate model capable of capturing the FastDDS-specific timing-related effects, and we corroborated our findings by running validation experiments on an actual FastDDS system. Building on the model, we derived an analysis to bound the data-delivery latency of messages. We evaluated our analysis based on the WATERS 2019 Industrial Challenge showing how thanks to our analysis, it becomes easily possible to compare a vast range of configurations without the need to deploy them on a real system.

Furthermore, we compared analysis results with actual measurements on a real platform running FastDDS and a relatively simple application, showing the tightness of our analysis for the specific use case. The proposed analysis will enable system designers to configure DDS-based systems, guiding choices such as thread-to-core allocation, priorities, and reservation budgets in a timing-constraints-driven perspective. It will set the foundation to account for DDS-related delays in analysis-driven orchestration algorithms, which will

be the subject of future work. Other directions for future research include combining this analysis with other analysis techniques [47] to improve the analysis precision, the holistic consideration of scheduling effects due to the DDS with OS overheads [24], I/O-related contention delays [59, 71], and network delays [14, 18, 44, 52] with special emphasis on Time-Sensitive Networking [49].

References

1 L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, 1998.

2 Tanushree Agarwal, Payam Niknejad, Mohammadreza Barzegaran, and Luigi Vanfretti. Multi-level time-sensitive networking (TSN) Using the Data Distribution Services (DDS) for synchronized three-phase measurement data transfer. *IEEE Access*, PP:1–1, September 2019.

3 Abdullah Al Arafat, Sudharsan Vaidhun, Kurt M Wilson, Jinghao Sun, and Zhishan Guo. Response time analysis for dynamic priority scheduling in ROS2. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 301–306, 2022.

4 AUTOSAR. Specification of Communication Management, 2020. URL: `https://www.autosar.org/fileadmin/standards/R22-11/AP/AUTOSAR_SWS_CommunicationManagement.pdf`.

5 Daniel Balouek-Thomert, Ali Reza Zamani Eduard Gibert Renart, and Manish Parashar Anthony Simonet. Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows. *The International Journal of High Performance Computing Applications*, 33(6):1159–1174, 2019.

6 Matthias Becker, Dakshina Dasari, and Daniel Casini. On the QNX IPC: Assessing predictability for local and distributed real-time systems. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2023.

7 Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. End-to-end timing analysis of cause-effect chains in automotive embedded systems. *Journal of Systems Architecture*, 80:104–113, 2017.

8 P. Bellavista, A. Corradi, L. Foschini, and A. Pernafini. Data distribution service (DDS): A performance comparison of OpenSplice and RTI implementations. In *2013 IEEE Symposium on Computers and Communications (ISCC)*, pages 000377–000383, Los Alamitos, CA, USA, July 2013. IEEE Computer Society.

9 Luca Belluardo, Andrea Stevanato, Daniel Casini, Giorgiomaria Cicero, Alessandro Biondi, and Giorgio Buttazzo. A multi-domain software architecture for safe and secure autonomous driving. In *2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 73–82, 2021.

10 Alessandro Biondi, Giorgio C. Buttazzo, and Marko Bertogna. Schedulability analysis of hierarchical real-time systems under shared resources. *IEEE Transactions on Computers*, 65(5):1593–1605, 2016.

11 Alessandro Biondi, Alessandra Melani, and Marko Bertogna. Hard constant bandwidth server: Comprehensive formulation and critical scenarios. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, pages 29–37, 2014.

12 Tobias Blass, Arne Hamann, Ralph Lange, Dirk Ziegenbein, and Björn B. Brandenburg. Automatic latency management for ROS 2: Benefits, challenges, and open problems. In *Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021.

13 Tobias Blaß, Daniel Casini, Sergey Bozhko, and Björn B. Brandenburg. A ROS 2 response-time analysis exploiting starvation freedom and execution-time variance. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 41–53, 2021.

14 C. Blumschein, I. Behnke, L. Thamsen, and O. Kao. Differentiating Network Flows for priority-aware scheduling of incoming packets in real-time IoT systems. In *2022 IEEE 25th*

International Symposium On Real-Time Distributed Computing (ISORC), pages 1–8, Los Alamitos, CA, USA, May 2022. IEEE Computer Society.

15 Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Publishing Company, Incorporated, 3rd edition, 2011.

16 Daniel Casini, Luca Abeni, Alessandro Biondi, Tommaso Cucinotta, and Giorgio Buttazzo. Constant bandwidth servers with constrained deadlines. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 68–77, 2017.

17 Daniel Casini, Tobias Blaß, Ingo Lütkebohle, and Björn B Brandenburg. Response-time analysis of ROS 2 processing chains under reservation-based scheduling. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

18 Pierre-Julien Chaine, Marc Boyer, Claire Pagetti, and Franck Wartel. Egress-TT Configurations for TSN Networks. In *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, RTNS 2022, 2022.

19 Hyon-Young Choi, Andrew L. King, and Insup Lee. Making DDS really real-time with OpenFlow. In *2016 International Conference on Embedded Software (EMSOFT)*, 2016.

20 Hyunjong Choi, Yecheng Xiang, and Hyoseung Kim. PiCAS: New design of priority-driven chain-aware scheduling for ROS2. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 251–263. IEEE, 2021.

21 Rut Diane Cuebas, Seonghyeon Park, Youngeun Cho, Daechul Park, and Chang-Gun Lee. Extension of functionally and temporally correct simulation of cyber-systems of automotive systems based on ROS system. *Korean Information Science Society Academic Papers*, pages 1174–1176, 2019.

22 Dakshina Dasari, Matthias Becker, Daniel Casini, and Tobias Blaß. End-to-end analysis of event chains under the QNX adaptive partitioning scheduler. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 214–227, 2022.

23 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*, 2007.

24 Daniel Bristot de Oliveira, Daniel Casini, Rômulo Silva de Oliveira, and Tommaso Cucinotta. Demystifying the real-time Linux scheduling latency. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

25 Jonas Diemer, Philip Axer, and Rolf Ernst. Compositional Performance Analysis in Python with pyCPA. In *In Proceedings of WATERS'12*, 2012.

26 Zheng Dong, Weisong Shi, Guangmo Tong, and Kecheng Yang. Collaborative autonomous driving: Vision and challenges. In *2020 International Conference on Connected and Autonomous Driving (MetroCAD)*, pages 17–26. IEEE, 2020.

27 eProsima. Fast-DDS, 2022. `https://fast-dds.docs.eprosima.com/en/latest/`.

28 eProsima. Fast-DDS Github repository, 2022. `https://github.com/eProsima/Fast-DDS`.

29 Mario Günzel, Kuan-Hsun Chen, Niklas Ueter, Georg von der Brüggen, Marco Dürr, and Jian-Jia Chen. Timing analysis of asynchronized distributed cause-effect chains. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021.

30 A. Hamann, D. Dasari, F. Wurst, I. Sañudo, N. Capodieci, P. Burgio, and M Bertogna. Waters industrial challenge 2019.

31 Arne Hamann, Selma Saidi, David Ginthoer, Christian Wietfeld, and Dirk Ziegenbein. Building end-to-end IoT applications with QoS guarantees. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

32 R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis – The SymTA/S approach. *IEEE Proceedings – Computers and Digital Techniques*, March 2005.

33 Marek Jersak. *Compositional Performance Analysis for Complex Embedded Applications*. PhD thesis, Technical University of Braunschweig, June 2004.

34 Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 287–296, 2018.

35 Bonjun Kim and Kiejin Park. Probabilistic delay model of dynamic message frame in flexray protocol. *IEEE Transactions on Consumer Electronics*, 55(1):77–82, 2009.

36 Kirill Krinkin, Antoni Filatov, Artyom Filatov, Oleg Kurishev, and Alexander Lyanguzov. Data distribution services performance evaluation framework. In *2018 22nd Conference of Open Innovations Association (FRUCT)*, pages 94–100, 2018.

37 Tobias Kronauer, Joshwa Pohlmann, Maximilian Matthé, Till Smejkal, and Gerhard P. Fettweis. Latency analysis of ROS2 multi-node systems. *2021 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*, pages 1–7, 2021.

38 J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *[1989] Proceedings. Real-Time Systems Symposium*, pages 166–171, 1989.

39 Juri Lelli, Claudio Scordino, Luca Abeni, and Dario Faggioli. Deadline scheduling in the linux kernel. *Softw. Pract. Exper.*, 46(6):821–839, June 2016.

40 G. Lipari and E. Bini. Resource partitioning among real-time applications. In *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings.*, pages 151–158, 2003.

41 Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ROS2. In *EMSOFT '16*, New York, NY, USA, 2016. Association for Computing Machinery.

42 Philipp Mundhenk, Arne Hamann, Andreas Heyl, and Dirk Ziegenbein. Reliable distributed systems. In *2022 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2022.

43 Mitra Nasri and Bjorn B Brandenburg. An exact and sustainable analysis of non-preemptive scheduling. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 12–23. IEEE, 2017.

44 Ramon Serna Oliver and Gerhard Fohler. Probabilistic estimation of end-to-end path latency in wireless sensor networks. In *2009 IEEE 6th International Conference on Mobile Adhoc and Sensor Systems*, pages 423–431. IEEE, 2009.

45 OMG. Supported QoS, April 2015. `https://www.omg.org/spec/DDS/1.4/PDF`.

46 OMG. The real-time publish-subscribe protocol dds interoperability wire protocol specification (v2.5), March 2021. `https://www.omg.org/spec/DDSI-RTPS/2.5/PDF`.

47 J.C. Palencia and M. Gonzalez Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings 19th IEEE Real-Time Systems Symposium*, pages 26–37, 1998.

48 G. Pardo-Castellote. OMG data distribution service: architectural overview. In *IEEE Military Communications Conference, 2003. MILCOM 2003.*, volume 1, pages 242–247 Vol.1, 2003.

49 Gaetano Patti, Lucia Lo Bello, and Luca Leonardi. Deadline-Aware Online scheduling of TSN flows for automotive applications. *IEEE Transactions on Industrial Informatics*, 2022.

50 Jonas Peeck, Mischa Möstl, Tasuku Ishigooka, and Rolf Ernst. A middleware protocol for time-critical wireless communication of large data samples. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 1–13, 2021.

51 Michael Pöhnl, Alban Tamisier, and Tobias Blass. A Middleware Journey from Microcontrollers to Microprocessors. In *2022 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 282–286, 2022.

52 Hootan Rashtian and Sathish Gopalakrishnan. Balancing message criticality and timeliness in IoT networks. *IEEE Access*, 7:145738–145745, 2019.

53 Joao Rodrigues, Hugo Miranda, João Ventura, and Luıs Rodrigues. The design of RT-Appia. In *Proceedings Sixth International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 261–268. IEEE, 2001.

54 RTI. Connext-DDS, 2013. `https://www.rti.com/products/dds-standard`.

55 Johannes Schlatow and Rolf Ernst. Response-time analysis for task chains with complex precedence and blocking relations. *ACM Trans. Embed. Comput. Syst.*, September 2017.

56 Claudio Scordino, Angela Gonzalez Mariño, and Francesc Fons. Hardware Acceleration of Data Distribution Service (DDS) for Automotive Communication and Computing. *IEEE Access*, 10:109626–109651, 2022.

57 Katherine Scott, Chris Lalancette, and Audrow Nash. 2021 ROS Middleware Evaluation Report, 2021. https://github.com/osrf/TSC-RMW-Reports/tree/main/humble.

58 Maria A Serrano, Alessandra Melani, Roberto Vargas, Andrea Marongiu, Marko Bertogna, and Eduardo Quinones. Timing characterization of OpenMP4 tasking model. In *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 157–166. IEEE, 2015.

59 Alejandro Serrano-Cases, Juan M Reina, Jaume Abella, Enrico Mezzetti, and Francisco J Cazorla. Leveraging hardware QoS to control contention in the Xilinx Zynq UltraScale+ MPSoC. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

60 Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*, pages 2–13, 2003.

61 Emiliano Sisinni, Abusayeed Saifullah, Song Han, Ulf Jennehag, and Mikael Gidlund. Industrial Internet of Things: Challenges, opportunities, and directions. *IEEE transactions on industrial informatics*, 14(11):4724–4734, 2018.

62 A. Stevanato, A. Biondi, A. Biasci, and B. Morelli. Virtualized DDS Communication for Multi-Domain systems: Architecture and performance evaluation of design alternatives. In *Proceedings of the 29th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), San Antonio, USA, May 9-12, 2023*, 2023.

63 Jinghao Sun, Nan Guan, Zhishan Guo, Yekai Xue, Jing He, and Guozhen Tan. Calculating worst-case response time bounds for OpenMP programs with loop structures. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 123–135. IEEE, 2021.

64 Yue Tang, Zhiwei Feng, Nan Guan, Xu Jiang, Mingsong Lv, Qingxu Deng, and Wang Yi. Response time analysis and priority assignment of processing chains on ROS2 executors. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 231–243, 2020.

65 Yue Tang, Xu Jiang, Nan Guan, Dong Ji, Xiantong Luo, and Wang Yi. Comparing communication paradigms in cause-effect chains. *IEEE Transactions on Computers*, 2022.

66 H. Teper, M. Günzel, N. Ueter, G. von der Brüggen, and J. Chen. End-to-end timing analysis in ROS2. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, 2022.

67 Ludovic Thomas, Ahlem Mifdaoui, and Jean-Yves Le Boudec. Worst-case delay bounds in time-sensitive networks with packet replication and elimination. *IEEE/ACM Transactions on Networking*, pages 1–15, 2022.

68 Vortex. Cyclone-DDS, September 2021. https://projects.eclipse.org/projects/iot.cyclonedds.

69 Tianze Wu, Baofu Wu, Sa Wang, Liangkai Liu, Shaoshan Liu, Yungang Bao, and Weisong Shi. Oops! It's Too Late. Your Autonomous Driving System Needs a Faster Middleware. *IEEE Robotics and Automation Letters*, 6(4):7301–7308, 2021.

70 Xiaoming Zhou and Piet Van Mieghem. Reordering of IP packets in Internet. In Chadi Barakat and Ian Pratt, editors, *Passive and Active Network Measurement*, pages 237–246, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

71 Matteo Zini, Giorgiomaria Cicero, Daniel Casini, and Alessandro Biondi. Profiling and controlling I/O-related memory contention in COTS heterogeneous platforms. *Software: Practice and Experience*, 52(5):1095–1113, 2022.

On the Equivalence of Maximum Reaction Time and Maximum Data Age for Cause-Effect Chains

Mario Günzel
Department of Computer Science, TU Dortmund University, Germany

Harun Teper
Department of Computer Science, TU Dortmund University, Germany

Kuan-Hsun Chen
University of Twente, The Netherlands

Georg von der Brüggen
Department of Computer Science, TU Dortmund University, Germany

Jian-Jia Chen
Lamarr Institute for Machine Learning and Artificial Intelligence, Dortmund, Germany
Department of Computer Science, TU Dortmund University, Germany

Abstract

Real-time systems require a formal guarantee of timing-constraints, not only for individual tasks but also for data-propagation. The timing behavior of data-propagation paths in a given system is typically described by its maximum reaction time and its maximum data age. This paper shows that they are equivalent.

To reach this conclusion, partitioned job chains are introduced, which consist of one immediate forward and one immediate backward job chain. Such partitioned job chains are proven to describe maximum reaction time and maximum data age in a universal manner. This universal description does not only show the equivalence of maximum reaction time and maximum data age, but can also be exploited to speed up the computation of such significantly. In particular, the speed-up for synthesized task sets based on automotive benchmarks can be up to 1600.

Since only very few non-restrictive assumptions are made, the equivalence of maximum data age and maximum reaction time holds for almost any scheduling mechanism and even for tasks which do not adhere to the typical periodic or sporadic task model. This observation is supported by a simulation of a ROS2 navigation system.

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

Keywords and phrases End-to-End, Timing Analysis, Maximum Data Age, Maximum Reaction Time, Cause-Effect Chain, Robot Operating Systems 2 (ROS2)

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.10

Funding This work has been supported by Deutsche Forschungsgemeinschaft (DFG), as part of Sus-Aware (Project No. 398602212). This result is part of a project (PropRT) that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 865170). This work was partially funded by the German Federal Ministry of Education and Research (BMBF) in the course of the 6GEM research hub under grant number 16KISK038.

6GEM

1 Introduction

In various embedded system applications, e.g., in automotive or avionics systems, a sequence of tasks is necessary to perform a certain functionality. The data dependency between these tasks is described by a cause-effect chain. A typical example is a cause-effect chain from a

© Mario Günzel, Harun Teper, Kuan-Hsun Chen, Georg von der Brüggen, and Jian-Jia Chen;
licensed under Creative Commons License CC-BY 4.0

sensor to an actuator, where the first task reads the sensor value (cause), the second task processes the data, and the third task produces an output for the actuator (an effect is triggered). Timing properties of such cause-effect chains must be validated to ensure system safety.

Typical metrics to describe end-to-end timing properties are maximum reaction time (MRT) and maximum data age (MDA). The MRT is the longest time interval from an external cause until the earliest time where this external cause is fully processed (also called the maximum button to action delay). The MDA is the longest time interval starting from sampling a value and ending at an actuation that takes place based on that sampled data (also called the data freshness). Due to their importance, multiple approaches to calculate or bound MRT or MDA have been provided [2, 3, 6, 8–11, 13, 17, 24, 26].

However, the relation between the MRT and MDA values is less analyzed. This is kind of surprising, since answering the questions *if, how, and in which scenarios the MRT and MDA values are related* is interesting – both from a practical perspective (since it may be sufficient to analyze one metric instead of two) and from a research perspective (since analysis methods for one metric may also be applied when analyzing the other). Hence, in this work, we focus on such relations between MRT and MDA. The strongest analytical result known has been provided by Günzel et al. [13], showing that the MRT is an upper bound for the MDA.

Nonetheless, empirical observations suggest a stronger relation. More specifically, the AUTOSAR Timing Extensions [1] provide an important observation about the relation between MRT and MDA, namely, that "*without over- and undersampling, age and reaction are the same*" [1, Section 7.2, p. 149]. However, while this observation seems to imply that MRT and MDA can differ for systems with over- or undersampling, recent measurements in Robot Operating System 2 (ROS2) show that the observed MRT and MDA always coincide [27]. Hence, it is unclear in which scenarios MRT and MDA coincide. Even more, while both observations suggest a strong relation between MRT and MDA, no proof for such a relation is provided.

Hence, in this paper, we further investigate MRT and MDA through analytical discussion to determine *if, how, and in which scenarios the MRT and MDA values are related.* Specifically, we formally prove that they are equivalent after a warm-up period, i.e., after the data passes the complete cause-effect chain once. This insight allows the verification of timing constraints for both metrics at the same time. Moreover, analytical results in the literature for one metric can be utilized for the other one.

We build on the established result [9, 13] that for each cause-effect chain MRT and MDA can be calculated based on the length of the related job chains; that is, the time interval between the moment the first job in the chain reads data and the moment the last job in the chain writes data. On a high level, our idea is to first examine the job chains that must be considered for the MRT; that is, the job chains from the first task in the chain (i.e., the sensor) to the last (i.e., the actuator). In the next step, we consider a chain comprised of two sub-chains, both starting from the second task in the chain – one going back to the sensor and one going forward to the actuator. We call such a chain a p-partitioned job chain, where p denotes the position of the task in the chain (in this case, $p = 2$). We show that one of these 2-partitioned job chains has at least the same length as the job chain that determines the MRT. We continue by induction over the tasks in the chain, showing that the MRT is upper-bounded by the MDA. Afterwards, we similarly show that the MDA is upper-bounded by the MRT by starting from the last task in the chain.

Contributions. We show the equivalence of MRT and MDA while making only very few non-restrictive assumptions regarding tasks, communication, and scheduling model. Therefore, our results apply for a large variety of systems. Specifically, our results can be applied for but are not limited to periodic or sporadic tasks and implicit communication or logical execution time (LET) [15]. The underlying scheduler may be (i) a job-level or task-level static priority scheduler (e.g., EDF and rate-monotic, respectively), (ii) work-conserving or non-work-conserving, and (iii) preemptive or non-preemptive. Furthermore, jobs may be executed on different processing units (e.g., on different electronic control units (ECUs)). In detail, we make the following technical contributions:

- In Section 4, we define p-partitioned job chains, where p is an integer value not larger than the number of jobs $|\overline{E}|$ in the evaluated job chain. We show that MRT and MDA can be expressed as 1-partitioned job chains and $\overline{E}$-partitioned job chains, respectively.
- In Section 5, we discuss the equivalence between MRT and MDA using partitioned job chains and show that the timing behavior is independent of p, i.e., any arbitrary p can be chosen to compute MDA and MRT.
- The implication of our results in practice is discussed in Section 6.
- We discuss how to apply our results to a reduced version of MRT and MDA in Section 7.1 and how this equivalence can be transferred to a definition of MRT and MDA based on valid job chains Section 7.2.
- We evaluate our results considering randomly generated periodic tasks, communicating via Logical Execution Time (LET) [15] in Section 8. In particular, we show that by the right choice of p the required time for computation is reduced significantly.
- To validate our theoretical results, we examine MRT, MDA on a ROS2 system with non-periodic tasks under implicit communication in Section 9.

2 System Model and Problem Definition

This section introduces the definitions and notations for task model, communication model, and cause-effect chains, as well as our problem definition. For our analysis, a very general task and communication model with very few assumptions is sufficient. Yet, we introduce the notion of *periodic* and *sporadic* tasks as well as the communication policies *implicit communication* and *logical execution time (LET)*, as they are utilized to provide intuitive examples, in the empirical evaluation with synthesized tasks sets in Section 8, and for the case study in Section 9. Our notation is summarized in Table 1.

Jobs and Tasks. A *job* is a program instance that produces output based on its input. The aggregation of all jobs of the same program is called a *task*, denoted by τ. We denote the (countably many) jobs of task τ by $(\tau(m))_{m\in\mathbb{N}^+}$, and denote the induced ordering by $\tau(i) \preccurlyeq \tau(j)$ if and only if $i \leq j$. The set of all tasks in the system is denoted by $\mathbb{T}$.

For our analysis, we need no further assumption on the job releases. Specifically, we make no assumptions regarding the first time a job of a task arrives or on the inter-arrival pattern of jobs. However, the most common task models, namely the periodic and the sporadic model, both fulfill our assumptions.

A **periodic task** τ is described as $\tau = (C_\tau, T_\tau, \phi_\tau) \in \mathbb{R}^3$, where $C_\tau \geq 0$ is the worst-case execution time (WCET), $T_\tau > 0$ is the period, and ϕ_τ is the phase of the task. The first job $\tau(1)$ of τ is released at time ϕ_τ and subsequent jobs are released every T_τ time units, i.e., $\tau(m)$ is released at time $\phi_\tau + (m-1)T_\tau$. Every job of τ executes for at most C_τ time units.

Table 1 Notation in this work.

Variable	Definition
$\mathbb{T}$	task set under analysis
$\tau \in \mathbb{T}$	a task
$\tau(m),\ m \in \mathbb{N}^+$	m-th job of task τ
$\preccurlyeq$	job ordering for jobs of one task
$\tau = (C_\tau, T_\tau, \phi_\tau)$	periodic task
$\tau = (C_\tau, T_\tau)$	sporadic task
$\text{re}(J)$	time of read-event of a job J
$\text{we}(J)$	time of write-event of a job J
$\overline{E}$	cause-effect chain under analysis
E	any cause-effect chain (can be $\overline{E}$ or a sub-chain of $\overline{E}$)
$\lvert E\rvert$	number of tasks in E
$E(i),\ i \in \{1, \dots, \lvert E\rvert\}$	i-th task of E

A **sporadic tasks** τ is described by the tuple $\tau = (C_\tau, T_\tau) \in \mathbb{R}^2$, where $C_\tau \geq 0$ is the worst-case execution time (WCET), and $T_\tau > 0$ is the minimum inter-arrival time between two jobs. The release of two subsequent jobs of τ are separated by at least T_τ time units and each job executes for at most C_τ time units.

Communication. Jobs communicate by receiving (reading) their input from a shared resource and handing over (writing) their output to a shared resource. We denote the time that the *read-event* of a job J takes place by $\text{re}(J) \in \mathbb{R}$ and the time that the *write-event* of J takes place by $\text{we}(J) \in \mathbb{R}$. We assume the following two requirements are met:

- **(R1)** For each task $\tau \in \mathbb{T}$, the read- and write-events of its jobs are ordered in the sense that $\text{re}(\tau(m)) < \text{re}(\tau(m+1))$, $\text{we}(\tau(m)) < \text{we}(\tau(m+1))$, and $\text{re}(\tau(m)) \leq \text{we}(\tau(m))$ for all $m \in \mathbb{N}$.
- **(R2)** The sets $\{\text{re}(\tau(m)) \mid m \in \mathbb{N}\}$ and $\{\text{we}(\tau(m)) \mid m \in \mathbb{N}\}$ have no accumulation point, i.e., the number of read- and write-events in each bounded time interval is finite.

These not very restrictive requirements are fulfilled by commonly considered communication semantics, e.g., for *logical execution time* and *implicit communication.*

Logical Execution Time (LET): Under logical execution time [15], each task τ is assigned an arbitrary deadline D_τ, and the read-event and write-event of each job J of τ is set to its release time r_J and its absolute deadline $r_J + D_\tau$, respectively.

Implicit Communication: Under implicit communication, each job has its read-event at the first time that it is executed, i.e., when the job starts, and the job has its write-event at the last time it is executed, i.e., when the job finishes.

Cause-effect chains. A *cause-effect chain* $E = (\tau_1 \to \tau_2 \to \dots \to \tau_k)$, with $k \in \mathbb{N}$, describes the path of data through different programs by a finite sequence of tasks $\tau_i \in \mathbb{T}$. The number of entries in the sequence E is denoted as $|E|$ and $E(i)$ is the task at the i-th entry of E for $i \in \{1, \dots, |E|\}$. This definition of cause-effect chains is inspired by event-chains of the AUTOSAR Timing Extensions [1], which represent chains of more general functional dependency. We assume implicit sampling, where the sampling for a cause-effect chain E happens at the read-event of each job of $E(1)$. However, we can easily model any kind of sampling by adding a *sampling task* τ^{sample} to the system, where each job $\tau^{sample}(1), \tau^{sample}(2), \dots$ reads and writes data at a time when the sampling happens.

If the data dependency is described by a directed acyclic graph (DAG) with several sources and sinks, we follow the typical approach from the literature and analyze each cause-effect chain (i.e., path through the DAG from a source to a sink) individually.

Problem definition. For a given cause-effect chain $\overline{E}$, we discuss the equivalence of its maximum reaction time and maximum data age. Specifically, we answer the question: *To which extent can the maximum reaction time and maximum data age (and their variations) be derived from each other?* We answer this question for large variety of systems, including periodic or sporadic task systems with LET or implicit job communication.

3 Maximum Reaction Time & Maximum Data Age

This section specifies the analyzed End-to-End latencies, namely, the *maximum reaction time* (MRT), which is the length of the longest time interval from an external cause until the earliest time where this external cause is fully processed (i.e., the maximum button to action delay), and the *maximum data age* (MDA), which is the length of the longest time interval starting from sampling a value until an actuation based on that sampled value takes place (i.e., the data freshness). The definitions are based on *job chains* which represent the path of data through the schedule. We recap the job chain definitions for arbitrary cause-effect chains E of the task set $\mathbb{T}$ as stated by Günzel et al. [13].

▶ **Definition 1** (Job chain [13])**.** Let E be a cause effect chain of the task set $\mathbb{T}$. A job chain $c = (J_1, \dots, J_{|E|})$ for E is a sequence of jobs where the following two conditions are fulfilled:

- J_i is a job of task $E(i)$ for all $i \in \{1, 2, \dots, |E|\}$.
- Each job in the chain reads the data not before it was written by the previous job in the chain. That is, $\text{we}(J_{i-1}) \leq \text{re}(J_i)$ for all $i \in \{2, 3, \dots, |E|\}$.

The *length of a job chain* c is the length of the time interval between the read-event of the first job J_1 in the chain and the write-event of the last job $J_{|E|}$ in the chain, i.e.,

$$\ell(J_1, J_2, \dots, J_{|E|}) = \text{we}(J_{|E|}) - \text{re}(J_1). \tag{1}$$

We denote the i-th job in c as $c(i)$ for $i \in \{1, 2, \dots, |E|\}$, hence, $\ell(c) = \text{we}(c(|E|)) - \text{re}(c(1))$.

Like in previous work [9,13], our analysis for MRT and MDA is built on two related types of jobs chains: *Immediate forward* job chains and *immediate backward* job chains.

▶ **Definition 2** (Immediate forward job chain [13])**.** Let E be a cause-effect chain for task set $\mathbb{T}$. A job chain $c = (J_1, J_2, \dots, J_{|E|})$ for E is *immediate forward* if for all $i \in \{2, \dots, |E|\}$ the job J_i is the *earliest* job of task $E(i)$ with read-event no earlier than the write-event of J_{i-1}. That is, J_i is the earliest job that fulfills the properties from Definition 1.
We denote the m-th immediate forward job chain for E (i.e., $J_1 = E(1)(m)$) by $\vec{c}^{E}_{m}, m \in \mathbb{N}^+$.

▶ **Definition 3** (Immediate backward job chain [13])**.** Let E be a cause-effect chain for task set $\mathbb{T}$. A job chain $c = (J_1, J_2, \dots, J_{|E|})$ for E is *immediate backward* if for all $i \in \{|E| - 1, \dots, 1\}$ the job J_i is the *latest* job of task $E(i)$ with write-event no later than the read-event of J_{i+1}. That is, J_i is the latest job that fulfills the properties from Definition 1.
For $m \in \mathbb{N}^+$, if there is an immediate backward job chain with $J_{|E|} = E(|E|)(m)$, then we call it the m-th immediate backward job chain $\overleftarrow{c}^{E}_{m}$.

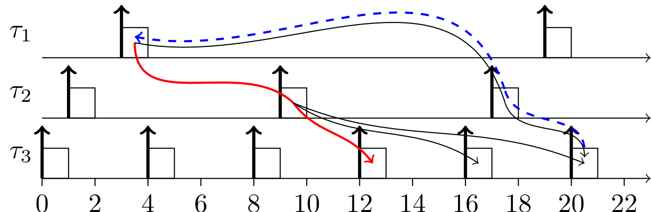


Figure 1 Cause-effect chain $E = (\tau_1 \to \tau_2 \to \tau_3)$ for three tasks with implicit communication. Forward arrows mark the job chains with first job $\tau_1(1)$, the *immediate forward* job chain $\vec{c}_1^E$ is marked red. The dashed blue arrow marks the *immediate backward* job chain $\overleftarrow{c}_6^E$ starting at $\tau_3(6)$.

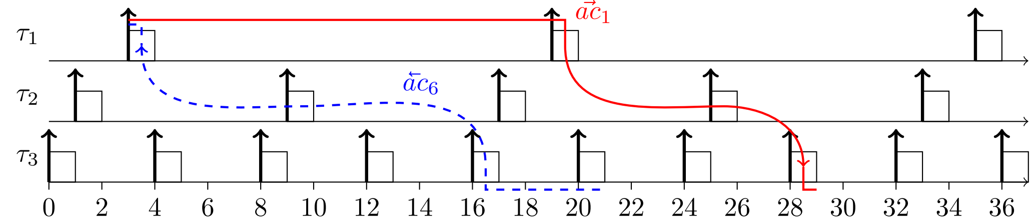


Figure 2 Cause-effect chain $E = (\tau_1 \to \tau_2 \to \tau_3)$ with implicit communication. The immediate forward augmented job chain $\vec{ac}_1 = (3, \tau_1(2), \tau_2(4), \tau_3(8), 29)$ is marked red, and the immediate backward augmented job chain $\overleftarrow{ac}_6 = (3, \tau_1(1), \tau_2(2), \tau_3(5), 21)$ is marked by a dashed blue arrow.

For an example system with three tasks and implicit communication, the forward arrows in Figure 1 mark all job chains starting at the job $\tau_1(1)$. The immediate forward job chain starting at $\tau_1(1)$ is marked red. The immediate backward job chain starting at $\tau_3(6)$ is marked with a dotted blue arrow.

To account for the time of the external activity (z) and the actuation (z'), we consider augmented job chains.

▶ **Definition 4** (Immediate forward augmented job chain [13]). Let $m \in \mathbb{N}^+$. We define the m-th immediate forward augmented job chain for $\overline{E}$ by $\vec{ac}_m = (z, J_1, \dots, J_{|\overline{E}|}, z')$, where $z \in \mathbb{R}$ is just after the read-event of the m-th job of task $\overline{E}(1)$, $(J_1, \dots, J_{|\overline{E}|})$ is the $(m+1)$-th immediate forward job chain for $\overline{E}$, and z' is at the write-event of $J_{|\overline{E}|}$.

▶ **Definition 5** (Immediate backward augmented job chain [13]). Let $m \in \mathbb{N}^+$. If $\overleftarrow{c}_{m-1}^{\overline{E}}$ exists, then we define the m-th immediate backward augmented job chain for $\overline{E}$ by $\overleftarrow{ac}_m = (z, J_1, \dots, J_{|\overline{E}|}, z')$, where z' is just before the write-event of the m-th job of task $\overline{E}(|\overline{E}|)$, $(J_1, \dots, J_{|\overline{E}|})$ is the $(m-1)$-th immediate backward job chain for $\overline{E}$, and z is at the read-event of job J_1.

Examples of an immediate forward augmented job chain and an immediate backward augmented job chain are illustrated in Figure 2. Note that $\overleftarrow{ac}_m$ can only be constructed if $\overleftarrow{c}_{m-1}^{\overline{E}}$ exists. Moreover, if $\overleftarrow{ac}_m$ exists, then for all $\tilde{m} \geq m$, $\overleftarrow{ac}_{\tilde{m}}$ exists as well. Our notation related to job chains is summarized in Table 2.

MRT and MDA can be defined based on Definition 4 and Definition 5 directly. However, if not all tasks in the chain are released at system start, data may be processed by some tasks in the chain but not by all and no actuation based on this data can happen. For example, in Figure 3 data written by $\tau_1(1)$ is overwritten by $\tau_1(2)$. Similarly, data written by $\tau_1(2)$ is

Table 2 Job chains and extensions defined in this paper.

Symbol	Name	Defined in
c	Job chain	Sec. 3, Def. 1
$\vec{c}$	Immediate forward job chain	Sec. 3, Def. 2
$\overleftarrow{c}$	Immediate backward job chain	Sec. 3, Def. 3
$\vec{ac}$	Immediate forward augmented job chain	Sec. 3, Def. 4
$\overleftarrow{ac}$	Immediate backward augmented job chain	Sec. 3, Def. 5
pc	Partitioned job chain	Sec. 4, Def. 9

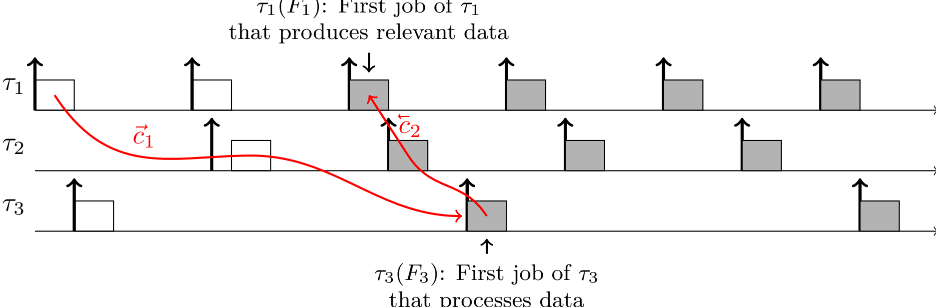


Figure 3 Schedule for the cause-effect chain $\overline{E} = (\tau_1 \to \tau_2 \to \tau_3)$. Data arrives at job $\tau_3(2)$. The warm-up covers the white jobs. MRT and MDA of the gray jobs coincides.

used only by $\tau_2(1)$ but never reaches the last task because $\tau_2(2)$ overwrites the data written by $\tau_2(1)$. Hence, intuitively, only the gray jobs that process data that might be used in an actuation should be considered when determining MRT and MDA. We say that the system has *warmed up* when the complete chain processes data for the first time.

The first job of task $\overline{E}(|\overline{E}|)$ that reads data processed by all jobs in $\overline{E}$ can be determined by one immediate forward job chain, i.e., it is the last entry of $\vec{c}_1$. The immediate backward job chain based on that job exists and determines the first job of $\overline{E}(1)$ which processes data that reaches task $\overline{E}(|\overline{E}|)$. The procedure is summarized in Figure 3. We formalize this by specifying the *warm-up* period of the system.

▶ **Definition 6** (Warm-up). Let $F \in \mathbb{N}^+$ such that $\vec{c}_1^{\overline{E}}(|\overline{E}|)$ is the F-th job of $\overline{E}(|\overline{E}|)$. Then we construct the immediate backward job chain $\overleftarrow{c}_F^{\overline{E}}$ and denote by $F_1, \dots, F_{|\overline{E}|} \in \mathbb{N}^+$ the job number of each job in $\overleftarrow{c}_F^{\overline{E}}$, i.e., $\overleftarrow{c}_F^{\overline{E}} = (\overline{E}(1)(F_1), \dots, \overline{E}(|\overline{E}|)(F_{|\overline{E}|}))$.[1]

The *warm-up* covers all jobs of $\overline{E}(i)$ before $\overline{E}(i)(F_i)$, $i = 1, \dots, |\overline{E}|$. In particular, those jobs are not considered for the maximum reaction time or maximum data age.

Only augmented job chains with z no earlier than the read-event of $\overline{E}(1)(F_1)$ should be considered for the end-to-end latency. These are all $\vec{ac}_m$ with $m \geq F_1$ and all $\overleftarrow{ac}_m$ with $m \geq F_{|\overline{E}|} + 1$. For the following definitions, the length of an immediate forward or an immediate backward augmented job chain is the length of the time interval from z to z', i.e., $\ell(z, J_1, \dots, J_{|\overline{E}|}, z') = z' - z$.

[1] The existence of the immediate forward job chain $\vec{c}_1^{\overline{E}}$ ensures that $\overleftarrow{c}_F^{\overline{E}}$ can be fully constructed, i.e., during the backwards construction a job that writes data early enough can always be found. This is captured by Lemma 13 and not further discussed here to improve the reading flow.

▶ **Definition 7** (Maximum reaction time). The maximum reaction time for $\overline{E}$ is defined as

$$\mathrm{MRT} = \sup_{m \geq F_1} \ell(\vec{ac}_m) \tag{2}$$

▶ **Definition 8** (Maximum data age). The maximum data age for $\overline{E}$ is defined as

$$\mathrm{MDA} = \sup_{m \geq F_{|\overline{E}|}+1} \ell(\overleftarrow{ac}_m) \tag{3}$$

In this work, for MRT the time from the external event z is included in the definition, as it is also done by Feiertag et al. [10], Dürr et al. [9], and Günzel et al. [13]. For the MDA, however, z' is included by Günzel et al. [13] but not by Feiertag et al. [10] and Dürr et al. [9]. We refer to this definition as maximum reduced data age (MRDA). These definitions cover slightly different scenarios: (i) The MRDA assumes that the actuation is directly triggered by the write event of the last task in the chain, while (ii) the MDA assumes the actuation is not directly triggered; thus, additional time for the actuation has to be included, which, in the worst case, may happen directly before the next write event of the last task in the chain. How MDA relates to MRDA is discussed in Section 7.1. Note, while not specifically stating this, AUTOSAR [1] considers MDA, as explained at the end of Section 7.1.

The definitions of MRT and MDA are similar in the work by Günzel et al. [13]. However, their calculation of MRT and MDA starts as soon as the augmented job chains become *valid*, which potentially includes part of the warm-up period. We discuss in Section 7.2 how our results can be extended to the definition of [13].

4 Partitioned Job chains

In this section, for a given task chain $\overline{E}$, we define p-partitioned job chains ($p \in \{1, 2, \ldots, |\overline{E}|\}$). We show that these p-partitioned job chains allow maximum reaction time (MRT) and maximum data age (MDA) definitions that are equivalent to the ones based on augmented job chains stated in Section 3. In particular, a 1-partitioned job chain is equivalent to an immediate forward augmented job chain and an $|\overline{E}|$-partitioned job chain is equivalent to an immediate backward augmented job chain. In Section 5, we utilize these p-partitioned job chains to show the equivalence between MRT and MDA by discussing the difference of p-partitioned and $(p+1)$-partitioned job chains for $p = 1, \ldots, |\overline{E}| - 1$.

A p-partitioned job chain is a combination of (i) two subsequent jobs J_p and $\tilde{J}_p$ of task $\overline{E}(p)$, (ii) an immediate backward job chain that starts at J_p, and (iii) an immediate forward job chain that starts at $\tilde{J}_p$. In Figure 4, the dashed chain pc_1^2 is a 2-partitioned job chain consisting of an immediate backward job chain with last entry $\tau_2(1)$, an immediate forward job chain with first entry $\tau_2(2)$, and the connection between $\tau_2(1)$ and $\tau_2(2)$.

We start by formally defining partitioned job chains.

▶ **Definition 9** (Partitioned job chain). Let $p \in \{1, \ldots, |\overline{E}|\}$ and $m \in \mathbb{N}^+$. Moreover, let $\overline{E}_p^{first} = (\overline{E}(1) \to \cdots \to \overline{E}(p))$, and let $\overline{E}_p^{last} = (\overline{E}(p) \to \cdots \to \overline{E}(|E|))$. If $\overleftarrow{c}_m^{\overline{E}_p^{first}}$ exists, then we define the m-th p-partitioned job chain pc_m^p by

$$pc_m^p = (J_1, \ldots, J_p, \tilde{J}_p, \ldots, \tilde{J}_{|\overline{E}|}) \tag{4}$$

where $(J_1, \ldots, J_p) = \overleftarrow{c}_m^{\overline{E}_p^{first}}$ is the m-th immediate backward job chain for the cause-effect chain $\overline{E}_p^{first}$ and $(\tilde{J}_p, \ldots, \tilde{J}_{|E|}) = \vec{c}_{m+1}^{\overline{E}_p^{last}}$ is the $(m+1)$-th immediate forward job chain for $\overline{E}_p^{last}$. In particular, $J_p = \overline{E}(p)(m)$ and $\tilde{J}_p = \overline{E}(p)(m+1)$.

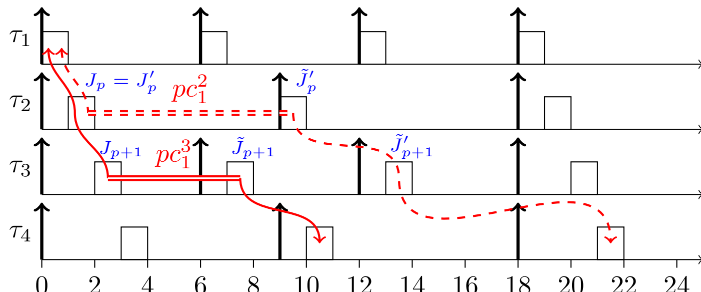


Figure 4 Four tasks with cause-effect chain $(\tau_1 \to \tau_2 \to \tau_3 \to \tau_4)$ communicating via implicit communication. The red line depicts the chain pc_1^3 whereas the dashed red line depicts pc_1^2. The length of pc_1^3 is upper bounded by the length of pc_1^2. The blue job annotations illustrate the proof of Lemma 12 ($\geq$-relation).

The length of a partitioned job chain is $\ell(J_1, \dots, J_p, \tilde{J}_p, \dots, \tilde{J}_{|\overline{E}|}) = \mathrm{we}(\tilde{J}_{|\overline{E}|}) - \mathrm{re}(J_1)$.

Since pc_m^1 is composed of $\vec{c}_{m+1}^{\overline{E}}$ and one additional job at the beginning (for the external activity), it is equivalent to $\vec{ac}_m$. Since $pc_m^{|\overline{E}|}$ is composed of $\overleftarrow{c}_m^{\overline{E}}$ and one additional job at the end (for the actuation), it is equivalent to $\overleftarrow{ac}_{m+1}$. For instance, in Figure 2 the chain $\vec{ac}_1$ is equivalent to the 1-partitioned job chain $pc_1^1 = (\tau_1(1), \tau_1(2), \tau_2(4), \tau_3(8))$, and $\overleftarrow{ac}_6$ is equivalent to the $|\overline{E}|$-partitioned job chain $pc_5^{|\overline{E}|} = (\tau_1(1), \tau_2(2), \tau_3(5), \tau_3(6))$. Note that the m-th $|\overline{E}|$-partitioned job chain $pc_m^{|\overline{E}|}$ is equivalent to the $(m+1)$-th augmented backward job chain $\overleftarrow{ac}_{m+1}$, i.e., the index is shifted by 1. Thus, their length is also the same:

▶ **Corollary 10.** *For all $m \in \mathbb{N}^+$ the following two properties hold.*
1. $\ell(pc_m^1) = \ell(\vec{ac}_m)$.
2. *$pc_m^{|\overline{E}|}$ exists if and only if $\overleftarrow{ac}_{m+1}$ exists. If they exist, then $\ell(pc_m^{|\overline{E}|}) = \ell(\overleftarrow{ac}_{m+1})$.*

We restate the definitions of MRT and MDA using p-partitioned job chains.

▶ **Definition 11** (MRT and MDA by partitioned job chains)**.** The maximum reaction time (MRT) and the maximum data age (MDA) of the cause-effect chain $\overline{E}$ can be expressed by partitioned job chains as follows:

$$\mathrm{MRT} = \sup_{m \geq F_1} \ell(pc_m^1) \tag{5}$$

$$\mathrm{MDA} = \sup_{m \geq F_{|\overline{E}|}} \ell(pc_m^{|\overline{E}|}) \tag{6}$$

5 Equivalence of MRT and MDA

In this section, we show the equivalence of maximum reaction time (MRT) and maximum data age (MDA). More precisely, we prove that, for a given cause-effect chain $\overline{E}$, the maximum length of a p-partitioned job chain is the same for all $p \in \{1, \dots, |\overline{E}|\}$. Since MRT and MDA can be expressed by 1-partitioned and by $|\overline{E}|$-partitioned job chains, respectively, this directly shows the equivalence of MRT and MDA.

We prove that the maximum length of a p-partitioned job chain is the same for all $p \in \{1, \dots, |\overline{E}|\}$ in two steps: (i) We show that for all $p \in \{|\overline{E}|, \dots, 2\}$ the length of every p-partitioned job chain is upper bounded by the length of a $(p-1)$ partitioned job chain. This scenario is depicted in Figure 4, where the length of pc_1^3 is upper bounded by the

Figure 5 Four tasks with cause-effect chain ($\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \rightarrow \tau_4$) communicating via implicit communication. The red line depicts the chain pc_2^2 whereas the dashed red line depicts pc_1^3. The length of pc_2^2 is upper bounded by the length of pc_1^3. The blue job annotations illustrate the proof of Lemma 12 ($\geq$-relation).

length of pc_1^2. (ii) Conversely, we show that for all $p \in \{1, \dots, |\overline{E}| - 1\}$ the length of every p-partitioned job chain is upper bounded by the length of a $(p+1)$-partitioned job chain. This is depicted in Figure 5, where the length of pc_2^2 is upper bounded by the length of pc_1^3.

One problem we have to consider during step (ii) is that some $(p+1)$-partitioned job chains may not be fully constructed. For example, if the first job of τ_1 in Figure 5 would be missing, then pc_1^3 could not be fully constructed and there would be no 3-partitioned job chain which provides an upper bound on the length of pc_2^2. This, however, does not impact our result, since we show that this scenario never occurs for p-partitioned job chains pc_m^p after the warm-up, i.e., if $m \geq F_p$ with F_p from Definition 6.

The following lemma indicates that p-partitioned job chains for different p can be used equivalently for the computation of MDA and MRT according to their description by partitioned job chains from Definition 11.

▶ **Lemma 12.** *For all $p \in \{1, \dots, |\overline{E}| - 1\}$, we have*

$$\sup \{\ell(pc_m^p) \,|\, m \geq F_p\} = \sup \left\{\ell(pc_m^{p+1}) \,\middle|\, m \geq F_{p+1}\right\}. \tag{7}$$

To prove this lemma, we apply fundamental properties of immediate forward and immediate backward job chains (which are part of the partitioned job chains). In particular:

- If an immediate forward job chain $\vec{c}$ starts at or before another job chain c, then no job of c can be *before* the job of $\vec{c}$ because in the forward chain always the earliest possible job is chosen during construction.
- If an immediate backward job chain $\overleftarrow{c}$ ends at or after another job chain c, then no job of c can be *after* the job of $\overleftarrow{c}$ because in the backward chain always the latest possible job is chosen during construction.

We formalize those two properties in the following lemma that we utilize to prove Lemma 12. It is formulated for an arbitrary cause-effect chain E which can be $\overline{E}$ or a sub-chain of $\overline{E}$.

▶ **Lemma 13.** *Let E be a cause-effect chain in $\mathbb{T}$, let c be a job chain for E, let $\vec{c}$ be an immediate forward job chain for E, and let $\overleftarrow{c}$ be an immediate backward job chain for E.*

1. *If $i \in \{1, \dots, |E|\}$ exists such that $\vec{c}(i) \preccurlyeq c(i)$, then $\vec{c}(j) \preccurlyeq c(j)$ for all $j \in \{i, \dots, |E|\}$.*
2. *If $i \in \{1, \dots, |E|\}$ exists such that $c(i) \preccurlyeq \overleftarrow{c}(i)$, then $c(j) \preccurlyeq \overleftarrow{c}(j)$ for all $j \in \{1, \dots, i\}$.*

The proof for Lemma 13 is provided in the appendix. We now provide the proof of Lemma 12. We split the proof into two steps, showing the $\geq$-relation and the $\leq$-relation, from which the equality directly follows.

To proof idea for the $\geq$-relation is as follows. First, we pick any p-partitioned and the related $(p+1)$-partitioned job chain that have the job of $\overline{E}(p)$ in common. Second, we show (i) that their immediate backward job chains both end at the same job of $\overline{E}(1)$, and (ii) that the immediate forward job chain related to the $p+1$-partitioned job chain ends not later than the one related to the p-partitioned job chain. For instance, in Figure 4, pc_1^3 and pc_1^2 have job $\tau_2(1)$ in common, both immediate backwards job chains end at $\tau_1(1)$, and pc_1^3 ends at $\tau_4(2)$ which is not later than the end of pc_1^2 at $\tau_4(3)$. Hence, $\ell(pc_1^2) \geq \ell(pc_1^3)$.

Proof of Lemma 12, $\geq$-relation. Let $m \geq F_{p+1}$. We denote the jobs of the partitioned job chain pc_m^{p+1} by $pc_m^{p+1} = (J_1, \dots, J_p, J_{p+1}, \tilde{J}_{p+1}, \dots, \tilde{J}_{|\overline{E}|})$. Let $\xi \in \mathbb{N}$ such that J_p is the ξ-th job of $\overline{E}(p)$, i.e., $\overline{E}(p)(\xi) = J_p$. We prove that the length of pc_m^{p+1} is upper bounded by the length of $pc_\xi^p = (J'_1, \dots, J'_p, \tilde{J}'_p, \dots, \tilde{J}'_{|\overline{E}|})$ by showing $\text{we}(\tilde{J}'_{|\overline{E}|}) - \text{re}(J'_1) \geq \text{we}(\tilde{J}_{|\overline{E}|}) - \text{re}(J_1)$ in two substeps: First, we show that J_1 and J'_1 are the same (i.e., $J_1 = J'_1$), and second, that pc_m^{p+1} finishes not later than pc_ξ^p (i.e., $\tilde{J}_{|\overline{E}|} \preccurlyeq \tilde{J}'_{|\overline{E}|}$). Important jobs of this proof are illustrated in Figure 4.

Step 1 ($J_1 = J'_1$): By definition, $J_p = J'_p$. Since $(J_1, \dots, J_p)$ and $(J'_1, \dots, J'_p)$ are immediate backward job chains with the same last entry, they coincide. Hence, $J_1 = J'_1$.

Step 2 ($\tilde{J}_{|\overline{E}|} \preccurlyeq \tilde{J}'_{|\overline{E}|}$): Since $(J_1, \dots, J_{p+1})$ is an immediate backward job chain, this means that $J_p = \overline{E}(p)(\xi)$ is the latest job with write-event no later than the read-event of J_{p+1}. Therefore, the write-event of the subsequent job of the same task, which is $\overline{E}(p)(\xi+1) = \tilde{J}'_p$, must be after the read-event of J_{p+1}, i.e., $\text{re}(J_{p+1}) < \text{we}(\tilde{J}'_p)$.
The job $\tilde{J}_{p+1}$ of $\overline{E}(p+1)$ subsequent to J_{p+1} either has its read-event before $\text{we}(\tilde{J}'_p)$ as well (i.e., $\text{re}(\tilde{J}_{p+1}) < \text{we}(\tilde{J}'_p)$) or is the earliest job of $\overline{E}(p+1)$ with $\text{re}(\tilde{J}_{p+1}) \geq \text{we}(\tilde{J}'_p)$. In both cases $\tilde{J}_{p+1} \preccurlyeq \tilde{J}'_{p+1}$ as $\text{re}(\tilde{J}'_{p+1}) \geq \text{we}(\tilde{J}'_p)$. Since $(\tilde{J}_{p+1}, \dots, \tilde{J}_{|\overline{E}|})$ and $(\tilde{J}'_{p+1}, \dots, \tilde{J}'_{|\overline{E}|})$ are both immediate forward job chains and $\tilde{J}_{p+1} \preccurlyeq \tilde{J}'_{p+1}$, we know $\tilde{J}_{|\overline{E}|} \preccurlyeq \tilde{J}'_{|\overline{E}|}$ by Lemma 13.

Combining Step 1 and 2, we get $\ell(pc_m^{p+1}) = \text{we}(\tilde{J}_{|\overline{E}|}) - \text{re}(J_1) \leq \text{we}(\tilde{J}'_{|\overline{E}|}) - \text{re}(J'_1) = \ell(pc_\xi^p)$. Since $(J_1, \dots, J_{p+1})$ is an immediate backward job chain and $E(p+1)(F_{p+1}) \preccurlyeq J_{p+1}$, by Lemma 13, $E(p)(F_p) \preccurlyeq J_p$ holds as well. As Step 1 shows $J_p = J'_p = E(p)(\xi)$, we conclude that $\xi \geq F_p$. Hence, $\ell(pc_m^{p+1}) \leq \ell(pc_\xi^p) \leq \sup \left\{ \ell(pc_\eta^p) \,\middle|\, \eta \geq F_p \right\}$.

Since $m \geq F_{p+1}$ is arbitrarily chosen, we obtain $\sup \left\{ \ell(pc_m^{p+1}) \,\middle|\, m \geq F_{p+1} \right\} \leq \sup \left\{ \ell(pc_\eta^p) \,\middle|\, \eta \geq F_p \right\}$, i.e., the relation $\geq$ holds for Equation (7) ◀

Similarly, we show the $\leq$-relation by picking any p-partitioned and the related $(p+1)$-partitioned job chain that have the job of $\overline{E}(p+1)$ in common. Second, we show (i) that their immediate forward job chains both end at the same job of $\overline{E}(|\overline{E}|)$, and (ii) that the immediate backward job chain related to the $p+1$-partitioned job chain ends not later than the one related to the p-partitioned job chain. For instance, in Figure 5, pc_1^3 and pc_1^2 have job $\tau_3(2)$ in common, both immediate forward job chains ends $\tau_4(2)$, and pc_1^3 ends at $\tau_1(1)$ which is not later then the end of pc_2^2 at $\tau_1(2)$. Hence, $\ell(pc_2^2) \leq \ell(pc_1^3)$.

Proof of Lemma 12, $\leq$-relation. Let $m \geq F_p$. We denote the jobs of the partitioned job chain pc_m^p by $pc_m^p = (J_1, \dots, J_p, \tilde{J}_p, \tilde{J}_{p+1}, \dots, \tilde{J}_{|\overline{E}|})$. Let $\xi \in \mathbb{N}$ such that $\tilde{J}_{p+1}$ is the ξ-th job of $\overline{E}(p+1)$, i.e., $\overline{E}(p+1)(\xi) = \tilde{J}_{p+1}$.

As an additional step, we must show that $\xi - 1 \geq F_{p+1}$ holds for the previous job of $\overline{E}(p+1)(\xi-1)$. Assume for contradiction that $\xi - 1 < F_{p+1}$. Then $\xi \leq F_{p+1}$. Therefore, $\tilde{J}_{p+1} \preccurlyeq \overline{E}(p+1)(F_{p+1})$ holds. Since $(\overline{E}(1)(F_1), \dots, \overline{E}(|\overline{E}|)(F_{|\overline{E}|}))$ is an immediate backward job chain, by Lemma 13 we obtain that $\tilde{J}_p \preccurlyeq \overline{E}(p)(F_p)$. Furthermore, since $\tilde{J}_p = \overline{E}(p)(m+1)$ by definition of pc_m^p, we obtain $m+1 \leq F_p$, i.e., $m < F_p$ which contradicts that $m \geq F_p$.

Since $\xi - 1 \geq F_{p+1}$ and for F_{p+1} an immediate backward job chain exists, the immediate backward job chain $\overleftarrow{c}_{\xi-1}^{\overline{E}^{first}_{p+1}}$ can be fully constructed and $pc_{\xi-1}^{p+1}$ exists. We now prove that the length of pc_m^p is upper bounded by the length of $pc_{\xi-1}^{p+1} = (J'_1, \dots, J'_{p+1}, \tilde{J}'_{p+1}, \dots, \tilde{J}'_{|\overline{E}|})$, i.e., we show that $\mathrm{we}(\tilde{J}'_{|\overline{E}|}) - \mathrm{re}(J'_1) \geq \mathrm{we}(\tilde{J}_{|\overline{E}|}) - \mathrm{re}(J_1)$. Specifically, we show $\tilde{J}_{|\overline{E}|} = \tilde{J}'_{|\overline{E}|}$ and $J'_1 \preccurlyeq J_1$ in two individual steps. Important jobs of this proof are illustrated in Figure 5.

Step 1 ($\tilde{J}'_{|\overline{E}|} = \tilde{J}_{|\overline{E}|}$): By definition, we have $\tilde{J}'_{p+1} = \overline{E}(p+1)(\xi) = \tilde{J}_{p+1}$. Since the immediate forward job chains $(\tilde{J}'_{p+1}, \dots, \tilde{J}'_{|\overline{E}|})$ and $(\tilde{J}_{p+1}, \dots, \tilde{J}_{|\overline{E}|})$ both have the same job as the first entry, these job chains coincide. In particular, $\tilde{J}'_{|\overline{E}|} = \tilde{J}_{|\overline{E}|}$.

Step 2 ($J'_1 \preccurlyeq J_1$): Since $(\tilde{J}_p, \dots, \tilde{J}_{|\overline{E}|})$ is an immediate forward job chain, the job $\tilde{J}_{p+1} = \overline{E}(p+1)(\xi)$ is the earliest job with read-event no earlier than $\mathrm{we}(\tilde{J}_p)$. Thus, the read-event of $\overline{E}(p+1)(\xi - 1) = J'_{p+1}$ must be before the write-event of $\tilde{J}_p$, i.e., $\mathrm{we}(\tilde{J}_p) > \mathrm{re}(J'_{p+1})$. For the job of $\overline{E}(p)$ previous to $\tilde{J}_p$, which is J_p, we either have $\mathrm{we}(J_p) > \mathrm{re}(J'_{p+1})$ as well, or J_p is the latest job of $\overline{E}(p)$ with $\mathrm{we}(J_p) \leq \mathrm{re}(J'_{p+1})$. In both cases $J'_p \preccurlyeq J_p$ because $\mathrm{we}(J'_p) \leq \mathrm{re}(J'_{p+1})$. Since $(J'_1, \dots, J'_p)$ and $(J_1, \dots, J_p)$ are both immediate backward job chains and $J'_p \preccurlyeq J_p$, we have $J'_1 \preccurlyeq J_1$ as well by Lemma 13.

Therefore, we obtain $\ell(pc_m^p) = \mathrm{we}(\tilde{J}_{|\overline{E}|}) - \mathrm{re}(J_1) \leq \mathrm{we}(\tilde{J}'_{|\overline{E}|}) - \mathrm{re}(J'_1) = \ell(pc_{\xi-1}^{p+1})$. We already showed that $\xi - 1 \geq F_{p+1}$, hence $\ell(pc_m^p) \leq \ell(pc_{\xi-1}^{p+1}) \leq \sup\left\{\ell(pc_\eta^{p+1}) \,\middle|\, \eta \geq F_{p+1}\right\}$.

Since $m \geq F_p$ is arbitrarily chosen, the relation $\leq$ holds for Equation (7) as $\sup\{\ell(pc_m^p) \mid m \geq F_p\} \leq \sup\left\{\ell(pc_\eta^{p+1}) \,\middle|\, \eta \geq F_{p+1}\right\}$. ◀

We have shown that p-partitioned job chains for different p can be utilized equivalently. The equivalence of MRT and MDA follows directly by applying Lemma 12 multiple times.

▶ **Theorem 14** (Equivalence). *The maximum reaction time and maximum data age are equivalent, i.e.,*

$$\mathrm{MRT} = \sup\{\ell(pc_m^p) \mid m \geq F_p\} = \mathrm{MDA} \tag{8}$$

for all $p \in \left\{1, \dots, |\overline{E}|\right\}$.

6 Implication in Practice

The provided result is much stronger than the observation made in the AUTOSAR timing specification, namely, that "*without over- and undersampling, age and reaction are the same*" [1, Section 7.2, p. 149]. Specifically, in Section 5 we show that MRT and MDA are *always* the same. Since we only assume that each task releases a countably infinite number of jobs, this equivalence holds for a variety of scenarios. This covers, for example:

- Systems with over- and undersampling
- Implicit communication or communication by logical execution time (LET)
- Fixed-priority or Dynamic-priority schedulers
- Tasks scheduled by the Robot Operating System 2 (ROS2), as demonstrated in Section 9
- Synchronized or asynchronized distributed systems
- Periodic or sporadic task systems

This implies that for many industrial applications MRT and MDA can be used and analyzed equivalently. In particular, any guarantee for one metric holds for the other one as well. Furthermore, end-to-end timing specification in industrial systems only needs to consider one instead of two different latencies. This eases the verification of timing constraints.

The model of p-partitioned job chains, introduced in Section 4, and the description of MRT and MDA independent of p can be used to improve end-to-end latency analysis. For instance, a significant speedup of the latency calculation of periodic tasks communicating via LET can be obtained by choosing the right p, as demonstrated in Section 8.

7 Extension to Alternative Definitions

In this section we discuss how the results from Section 5 can be used for alternative definitions of MRT and MDA. In particular, we discuss *reduced* end-to-end latencies used by Dürr et al. [9] and Feiertag et al. [10] in Section 7.1, and we discuss a definition based on *valid* job chains used Günzel et al. [13] in Section 7.2.

7.1 Maximum Reduced Data Age and Reaction Time

The reaction time definition by, e.g., Dürr et al. [9] and Feiertag et al. [10] coincides with the maximum reaction time specified in Section 3. However, their definition of maximum data age does not include the additional time between the last job in the chain and the actuation (at time z'); they use MRDA instead. In this section, we discuss how our results can be transferred to this alternative definition of the end-to-end latencies.

We start by defining the reduced length of a job chain providing a maximum reduced reaction time (MRRT), analogously to the MRDA definition by Günzel et al. [13]. The MRT assumes a similar scenario.

▶ **Definition 15** (Reduced length)**.** For an immediate forward or immediate backward augmented job chain $(z, J_1, \dots, J_{|\overline{E}|}, z')$, we define the reduced length ℓ^* as the length of the intermediate job sequence, i.e., $\ell^*(z, J_1, \dots, J_{|\overline{E}|}, z') = \ell(J_1, \dots, J_{|\overline{E}|})$.

This MRDA definition assumes that the actuation is directly triggered by the write event of the last task in the chain, while the MDA definition assumes that the actuation is based on the calculated value but not triggered directly.

▶ **Definition 16** (Maximum reduced data age)**.** The maximum reduced data age (MRDA) is: $\text{MRDA} = \sup_{m \geq F_{|\overline{E}|}+1} \ell^*(\overleftarrow{ac}_m)$.

Similarly, one can distinguish between the MRT, assuming a scenario where an external cause happens at any time and is registered at the read-event of the first task in the chain, and the MRRT, where read-event and cause happen at the same time.

▶ **Definition 17** (Maximum reduced reaction time (MRRT))**.** We define the maximum reduced reaction time (MRRT) as: $\text{MRRT} = \sup_{m \geq F_1} \ell^*(\vec{ac}_m)$.

Since the reduced length can be computed from the length of a cause-effect chain by removing the additional time at the beginning or at the end, there is the following relation between MRT (MDA, respectively) and MRRT (MRDA, respectively).

▶ **Theorem 18.** *Let $\rho^+(\tau)$ be the* maximal *time between two subsequent read-events of a task τ, and let $\rho^-(\tau)$ be the* minimal *time between two subsequent read-events of a task τ. Then the following two bounds hold:*

$$\text{MRT} - \text{MRRT} \in [\rho^-(\overline{E}(1)), \rho^+(\overline{E}(1))] \tag{9}$$

$$\text{MDA} - \text{MRDA} \in [\rho^-(\overline{E}(|\overline{E}|)), \rho^+(\overline{E}(|\overline{E}|))] \tag{10}$$

We omit the proof, as the relation directly follows from the definition. The above relations can be utilized to transfer the statements from Theorem 14 to MRDA and MRRT. We note that for periodic or sporadic systems MRT = MDA > MRDA holds, i.e., MRT $\neq$ MRDA in all scenarios. In particular, AUTOSAR considers MDA since they observe that "*without over- and undersampling, age and reaction are the same*" [1, Section 7.2, p. 149].

7.2 MRT and MDA Based On Valid Chains

In this section, we extend the equivalence results from the Section 5 to MRT and MDA as defined by Günzel et al. [13] based on *valid* immediate forward and *valid* immediate backward augmented job chains. In particular, valid job chains are defined in [13] as follows.

▶ **Definition 19** (Valid [13])**.** Let $(z, J_1, \dots, J_{|\overline{E}|}, z')$ be an immediate forward or immediate backward augmented job chain. Moreover, let $j \in \mathbb{N}^+$ such that $z = \mathrm{re}(\overline{E}(1)(j))$, i.e., z is at the read-event of the j-th job of $\overline{E}(1)$. Then $(z, J_1, \dots, J_{|\overline{E}|}, z')$ is called *valid*, if and only if $j \geq v^{\overline{E}}$, where $v^{\overline{E}}$ be the smallest integer such that $\mathrm{re}(\overline{E}(1)(v^{\overline{E}}+1)) > \max_{i \in \{1,\dots,|\overline{E}|\}} \mathrm{re}(\overline{E}(i)(1))$.

We denote the MRT and MDA based on valid chains by $\mathrm{MRT}^{\mathrm{V}}$ and $\mathrm{MDA}^{\mathrm{V}}$, respectively. For the MRT based on valid chains, instead of $m \geq F_1$ all $m \in \mathbb{N}^+$ such that $\vec{ac}_m$ is valid are considered. Similarly, for the MDA based on valid chains, instead of $m + 1 \geq F_{|E|}$ all $m \in \mathbb{N}^+$ such that $\overleftarrow{ac}_m$ exists and is valid are considered.

$$\mathrm{MRT}^{\mathrm{V}} = \sup \left\{ \ell(\vec{ac}_m) \,\middle|\, m \in \mathbb{N}^+, \vec{ac}_m \text{ valid} \right\} \tag{11}$$

$$\mathrm{MDA}^{\mathrm{V}} = \sup \left\{ \ell(\overleftarrow{ac}_m) \,\middle|\, m \in \mathbb{N}^+, \overleftarrow{ac}_m \text{ exists and valid} \right\} \tag{12}$$

In the following we discuss the equivalence of $\mathrm{MRT}^{\mathrm{V}}$ and $\mathrm{MDA}^{\mathrm{V}}$. To that end, we first define $V_1, \dots, V_E$ through an immediate backward job chain as follows.

▶ **Definition 20.** Let $V \in \mathbb{N}^+$ such that $\overleftarrow{c}^{\overline{E}}_V$ is the first immediate backward job chain with $\overline{E}(1)(v^{\overline{E}}) \preccurlyeq \overleftarrow{c}^{\overline{E}}_V(1)$, with $v^{\overline{E}}$ defined as in Definition 19. We denote by $V_1, \dots, V_{|\overline{E}|} \in \mathbb{N}^+$ the job number of each job in $\overleftarrow{c}^{\overline{E}}_V$, i.e., $\overleftarrow{c}^{\overline{E}}_V = (\overline{E}(1)(V_1), \dots, \overline{E}(|\overline{E}|)(V_{|\overline{E}|}))$.

With an analogous proof as in Lemma 12 we obtain that for all $p \in \{1, \dots, |\overline{E}| - 1\}$, we have $\sup \{\ell(pc^p_m) \mid m \geq V_p\} = \sup \left\{\ell(pc^{p+1}_m) \,\middle|\, m \geq V_{p+1}\right\}$. As a result,

$$\sup_{m \geq V_1} \ell(\vec{ac}_m) = \sup_{m \geq V_1} \ell(pc^1_m) = \sup_{m \geq V_p} \ell(pc^p_m) = \sup_{m \geq V_{|\overline{E}|}} \ell(pc^{|\overline{E}|}_m) = \sup_{m \geq V_{|\overline{E}|}+1} \ell(\overleftarrow{ac}_m). \tag{13}$$

By the definition of $V_{|\overline{E}|}$ it can be shown that $\mathrm{MDA}^{\mathrm{V}} = \sup_{m \geq V_{|\overline{E}|}+1} \ell(\overleftarrow{ac}_m)$. For $\mathrm{MRT}^{\mathrm{V}}$ one additional immediate augmented forward job chain has to be included to account for the m in $\left\{v^{\overline{E}}, \dots, V_1\right\}$. In particular, $\mathrm{MRT}^{\mathrm{V}} = \max(\ell(\vec{ac}_{v^{\overline{E}}}), \sup_{m \geq V_1} \ell(\vec{ac}_m))$. We conclude that for the alternative definition of MRT and MDA based on valid chains, an equivalence can be obtained up to the first immediate forward augmented job chain, i.e.,

$$\mathrm{MRT}^{\mathrm{V}} = \sup(\ell(\vec{ac}_{v^{\overline{E}}}), \mathrm{MDA}^{\mathrm{V}}), \tag{14}$$

where $\vec{ac}_{v^{\overline{E}}}$ is the first valid immediate forward augmented job chain. The proof of this equivalence is provided in the appendix.

8 Application: Analysis of Chains with Periodic LET Tasks

In this section, we experimentally evaluated the correctness of the relations from Theorem 14 and Theorem 18 by comparing the results to the state of the art by Günzel et al. [13]. In particular, we applied them to periodic tasks scheduled on one electronic control unit (ECU) and communicating under Logical Execution Time (LET) to see if the correct latency values were calculated. Furthermore, we examined whether our approach resulted in faster computation time than the state of the art.

Considered approaches. We compared our approach to Günzel et al. [13].

Approach by Günzel et al. [13] (G21): Let $\Phi(\overline{E}) := \max\left\{\phi_{\overline{E}(i)} \,\middle|\, i = 1, \dots, |\overline{E}|\right\}$ be the maximal phase of the tasks in $\overline{E}$ and let $H(\overline{E}) := \text{lcm}(\left\{T_{\overline{E}(i)} \,\middle|\, i = 1, \dots, |\overline{E}|\right\})$ be the hyperperiod of all tasks in $\overline{E}$. Then under LET the immediate forward and immediate backward augmented job chains with external activity during the interval $[\Phi(\overline{E}) + H(\overline{E}), \Phi(\overline{E}) + 2H(\overline{E}))$ repeat each hyperperiod. In particular only those immediate forward and immediate backward augmented job chains with external activity no later than $\Phi(\overline{E}) + 2H(\overline{E})$ have to be constructed and compared to obtain the MDA, MRDA, MRT and MRRT.

Our approach (Our): According to Theorem 14 it is sufficient to examine the p-partitioned job chains for one $p \in \{1, \dots, |\overline{E}|\}$ to compute the maximum data age. To minimize the calculation time, we considered the task $E(p)$ with the highest period. By this choice of p, the number of constructed job chains can be reduced significantly. MRT and MDA are computed by Theorem 14. By applying Theorem 18, the MRDA and the MRRT can be computed from the MDA and MRT, respectively, since $\rho^-(\tau) = T_\tau = \rho^+(\tau)$ for all LET tasks τ.

Task set generation. We randomly generated 10000 task sets $\mathbb{T}$, each with a random cardinality of $n_{\mathbb{T}} \in [50, 100] \cap \mathbb{N}$. To evaluate the approaches with tasks similar to a real-world application, we generated tasks according to the Automotive Benchmark by Kramer et al. [19]. In particular, for each task τ we drew a period T_τ from the set $\{1, 2, 5, 10, 20, 50, 100, 200, 1000\}$ according to the related share[2] of these periods in [19, Table III, IV and V]. We assumed implicit deadlines, i.e., the relative deadline D_τ was set to the period T_τ, and the phase ϕ_τ was set to 0 for all tasks. Since the read- and write-events under logical execution time are independent from the execution behavior, no execution time of the task was synthesized.

Cause-effect chain generation. For each task set $\mathbb{T}$, we generated a cause-effect chain E according to Kramer et al. [19, Section IV-E]. In particular, we applied the following steps:

1. The number of involved activation patterns $P_E \in \{1, 2, 3\}$, i.e., the number of unique periods of the tasks in the cause-effect chain E, was drawn according to the distribution in [19, Table VI].
2. A set S_E of P_E unique periods was uniformly drawn from the periods in $\mathbb{T}$.
3. For each period in S_E, we drew 2 to 5 tasks at random (without replacement) according to the distribution in [19, Table VII] from the tasks in $\mathbb{T}$ with the respective period. The cause-effect chain E consists of these tasks in random order.

[2] The sum of probabilities in [19] is only 85%. The remaining 15% are reserved for angle-synchronous tasks that we do not consider here. Therefore, all share values were divided by 0.85.

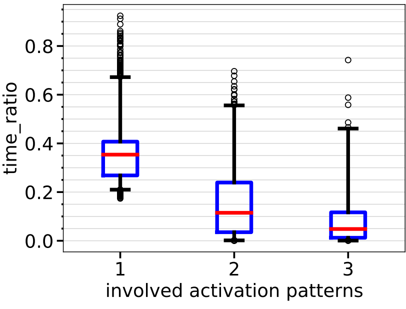


Figure 6 The time_ratio for different number of involved activation patterns. Red line depicts median, blue box 50% of the data, and the whiskers all data except the highest and the lowest 1%.

Table 3 Speed-up for different number of activation patterns.

involved patterns	1	2	3
number values	8109	1436	455
min speed-up	1.08	1.44	1.35
median speed-up	2.83	8.70	20.71
mean speed-up	3.03	48.13	96.27
max speed-up	5.76	1476.02	1623.50

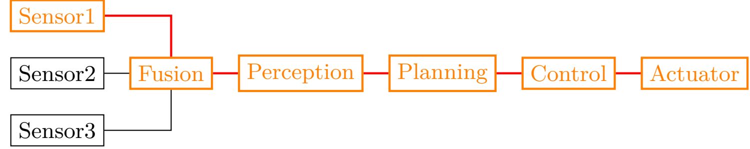


Figure 7 ROS2 basic navigation system.

If there were not sufficient tasks with required period in Step 3, we discarded the set and randomly drew another task set until a cause-effect chain was successfully created.

Evaluation results. For each cause-effect chain, we applied **(G21)** to compute MDA, MRT, MRDA, and MRRT. Additionally, we applied **(Our)** to calculate the same values using Theorem 14 and Theorem 18. For the runtime measurements, we use a machine equipped with 2x AMD EPYC 7742 running Linux, i.e., in total 256 threads with 2.25GHz and 256GB RAM. Each measurement runs on one independent thread.

Observation 1: For all 10000 cause-effect chains all latency values obtained by **(Our)** coincide with the corresponding values obtained by **(G21)**. In particular, the equivalence of MRT and MDA holds for all scenarios, even with over- and undersampling.

Observation 2: Our results can reduce the required time for computing the end-to-end latencies significantly. Specifically, Figure 6 depicts the time ratio, defined by $\frac{\text{time_our}}{\text{time_g21}}$, where time_our and time_g21 is the time needed by **(Our)** and **(G21)**, respectively, to derive all four latency values (considering the minimal runtime over 1000 runs for each of the 10000 cause effect chains). On the x-axis the number of different activation patterns, i.e., the number of different periods in the cause-effect chain, is shown. Additionally, Table 3 shows the speed-up, defined as $\frac{\text{time_g21}}{\text{time_our}}$.

We observe that **(Our)** reduces the required time significantly compared to **(G21)**. In particular, when all tasks in the cause-effect chain have the same period, then the number of computed chains, and hence the runtime, is halved on average. When there is more than one activation pattern, then **(Our)** reduces the number of constructed chains by choosing p such that $E(p)$ is the largest period, and a much larger speed-up is observed such cases.

Table 4 Periods and WCET of the nodes.

Component	Type	Period	WCET
Sensor	time-trig.	$100ms$	$10ms$
Fusion	time-trig.	$100ms$	$100ms$
Fusion	event-trig.	-	$5ms$
Perception	event-trig.	-	$200ms$
Planning	event-trig.	-	$100ms$
Control	event-trig.	-	$50ms$
Actuator	event-trig.	-	$5ms$

Table 5 Measurements of the ROS2 system.

MRT	4.0	$\rho^+(\overline{E}(1))$	0.5
MDA	4.0	$\rho^-(\overline{E}(1))$	0.5
MRRT	3.5	$\rho^+(\overline{E}(\lvert\overline{E}\rvert))$	0.5
MRDA	3.5	$\rho^-(\overline{E}(\lvert\overline{E}\rvert))$	0.5
$\ell(\vec{ac}_{m'})$	3.5		

9 Case Study: ROS2

In this section, we validate Theorem 18 and Theorem 14 considering a basic navigation system, as shown in Figure 7, and apply the scheduling mechanism of the Robot Operating System 2 [23] (ROS2) on a single ECU. The navigation system includes three sensors, whose data is combined and processed for the perception of the environment, planning the route, controlling the vehicle, and sending the output to the vehicle interfaces via an actuator.

A system in ROS2 consists of nodes and topics. Each node represents one component of the system, which can communicate with other nodes via topics, that implement a publish-subscribe architecture. Nodes are represented by tasks and each execution of a node can be considered as the execution of a job. The nodes follow an implicit communication policy, i.e., the read-event of a node is at its start and the write-event of a node is at its finish. Nodes are either time-triggered and event-triggered, i.e., some tasks have an aperiodic behavior. ROS2 has a non-standard custom scheduler that executes tasks instances under a round-robin scheduling approach. Specifically, the scheduler repeatedly collects at most one job of each task for the round, after which it executes them according to their priority. However, the results from this work can still be applied as the basic assumptions (R1) and (R2) for the read- and write-events in Section 2 are met. We do not provide details of the ROS2 scheduling approach here, as we only focus on the timing behavior of the resulting system.

The system depicted in Figure 7 has three sensors whose data is combined in the fusion node. The perception, planning, and control node process the data and supply the actuator node with instructions. For the simulation of the ROS2 scheduling behavior we assume that all jobs have a fixed execution time. Table 4 gives an overview of the WCET of each component and of the period of the time-triggered components.

For the first chain $\overline{E} = (\text{Sensor1}, \dots, \text{Actuator})$ marked in orange in Figure 7, we measured the MRT, MDA, MRRT, and MRDA by determining the longest immediate forward and immediate backward augmented job chains. The measured values are summarized in Table 5, showing that Theorem 14 and Theorem 18 hold.

10 Related Work

The first end-to-end latency analysis for maximum reaction time of a cause-effect chain was developed by Davare et al. [8] in 2007. Later, in 2009, the end-to-end semantics were further extended by Feiertag et al. [10]. Specifically, a formal definition of maximum data age and maximum reaction time is proposed by *timed paths* to distinguish the semantics for the data propagation delay. Based on forward reachability and overwriting of data, *first-to-first*

and *last-to-last* data propagation semantics are proposed, coinciding with the maximum reaction time and the maximum data age, respectively. In addition, they are also known as First-in-First-out (FIFO) and Last-in-Last-out (LILO), respectively [21, 22].

The semantics of maximum reaction time and maximum data age have been widely studied in the literature. Several techniques have been derived through formal verification and compiler to verify the data propagation delay for periodic task systems. Rajeev et al. [24] develop a model-checking based technique to compute the end-to-end delay under both semantics for periodic tasks. Forget et al. [28] propose a language-based approach to verify end-to-end delay at the model level. Klaus et al. [16] extend the design flow of the Real-Time Systems Compiler (RTSC) to take data propagation delay into account but only focus on the maximum data age.

The analysis approaches for end-to-end delays can be classified into two paradigm: active approaches [5, 7, 12, 25], where the release of jobs in the chain is actively controlled depending on the production of data to ensure the data coherence for read and write, and passive approaches [2, 3, 6, 8–11, 13, 17, 24, 26], that focus on how the data is produced and consumed among the jobs of the recurrent tasks in the cause-effect chain provided that the release of jobs of subsequent tasks is independent from the production of data. The approaches proposed in this work are passive ones, but the equivalence of both end-to-end semantics holds in general; that is, the equivalence is not limited to the passive paradigm.

Some recent results focus on end-to-end analysis under the Logical Execution Time (LET) communication model, proposed by Kirsch and Sokolova [15]. Martinez et al. [20] analyzed its effect on the end-to-end delay. Becker et al. [4] further extend their previous analysis from implicit communication to LET. Kordon and Tang [18] develop a framework to calculate the maximum data age under the LET communication based on a given general task dependency graph. Hamann et al. [14] propose a model transformation method to increase the expressiveness of current timing analysis. They address heterogeneous communication semantics including LET. Our analysis is also applicable for LET communication.

11 Conclusion

Over the last years different timing metrics for the end-to-end latency of cause-effect chains were considered. Of particular interest are the maximum reaction time and the maximum data age. AUTOSAR observed that both metrics coincide if no over- and undersampling occurs. However, it was assumed that both metrics differ for the general case.

In this paper, we show the equivalence of maximum reaction time and maximum data age. To this end, we introduce p-partitioned job chains and show that both timing metrics can be determined by constructing p-partitioned job chains for the same p.

The impact of the equivalence is twofold:

1. Analytical literature results for one metric can immediately be used for the other.
2. The choice of $p \in \{2, \dots, |\overline{E}| - 1\}$ allows novel analysis approaches and a reduced runtime of current approaches by reducing the number of necessary job chains under consideration.

We demonstrate the impact of this work by considering the case of periodic tasks communicating under LET. In particular, we show that the end-to-end analysis can be performed up to 1600 times faster in this scenario.

The equivalence holds for almost any scheduling mechanism and even for task systems which do not adhere to the typical periodic or sporadic task model. We support this statement with a case study based on ROS2.

References

1. AUTOSAR. Specification of timing extensions (AUTOSAR CP R21-11). `https://www.autosar.org/fileadmin/user_upload/standards/classic/21-11/AUTOSAR_TPS_TimingExtensions.pdf`, 2021. Accessed: 2022-10-18.
2. Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. Mechaniser-a timing analysis and synthesis tool for multi-rate effect chains with job-level dependencies. In *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2016.
3. Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. Synthesizing job-level dependencies for automotive multi-rate effect chains. In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 159–169, 2016. `doi:10.1109/RTCSA.2016.41`.
4. Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. End-to-end timing analysis of cause-effect chains in automotive embedded systems. *J. Syst. Archit.*, 80(C):104–113, October 2017. `doi:10.1016/j.sysarc.2017.09.004`.
5. Matthias Becker, Saad Mubeen, Dakshina Dasari, Moris Behnam, and Thomas Nolte. A generic framework facilitating early analysis of data propagation delays in multi-rate systems. In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–11, 2017. `doi:10.1109/RTCSA.2017.8046323`.
6. Albert Benveniste, Paul Caspi, Paul Le Guernic, Hervé Marchand, Jean-Pierre Talpin, and Stavros Tripakis. A protocol for loosely time-triggered architectures. In *EMSOFT*, pages 252–265, 2002. `doi:10.1007/3-540-45828-X_19`.
7. Hyunjong Choi, Mohsen Karimi, and Hyoseung Kim. Chain-based fixed-priority scheduling of loosely-dependent tasks. In *International Conference on Computer Design (ICCD)*. IEEE, 2020.
8. Abhijit Davare, Qi Zhu, Marco Di Natale, Claudio Pinello, Sri Kanajan, and Alberto L. Sangiovanni-Vincentelli. Period optimization for hard real-time distributed automotive systems. In *Design Automation Conference, DAC*, pages 278–283, 2007. `doi:10.1145/1278480.1278553`.
9. Marco Dürr, Georg von der Brüggen, Kuan-Hsun Chen, and Jian-Jia Chen. End-to-end timing analysis of sporadic cause-effect chains in distributed systems. *ACM Trans. Embedded Comput. Syst. (Special Issue for CASES)*, 18(5s):58:1–58:24, 2019. `doi:10.1145/3358181`.
10. Nico Feiertag, Kai Richter, Johan Nordlander, and Jan Jonsson. A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. In *Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, 2009.
11. Julien Forget, Frédéric Boniol, and Claire Pagetti. Verifying end-to-end real-time constraints on multi-periodic models. In *ETFA*, pages 1–8, 2017. `doi:10.1109/ETFA.2017.8247612`.
12. Alain Girault, Christophe Prevot, Sophie Quinton, Rafik Henia, and Nicolas Sordon. Improving and estimating the precision of bounds on the worst-case latency of task chains. *IEEE Trans. on CAD of Integrated Circuits and Systems, (Special Issue for EMSOFT)*, 37(11):2578–2589, 2018. `doi:10.1109/TCAD.2018.2861016`.
13. Mario Günzel, Kuan-Hsun Chen, Niklas Ueter, Georg von der Brüggen, Marco Dürr, and Jian-Jia Chen. Timing analysis of asynchronized distributed cause-effect chains. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 40–52, 2021. `doi:10.1109/RTAS52030.2021.00012`.
14. Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. Communication centric design in complex automotive embedded systems. In *Euromicro Conference on Real-Time Systems, ECRTS*, pages 10:1–10:20, 2017. `doi:10.4230/LIPIcs.ECRTS.2017.10`.
15. Christoph M. Kirsch and Ana Sokolova. The logical execution time paradigm. In *Advances in Real-Time Systems*, pages 103–120. Springer, 2012. `doi:10.1007/978-3-642-24349-3_5`.
16. Tobias Klaus, Florian Franzmann, Matthias Becker, and Peter Ulbrich. Data propagation delay constraints in multi-rate systems: Deadlines vs. job-level dependencies. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, pages 93–103. ACM, 2018.

17 Tomasz Kloda, Antoine Bertout, and Yves Sorel. Latency analysis for data chains of real-time periodic tasks. In *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, pages 360–367, 2018. doi:10.1109/ETFA.2018.8502498.

18 Alix Munier Kordon and Ning Tang. Evaluation of the Age Latency of a Real-Time Communicating System Using the LET Paradigm. In Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:20, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECRTS.2020.20.

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

20 Jorge Martinez, Ignacio Sañudo, and Marko Bertogna. End-to-end latency characterization of task communication models for automotive systems. *Real-Time Syst.*, 56(3):315–347, July 2020. doi:10.1007/s11241-020-09350-3.

21 Saad Mubeen, Jukka Mäki-Turja, and Mikael Sjödin. Implementation of end-to-end latency analysis for component-based multi-rate real-time systems in rubus-ice. In *Factory Communication Systems (WFCS), 2012 9th IEEE International Workshop on*, pages 165–168. IEEE, 2012.

22 Saad Mubeen, Jukka Mäki-Turja, and Mikael Sjödin. Translating end-to-end timing requirements to timing analysis model in component-based distributed real-time systems. *SIGBED Review*, 9(4):17–20, 2012. doi:10.1145/2452537.2452539.

23 Open Robotics. Ros 2 documentation: Foxy, May 2022. URL: https://docs.ros.org/en/foxy.

24 AC Rajeev, Swarup Mohalik, Manoj G Dixit, Devesh B Chokshi, and S Ramesh. Schedulability and end-to-end latency in distributed ecu networks: formal modeling and precise estimation. In *International Conference on Embedded Software*, pages 129–138, 2010.

25 Johannes Schlatow and Rolf Ernst. Response-time analysis for task chains in communicating threads. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 245–254, 2016. doi:10.1109/RTAS.2016.7461359.

26 Johannes Schlatow, Mischa Möstl, Sebastian Tobuschat, Tasuku Ishigooka, and Rolf Ernst. Data-age analysis and optimisation for cause-effect chains in automotive control systems. In *IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–9, 2018.

27 Harun Teper, Mario Günzel, Niklas Ueter, Georg von der Brüggen, and Jian-Jia Chen. End-to-end timing analysis in ros2. In *43rd IEEE Real-Time Systems Symposium (RTSS)*, 2022.

28 Rémy Wyss, Frédéric Boniol, Claire Pagetti, and Julien Forget. End-to-end latency computation in a multi-periodic design. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC*, pages 1682–1687, 2013. doi:10.1145/2480362.2480678.

A Proof of Lemma 13

We first restate Lemma 13.

▶ **Lemma 13.** *Let E be a cause-effect chain for the task set $\mathbb{T}$. Moreover, let c be a job chain for E, let $\vec{c}$ be an immediate forward job chain for E, and let $\overleftarrow{c}$ be an immediate backward job chain for E.*

1. *If there exists $i \in \{1, \dots, |E|\}$ such that $\vec{c}(i) \preccurlyeq c(i)$, then $\vec{c}(j) \preccurlyeq c(j)$ for all $j \in \{i, \dots, |E|\}$.*
2. *If there exists $i \in \{1, \dots, |E|\}$ such that $c(i) \preccurlyeq \overleftarrow{c}(i)$, then $c(j) \preccurlyeq \overleftarrow{c}(j)$ for all $j \in \{1, \dots, i\}$.*

Proof. We prove 1) by induction over $j = i, \dots, |E|$.

Initial case. For $j = i$, $\vec{c}(j) \preccurlyeq c(j)$ by assumption.

Induction step. If $\vec{c}(j) \preccurlyeq c(j)$ for $j \in \{i, \dots, |E| - 1\}$, then this means that the write-event of the job $\vec{c}(j)$ is no later than the write-event of the job $c(j)$. Since the read-event of the job $c(j+1)$ is no earlier than the write-event of $c(j)$ by definition of a job chain, the read-event of $c(j+1)$ is also no earlier than the write-event of $\vec{c}(j)$. Since $\vec{c}(j+1)$ is the earliest job with read-event no earlier than the write-event of $\vec{c}(j)$, we conclude that $\vec{c}(j+1) \preccurlyeq c(j+1)$.

We prove 2) by induction over $j = i, \dots, 1$.

Initial case. For $j = i$, $c(j) \preccurlyeq \overleftarrow{c}(j)$ by assumption.

Induction step. If $c(j) \preccurlyeq \overleftarrow{c}(j)$ for $j \in \{i, \dots, 2\}$, then this means that the read-event of the job $c(j)$ is no later than the read-event of the job $c(j)$. Since the write-event of the job $c(j-1)$ is no later than the read-event of $c(j)$ by definition of a job chain, the write-event of $c(j-1)$ is also no later than the read-event of $\overleftarrow{c}(j)$. Since $\overleftarrow{c}(j-1)$ is the latest job with write-event no later than the read-event of $\overleftarrow{c}(j)$, we conclude that $c(j-1) \preccurlyeq \overleftarrow{c}(j-1)$. ◀

B Proof of Extension to Valid Chains

In Section 7.2 it is shown that $\sup_{m \geq V_1} \ell(\vec{ac}_m) = \sup_{m \geq V_{|\overline{E}|}+1} \ell(\overleftarrow{ac}_m)$. It is left to show how this relates to $\mathrm{MDA}^{\mathrm{V}}$ and $\mathrm{MRT}^{\mathrm{V}}$. We start with $\mathrm{MDA}^{\mathrm{V}}$.

▶ **Lemma 14.** *We have* $\mathrm{MDA}^{\mathrm{V}} = \sup_{m \geq V_{|\overline{E}|}+1} \ell(\overleftarrow{ac}_m)$.

Proof. By definition in Equation (12), $\mathrm{MDA}^{\mathrm{V}} = \sup \{\ell(\overleftarrow{ac}_m) \,|\, m \in \mathbb{N}^+, \overleftarrow{ac}_m \text{ exists and valid}\}$. We divide the proof in two steps: $\mathrm{MDA}^{\mathrm{V}} \geq \sup_{m \geq V_{|\overline{E}|}+1} \ell(\overleftarrow{ac}_m)$ and $\mathrm{MDA}^{\mathrm{V}} \leq \sup_{m \geq V_{|\overline{E}|}+1} \ell(\overleftarrow{ac}_m)$.

Step 1 ($\mathrm{MDA}^{\mathrm{V}} \geq \sup_{m \geq V_{|\overline{E}|}+1} \ell(\overleftarrow{ac}_m)$). We know that $\overleftarrow{ac}_{V_{|\overline{E}|}+1}$ is composed of $z' = \mathrm{we}(\overline{E}(|\overline{E}|)(V_{|\overline{E}|} + 1))$, the immediate backward job chain $\overleftarrow{c}^{\overline{E}}_{V_{|\overline{E}|}}$, and $z \geq \mathrm{re}(\overline{E}(1)(v^{\overline{E}}))$. Therefore, $\overleftarrow{ac}_{V_{|\overline{E}|}+1}$ is valid. Consequently, all $\overleftarrow{ac}_m$ with $m \geq V_{|\overline{E}|} + 1$ are valid and $\mathrm{MDA}^{\mathrm{V}} \geq \sup_{m \geq V_{|\overline{E}|}+1} \ell(\overleftarrow{ac}_m)$ holds.

Step 2 ($\mathrm{MDA}^{\mathrm{V}} \leq \sup_{m \geq V_{|\overline{E}|}+1} \ell(\overleftarrow{ac}_m)$). We prove this step by contradiction and assume that there exists a valid $\overleftarrow{ac}_m = (z, J_1, \dots, J_{|\overline{E}|}, z')$ with $m < V_{|\overline{E}|} + 1$. In that case $J_{|\overline{E}|}$ must be a job *before* $\overline{E}(V_{|\overline{E}|})$. Since by Definition 20, $\overleftarrow{c}^{\overline{E}}_{V_{|\overline{E}|}}$ is the first chain with $\overline{E}(1)(v^{\overline{E}}) \preccurlyeq \overleftarrow{c}^{\overline{E}}_{V}(1)$, the job J_1 must be earlier than $\overline{E}(1)(v^{\overline{E}})$. Hence, $\overleftarrow{ac}_m = (z, J_1, \dots, J_{|\overline{E}|}, z')$ is not valid, which is a contradiction. ◀

For the MRT, we may need to account for additional immediate forward augmented job chains before V_1. By definition, valid immediate forward augmented job chains $\vec{ac}_m$ with $v^{\overline{E}} \leq m < V_1$ include the immediate forward job chain $\vec{c}^{\overline{E}}_{m+1}$ with $v^{|\overline{E}|} < m + 1 \leq V_1$. The following lemma examines those immediate forward job chains further, showing that all of them end at the same job, which means that the longest $\vec{ac}^1_m$ with $m < F_1$ is the first one.

▶ **Lemma 15.** *Let $m \in \mathbb{N}^+$ with $v^{\overline{E}} \leq m \leq V_1$. The last entry of the m-th immediate forward job chain $\vec{c}^{\overline{E}}_m$ is the job where $\overleftarrow{c}^{\overline{E}}_V$ ends, i.e., $\vec{c}^{\overline{E}}_m(|\overline{E}|) = \overline{E}(|\overline{E}|)(V_{|\overline{E}|})$.*

Proof. Let $\xi \in \mathbb{N}^+$ such that $\vec{c}^{\overline{E}}_m(|\overline{E}|) = \overline{E}(|\overline{E}|)(\xi)$. In the following, we show that $\xi = V$.

Since there exist job chains with last job $\overline{E}(|\overline{E}|)(\xi)$, this means that $\overleftarrow{c}^{\overline{E}}_\xi$ exists as well because for the backward construction of that chain in each step a job can be chosen. Moreover, since $\overleftarrow{c}^{\overline{E}}_\xi$ is immediate backward and $\vec{c}^{\overline{E}}_m$ is a job chain with the same job as last entry, by Lemma 13 $\vec{c}^{\overline{E}}_m(1) \preccurlyeq \overleftarrow{c}^{\overline{E}}_\xi(1)$. Hence, $E(1)(v^{|\overline{E}|}) \preccurlyeq \overleftarrow{c}^{\overline{E}}_\xi(1)$. Since $\overleftarrow{c}^{\overline{E}}_V$ is the earliest immediate backward job chain with this property, we obtain $\xi \geq V$.

Since $\vec{c}^{\overline{E}}_m(1) \preccurlyeq \overline{E}(|\overline{E}|)(V_1) = \overleftarrow{c}^{\overline{E}}_V(1)$ and $\vec{c}^{\overline{E}}_m$ is immediate forward, we can apply Lemma 13 and obtain $\vec{c}^{\overline{E}}_m(|\overline{E}|) \preccurlyeq \overleftarrow{c}^{\overline{E}}_V(|\overline{E}|) = \overline{E}(|\overline{E}|)(V_{|\overline{E}|})$, i.e., $\xi \leq V$. ◀

Since the last job of every $\vec{c}^{\overline{E}}_m$ with $v^{|\overline{E}|} \leq m \leq V_1$ is $\overline{E}(|\overline{E}|)(V_{|\overline{E}|})$, the last job of every $\vec{ac}_m$ with $v^{|\overline{E}|} \leq m < V_1$ is $\overline{E}(|\overline{E}|)(V_{|\overline{E}|})$ as well. Furthermore, since the $\vec{ac}^1_m$ with the lowest m has the earliest first job, it is the longest one, i.e., to obtain $\mathrm{MRT}^{\mathrm{V}}$ only the first valid immediate forward augmented job chain is considered in addition to $\sup_{m \geq V_1} \ell(\vec{ac}_m)$.

▶ **Lemma 16.** *We have $\mathrm{MRT}^{\mathrm{V}} = \max(\ell(\vec{ac}_{v^{\overline{E}}}), \sup_{m \geq V_1} \ell(\vec{ac}_m))$, where $\vec{ac}_{v^{\overline{E}}}$ is the first valid immediate forward augmented job chain.*

Proof. By definition, $V_1 \geq v^{\overline{E}}$. Moreover, 1-partitioned job chains $\vec{ac}_m$ are valid if and only if $m \geq v^{\overline{E}}$. Therefore, $\mathrm{MRT}^{\mathrm{V}}$ is the maximum of $\sup\{\ell(\vec{ac}_m) \,|\, m \geq V_1\}$ and $\max\left\{\ell(\vec{ac}_m) \,\middle|\, V_1 > m \geq v^{\overline{E}}\right\}$. Moreover, by Lemma 15 $\max\left\{\ell(\vec{ac}_m) \,\middle|\, V_1 > m \geq v^{\overline{E}}\right\}$ is either 0 if $V_1 = v^{\overline{E}}$, or it is the length of $\vec{ac}_{v^{\overline{E}}}$. ◀

We summarize the relation between $\mathrm{MDA}^{\mathrm{V}}$ and $\mathrm{MRT}^{\mathrm{V}}$.

▶ **Proposition 17.** *The maximum reaction time $\mathrm{MRT}^{\mathrm{V}} = \sup(\ell(\vec{ac}_{v^{\overline{E}}}), \mathrm{MDA}^{\mathrm{V}})$, where $\vec{ac}_{v^{\overline{E}}}$ is the first valid immediate forward augmented job chain.*

The $\mathrm{MRT}^{\mathrm{V}}$ can also be formulated as $\mathrm{MRT}^{\mathrm{V}} = \max(\ell(pc^1_{v^{\overline{E}}}), \mathrm{MDA}^{\mathrm{V}})$, where $pc^1_{v^{\overline{E}}}$ is the first valid immediate forward augmented job chain, since by Lemma 10 $\ell(pc^1_{v^{\overline{E}}}) = \ell(\vec{ac}_{v^{\overline{E}}})$. Moreover, $\mathrm{MDA}^{\mathrm{V}}$ can be computed through p-partitioned job chains as $\mathrm{MDA}^{\mathrm{V}} = \sup_{m \geq V_p} \ell(pc^p_m)$ for any $p \in \{1, \ldots, |\overline{E}|\}$.

A New Perspective on Criticality: Efficient State Abstraction and Run-Time Monitoring of Mixed-Criticality Real-Time Control Systems

Tim Rheinfels
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Maximilian Gaukler
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Peter Ulbrich
TU Dortmund, Germany

Abstract

The increasing complexity of real-time systems, comprising control tasks interacting with physics and non-control tasks, comes with substantial challenges: meeting various non-functional requirements implies conflicting design goals and a pronounced gap between worst and average-case resource requirements up to the overall timeliness being unverifiable. Mixed-criticality systems (MCS) is a well-known mitigation concept that operates the system in different criticality levels with timing guarantees given only to the subset of critical tasks. However, in many real-world applications, the criticality of control tasks is tied to the system's physical state and control deviation, with safety specifications becoming a crucial design objective. Monitoring the physical state and adapting scheduling is inaccessible to MCS but has been dedicated mainly to control engineering approaches such as self-triggered (model-predictive) control. These, however, are hard to integrate with scheduling or expensive at run-time.

This paper explores the potential of linking both worlds and elevating the physical state to a criticality criterion. We, therefore, propose a dedicated state estimation that can be leveraged as a run-time monitor for criticality mode changes. For this purpose, we develop a highly efficient one-dimensional state abstraction to be computed within the operating system's scheduling. Furthermore, we show how to limit abstraction pessimism by feeding back state measurements robustly. The paper focuses on the control fundamentals and outlines how to leverage this new tool in adaptive scheduling. Our experimental results substantiate the efficiency and applicability of our approach.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Embedded and cyber-physical systems; Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases Real-time Control, Mixed-Criticality, Switched Systems, State Monitoring

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.11

Supplementary Material *Software (ECRTS 2023 Artifact Evaluation approved artifact)*: https://doi.org/10.4230/DARTS.9.1.1

1 Introduction

For quite some time, we have been facing a rapidly increasing complexity of real-time (control) systems, such as the proliferation of autonomous driving and robotic applications. These are characterized by high performance requirements with numerous control tasks, which interact with physics in closed loops, and non-control tasks executed along with them, forming a heterogenous task set. Meeting all the tasks' non-functional requirements (e.g., QoC, performance, costs) implies conflicting design goals and a pronounced gap between worst and

© Tim Rheinfels, Maximilian Gaukler, and Peter Ulbrich;
licensed under Creative Commons License CC-BY 4.0

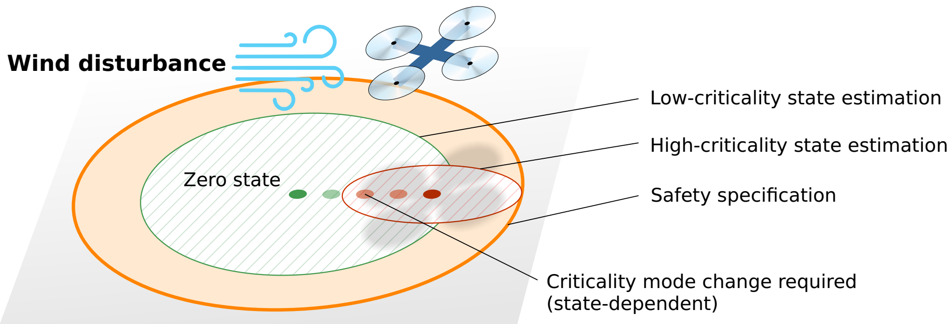


Figure 1 Illustration of a system with safety constraints: the UAV must stay within given bounds (filled area). Compliance is challenged by disturbances (i.e., wind), which, in the worst case, can only be rejected in *hi* mode. Note that criticality change depends on the physical state (i.e., position and velocity vectors). State estimation (hatched areas) varies between modes as laxer scheduling implies more pessimism due to less stringent control.

average-case resource requirements up to the overall timeliness and safety being unverifiable. This gap is a well-known challenge for heterogeneous task sets with a large body of work on mitigation techniques, particularly the concept of mixed-criticality systems (MCS) first introduced by Vestal [42]. It facilitates adapted design and verification of task sets with varying requirements, such that task parameters (e.g., WCETs, periods) become dependent on a criticality level. As a result, MCSs operate in different modes, transparently monitored (e.g., execution time) and enforced by the operating system, with timing guarantees given only to the subset of critical (control) tasks; MCS typically have no intended bearing on the physical side of the system.

As an orthogonal approach, conflicting design goals can be eased by reducing the timing requirements of control tasks based on the inherent robustness of controllers. The latter is exploited to relax, for example, periodicity [11] or deadline adherence [32] while guaranteeing stability. A popular approach is, for example, (m, K)-firm scheduling [21], which requires at least m out of K consecutive job releases to meet their deadlines in terms of a weakly-hard [7] real-time design. Consequently, the controller consistently provides the best possible performance, depending on the utilization, without failing in the worst case [4]. However, this behavior, in turn, represents significant over-provisioning on average. Extended variants of these co-design approaches also allow the controller to adapt to specific job drop scenarios [29] or switch between different controller modes [15], further softening timing constraints. We will discuss a large body of related work in Section 5.

1.1 Problem Statement

In real-world applications, a crucial design objective [2, 6, 38] is to ensure that a system will remain within given safety specifications (e.g., maximum deviation from equilibrium) even at the worst assumed disturbance. Here, the necessary system response's stringency (e.g., temporal) varies with the state-dependent deviation (e.g., control error). Although the control-aware scheduling approaches above can exploit varying demands, they typically aim solely at control stability and thus cannot guarantee compliance with such safety specifications. Note that, in this context, stability only implies that the system reaches equilibrium in the

absence of disturbances without any assertions of the largest possible deviation they inflict. On the other hand, a control task's criticality could be characterized by the risk of a safety violation. However, typically, MCS cannot meet such requirements since they do not assess criticality based on control error but on timing, whereby only an indirect relationship exists between temporal and physical properties.

In the following, we use the UAV in Figure 1 as an illustrative example to make our point. It must follow a given flight trajectory, perform collision avoidance, and simultaneously fulfill other mission objectives, all on a shared resource-limited computing platform. In addition, per safety specifications, the UAV must settle and stay within a one-meter radius around its reference (filled area) to avoid crashing into obstacles even in the presence of wind disturbance. On the other hand, higher accuracy inside this envelope is generally desirable yet not mandatory. Finally, assuming worst-case wind disturbances to be a rare failure scenario, we can design the system with a low (lo) and a high (hi) criticality mode. The former facilitates sharing and uses relaxed parameters, for example, larger periods, lax WCETs, or an (m, K) execution model. The latter provides strong isolation, high assurance, and fault rejection. In the given scenario, the maximum wind disturbance drives the UAV off. For some time (i.e., first two states), the relaxed execution behavior in low-criticality mode remains tolerable due to inertia. However, depending on the UAV's physical state, notably its position and velocity vectors, switching to high-criticality mode at a precise moment (i.e., third state) is imperative to avoid the momentum carrying the UAV outside the safety limit. Our concern is that monitoring timing parameters is insufficient to identify the criticality change. Cheng et al. [13] consequently propose using the quality of control as a changeover criterion, however, assuming uncorrelated noise and resorting to a probabilistical model, which is hardly suited for our scenario.

In control engineering, some approaches tie criticality to the current system state. For example, in self-triggered control (STC) methods [23], the controller itself computes the next necessary control instant based on the physical state. Thus, STC releases control tasks sporadically whenever the system threatens to deviate from equilibrium too far, establishing a form of adaptive scheduling. Self-triggered model predictive control (MPC) schemes go even one step further by controlling the process and computing the next sampling instant [25] while enforcing state and control value constraints in the presence of disturbances [28]. However, by unifying state estimation and scheduling, these approaches are inherently incompatible with traditional scheduling methods: jobs must be executed with high timing adherence once released. Thus we lose support for heterogeneous task sets and adaptive scheduling (e.g., criticality-dependent task parameters).

We can address the problem by separating state estimation from controller execution. For the UAV example, a safe decision must be made for a criticality mode. Since the execution conditions in low-criticality mode are more relaxed, the reachable physical state for a given observation period is larger: with less control, the drone can drift further. Conversely, the reachable set is much tighter in high mode. However, robustly observing and predicting the system's physical state is challenging. Established approaches, for example, a set-valued Kalman filter [30], can robustly locate the physical state up to a time-varying ellipsoid. Yet, computational costs have thwarted their use as run-time monitoring for mixed-criticality or other adaptive scheduling approaches at the operating system (OS) level. Our evaluation in Section 4 highlights the overheads associated with set-valued state estimation.

The fundamental issue addressed in this paper is to facilitate the use of the physical state of control systems as a general criticality and scheduling criterion at the OS level. *Our approach offers significant advantages over application-level solutions, such as isolation, standard interfaces, and, most importantly, seamless integration with temporal monitoring*

Figure 2 Overview of our approach: implementing state monitoring at the OS level facilitates strong isolation, an application-independent interface, and seamless integration with traditional (MC) execution-time monitoring and scheduling. State abstractions are parameterized by design-time analysis for safe mode switching.

and scheduling of all the system's tasks. As a first step, we focus on low-overhead yet robust (i.e., sound) state estimation at run time and design-time verification of it as a prerequisite for this aim. Therefore, we derive the necessary control engineering background and provide an interface to real-time scheduling. At this point, we emphasize that in this paper, we are concerned with the basic methodology rather than a specific scheduling approach; we believe our approach can serve a wide range of existing scheduling techniques.

Conceptually, we seek a time-dynamic and one-dimensional *state abstraction* v_k:

$$v_{k+1} = \rho_{\sigma_k} v_k + \beta_{\sigma_k}, \quad 0 \leq v_k \overset{!}{\leq} v_{max} \quad \forall k \in \mathbb{N}_0.$$

While the quantity v_k provides an upper bound on the system's state, ρ_{σ_k} denotes the time-varying decay rate (i.e., a lower bound for speed of convergence) and β_{σ_k} the disturbance term (i.e., how the system deviates from equilibrium due to disturbances). Finally, v_{max} gives the application-specific safety requirement (i.e., the maximum permissible deviation from the nominal state). The robust prediction v_{k+1} provides a unified framework to monitor the physical state in the OS-level at run time, for example, allowing us to identify the changeover instant between both modes in our example from Figure 1. However, this seemingly simple requirement is tied to a number of fundamental challenges.

Challenge 1: Run-Time Monitoring by an Easy-to-Compute State Abstraction

Reachability analysis using time-variant sets can yield precise results. However, for example, computing the distance between two sets is prohibitively expensive at run time. Therefore, we aim for a *state abstraction* (i.e., sound estimation) that reliably predicts future violations of the control's safety specification, serving as a run-time monitor for criticality modes. Furthermore, the abstraction should facilitate a simple application-independent interface and simultaneously be economical to compute.

Our Approach. Based on [19], we develop a one-dimensional *state abstraction* (called blind abstraction) for switched linear control systems. This allows us to restrict the prediction horizon to one time step (e.g., time slice of the scheduler), permitting a timely mode change and, if necessary, an adaptation of the control regime to reject worst-case disturbance safely. Furthermore, reducing the state dimensions and prediction horizon grants low run-time overhead and a simple interface. Figure 2 outlines our approach and illustrates how the state abstractions fit the OS kernel and scheduling.

Challenge 2: Selective Reduction of State Abstraction Overestimation

Abstraction reduces complexity but is typically accompanied by considerable pessimism. Accordingly, we must ensure appropriate accuracy by leveraging the available state information. Furthermore, executing modes of lower criticality causes the state information to become increasingly uncertain. Consequently, the abstraction's overapproximation could lead to premature anticipation of safety violations causing false changeovers. Such behavior would jeopardize its use as a monitoring function.

Our Approach. We introduce the concept of an *observer abstraction*, which mitigates the statically inferred worst-case pessimism of the abstraction by robustly feeding uncertain measurements of the system state into the (formerly blind) abstraction.

Challenge 3: Design-Time Analysis for Efficiently Choosing the Parameters

Finally, we must ensure that all abstractions fit the system and are safe to use. The challenge is to tailor our abstractions at design time to a system with multiple criticality levels and system parameters for which potentially only the most stringent one guarantees adherence to the safety specification. Simply put, starting with the worst-case observer abstraction for the worst case, we must parameterize the observers for the lower criticality levels such that the system remains in optimistic execution with high probability and that each criticality level has a reasonable operating range.

Our Approach. Based on semidefinite programming, we develop a heuristic which optimizes all design-time parameters for the average case while still guaranteeing safety in the worst case. This analysis, as shown in Figure 2, is used to parameterize the state abstractions offline based on the control applications' specification and criticality modes.

1.2 Contribution and Outline

This paper makes the following four contributions: (1) Extend the convergence rate abstractions presented in our previous work [19] to a more general linear system model. (2) An observer-based approach to mitigate over-estimation in those state abstractions by feeding back uncertain measurements robustly. (3) Determine safe change-over points for criticality and implement a prototypical adaptive run-time policy. (4) A design-time heuristic to parameterize the observer abstractions' parameters.

The remainder is organized as follows. In Section 2, we detail our system model and present relevant background information. Our approach on efficient and robust state monitoring to facilitate run-time adaptivity is presented informally as well as formally in Section 3 while the necessary mathematical proofs are detached to Appendix A. Section 4 provides experimental results as part of a case study. Section 5 discusses related work. Finally, we conclude our work in Section 6.

2 System Model

In this section, we define the paper's notation, describe the mathematical foundations, and formulate our control-theoretic system model. Further, we outline integrating the latter into a real-time system's scheduling and schedulability analysis.

2.1 Notation, Ellipsoids, and Relevant Norms

We denote both scalars $\rho \in \mathbb{R}$ and vectors $x \in \mathbb{R}^n$ as lowercase, matrices $A \in \mathbb{R}^{n\times n}$ as uppercase letters. The identity matrix of appropriate size is referred to as I. We use A^T and A^{-1} for a matrix's transposed and inverse and denote a symmetric and positive definite (s.p.d.) / semidefinite (s.p.sd.) matrix $P = P^T$, $x^T Px > 0 \forall x \neq 0$ / $x^T Px \geq 0 \forall x$ as $P \succ 0$ / $P \succeq 0$. For an s.p.d. matrix $P \succ 0$, we refer to its lower Cholesky decomposition as $P^{\frac{1}{2}}$, i.e. $P = P^{\frac{1}{2}} P^{\frac{T}{2}}$. Given two symmetric matrices $P = P^T, Q = Q^T$, we abbreviate $x^T Px \geq x^T Qx \ \forall x$ as $P \succeq Q$.

Based upon the Euclidean vector norm $||x||$, we define the P-weighted norm $||x||_P := ||P^{\frac{T}{2}} x||$ with $P \succ 0$. The inequality $||x||_P \leq 1$ defines a non-degenerate and centered ellipsoid.

Given the spectral norm $||\cdot||$, we generalize the vector norms $||\cdot||_P$ to matrix norms $||A||_{PQ}$ in terms of the s.p.d. weight matrices $P, Q \succ 0, P \in \mathbb{R}^{n\times n}, Q \in \mathbb{R}^{m\times m}$. With correctly sized x and s.p.d. $W \succ 0$, the following hold [46, pp. 34–35]:

$$||A||_{PQ} := \max_{||x||_Q=1} ||Ax||_P \qquad \textit{(Definition as Operator Norm)} \tag{1a}$$

$$||A||_{PQ} = ||P^{\frac{T}{2}} A Q^{-\frac{T}{2}}|| \qquad \textit{(Relation to Spectral Norm)} \tag{1b}$$

$$||Ax||_P \leq ||A||_{PQ} ||x||_Q \qquad \textit{(Consistency)} \tag{1c}$$

$$||AB||_{PQ} \leq ||A||_{PW} ||B||_{WQ} \qquad \textit{(Generalized Consistency)} \tag{1d}$$

The last equation (1d) is a generalization of [46] (Wang et al. assume P=Q). Given the sub-multiplicativity ($||AB|| \leq ||A||\ ||B||$) of the spectral norm, the proof is:

$$||AB||_{PQ} \overset{(1b)}{=} ||P^{\frac{T}{2}} A \underbrace{W^{-\frac{T}{2}} W^{\frac{T}{2}}}_{=I} B Q^{-\frac{T}{2}}|| \leq ||P^{\frac{T}{2}} A W^{-\frac{T}{2}}||\ ||W^{\frac{T}{2}} B Q^{-\frac{T}{2}}|| \overset{(1b)}{=} ||A||_{PW} ||B||_{WQ}. \tag{2}$$

Further, we will be using the defining properties of (semi-) norms, i.e., for any norm $||\cdot||$, vectors x, y, and scalar α

$$||\alpha x|| = |\alpha|\ ||x|| \qquad \textit{(Homogeneity)} \tag{3a}$$

$$||x|| \geq 0 \qquad \textit{(Non-negativity)} \tag{3b}$$

$$||x + y|| \leq ||x|| + ||y|| \qquad \textit{(Triangle Inequality)} \tag{3c}$$

The notation $a \overset{!}{\leq} b$ indicates that we must ensure $a \leq b$ by applying adequate measures.

2.2 System Model

This section defines the system model used in the remainder of the paper. For each control application, we assume a switched linear plant and controller with a finite number of modes. All states, potentially including sensor and actuator values, are summarized into an extended linear system. This model allows for complex real-time behavior, e.g., omitting controller executions or sampling only a part of the available sensors [45, 44, 18]. Adding safety outputs allows us to both disregard states (e.g., controller states or measurements) aiding the analysis or impose additional constraints (e.g., limits on the control signals). The safety goal is to keep these outputs inside of a given ellipsoid. The dynamics are influenced by an ellipsoidally bounded process uncertainty, as are the uncertain measurements. The initial state is also constrained to an ellipsoid. The state-space model reads

$$\begin{cases} x_{k+1} = A_{\sigma_k} x_k + G_{\sigma_k} d_k & \textit{(Dynamics)} \quad (4a)\\ y_k = C_{\sigma_k} x_k + H_{\sigma_k} z_k & \textit{(Measurements)} \quad (4b)\\ ||d_k||_{D_{\sigma_k}} \leq 1,\ ||z_k||_{Z_{\sigma_k}} \leq 1 & \textit{(Disturbances)} \quad (4c)\\ ||x_0||_{X_0} \leq 1 & \textit{(Initial State)} \quad (4d)\\ s_k := C_s x_k & \textit{(Safety Outputs)} \quad (4e)\\ ||s_k||_S \overset{!}{\leq} 1 & \textit{(Safety Specification)} \quad (4f)\\ \sigma_k \in \Sigma,\ |\Sigma| = n_\Sigma & \textit{(Switching Sequence)} \quad (4g) \end{cases}$$

$\forall k \in \mathbb{N}_0$, where $x_k \in \mathbb{R}^{n_x}$, $d_k \in \mathbb{R}^{n_d(\sigma_k)}$, $z_k \in \mathbb{R}^{n_z(\sigma_k)}$, $y_k \in \mathbb{R}^{n_y(\sigma_k)}$, and $s_k \in \mathbb{R}^{n_s}$ are the system state, process and measurement disturbances, as well as the measurements and the safety outputs. All matrices should be of appropriate (possibly zero) size. The ellipsoids (if not of zero dimension) are assumed to be non-degenerate, i.e. D_σ, Z_σ, X_0, and $S \succ 0\ \forall \sigma \in \Sigma$.

Given the system model, we define the *worst-case disturbance* as the values of x_0, d_k and z_k, which steer the safety outputs the closest to the specification boundary for a given switching sequence σ_k, i.e., $\sup_{x_0 \in X_0, d_k \in D_{\sigma_k}, z_k \in Z_{\sigma_k}, l \in \mathbb{N}_0} ||s_l||_S$. These values can not readily be obtained as the optimization horizon is infinite.

For the sake of clarity, we omitted any reference signals $w_k \in \mathbb{R}^{n_x}$, $x_{r,k+1} = A_{\sigma_k} x_{r,k} + w_k$ and $y_{r,k} = C_{\sigma_k} x_{r,k}$. They can, however, be incorporated by substituting x_k and y_k in the system above by their corresponding error signals $\Delta x_k := x_k - x_{r,k}$ and $\Delta y_k := y_k - y_{r,k}$. The abstraction then bounds the tracking error, which leads to the same analytical results as below due to the system's linearity.

We assume that, except for potential reference signals, a single application implements the above controller. Every criticality mode σ inside this application has its own task set. Our approach is agnostic to the underlying scheduling scheme as long as the signals meet their sampling points. This can, e.g., be achieved using the logical execution time (LET) paradigm. Multiple independent control applications can be used if the scheduling algorithm guarantees their timeliness.

3 Approach

With our system model defined, we proceed by detailing our state abstraction approach in this section. In an informal description in Section 3.1, we first recapitulate the intent of state abstractions, describe how they interact with the real-time control system, and discuss what sets them apart from well-known state observers. After Section 3.2 extends the convergence rate abstractions presented in our previous work [19] to our system model, Section 3.3 shows how to incorporate measurements to reduce their pessimism. In Section 3.4, we derive the run-time monitoring algorithm and a prototypical approach for guaranteed safe switching decisions. Finally, Section 3.5 describes a heuristic for choosing the design-time parameters. To better convey the ideas, we detach the necessary formal proofs to Appendix A and reference them as needed.

3.1 Informal Description

We seek a method of monitoring the system model presented in Section 2.2 at run time with respect to safety, i.e., how close the physical state is to its specified limits and, in extension, predict points where criticality changes. When reaching such a point, the scheduler

must alter the switching sequence σ_k to avoid specification violations. Note that safety is a stronger condition than mere exponential stability for which the literature already provides solutions [45].

Our approach provides *(state) abstractions* v_k, which are one-dimensional and positive dynamic systems. An abstraction's state v_k is an upper bound on the control system's physical state expressed in the *analysis norm* $||\cdot||_P$, i.e., $0 \leq ||x_k||_P \leq v_k \forall k \in \mathbb{N}_0$. As a geometric interpretation: instead of the ellipsoidal observers, which track how the system's dynamics recursively alters the initial state ellipsoid at run time, we choose a fix-shaped analysis ellipsoid parametrized by $P \succ 0$ at design time, leaving the scaling factor v_k as the only run-time parameter. The abstraction dynamics, i.e., the coefficients ρ_σ and β_σ mentioned above, are chosen such that the scaled analysis ellipsoid is guaranteed to contain the system's state for the switching sequence σ_k at hand. Tightly over-approximating the initial state and safety ellipsoids yields the two static factors v_0 and v_{max}. While the former is the initial value for the abstraction state, the latter links it to the safety goal: for our approach to guarantee safety, $v_k \leq v_{max}$ must be enforced at all times by choosing the switching sequence accordingly.

The intermediate steps introduce analysis pessimism as the ellipsoids are usually not aligned in practice. Further, the exponential envelope curve v_k has to incorporate all transient behavior and overshoot in the system state even under worst-case disturbance. Despite the high pessimism, the lack of overshoot allows us to derive scheduling decisions at a single time step prediction horizon. We will later introduce *observer abstractions* to alleviate the pessimism by feeding back uncertain state measurements. The analysis presented in this section expresses the scheduling horizon in terms of a *maximum safe value* v^*_σ for every mode $\sigma \in \Sigma$, i.e., the highest value of v_k for which the corresponding mode is sufficient. Consequentially, modes with higher v^*_σ indicate higher physical criticality as they allow the control loop to withstand more severe conditions. If $v^*_\sigma \geq v_{max}$ holds for a mode σ, it is a *safe mode* as applying it guarantees the system to remain safe at all times. We call the excess $v^*_\sigma - v_{max}$ the *safety margin.*

While set-valued observers are available for our system model [16, 30], they perform the aforementioned ellipsoidal analysis at run time, yielding a precise bound on the system's state. However, choosing the switching sequence σ_k requires further assessing the system's reachability at run time which is expensive as the ellipsoids are not aligned in general. In contrast, their non-robust counter parts (e.g., the classical Luenberger observer) give static guarantees on the estimation error's decay. Going down this path will eventually lead to similar results as presented in this paper.[1]

Since our analysis is sound as long as the IO timing is met, every (mixed-criticality) scheduling scheme suffices when choosing the control application's criticality high enough. Conversely, the abstraction introduces some overhead. The value of v_k has to be updated every time step given (8a), potentially reusing the measurements from the controller. Then, the scheduler has to select one of the feasible modes defined by (13) for the upcoming time step and possibly reconfigure the real-time system accordingly using a mode change. These operations look the same for every control loop and thus share a common interface. As shown in Section 4, they are also cheap to compute. Thus, they lend themselves to be executed as privileged operations inside the RTOS's kernel.

[1] We omitted the analysis for the sake of brevity. As an outline: compute the estimate $\hat{x}_k$ as usual and bound the uncertainty ellipsoid just as the blind abstractions bound the state. Then, $||\hat{x}_k||_P - v_k \leq ||x_k||_P \leq ||\hat{x}_k||_P + v_k \forall k$.

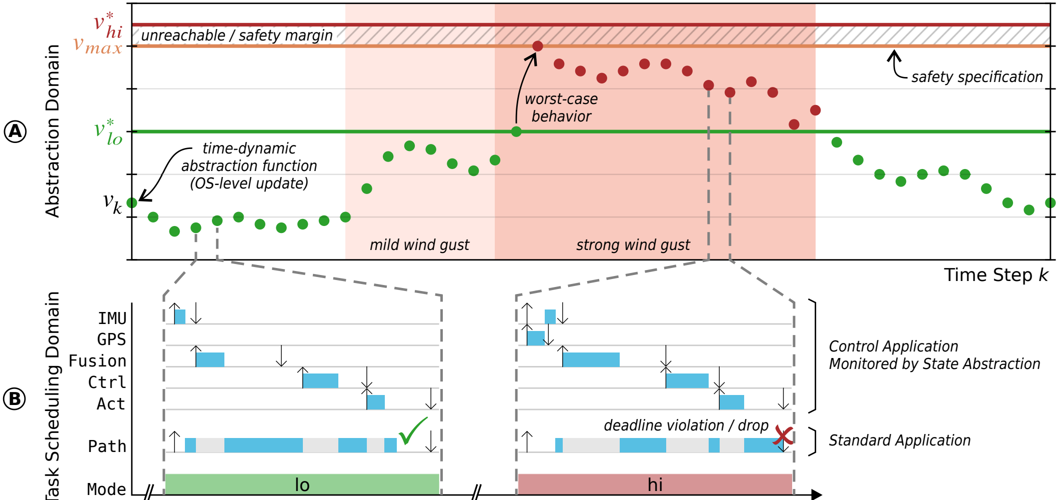


Figure 3 Example task set illustrating the application of state abstractions for mixed-criticality real-time control systems. (A) shows the time-dynamic state abstraction monitoring a UAV under the influence of varying wind conditions. (B) illustrates the resulting schedule with criticality change.

We exemplify our approach using the UAV from above as an illustrative example. The safety goal is to keep the UAV's position p_k inside a one-meter radius around its reference trajectory. Thus, the safety outputs omit all other state components such as the UAV's velocities, i.e., $s_k = p_k$. $S = I$ models the 1m radius. Figure 3 (B) shows the exemplary task set: the flight controller application carries out the state reconstruction by linearly fusing inertial (IMU) and GPS measurements, then computes the control values, and finally outputs them to the propellers. To avoid jitter, sensing and actuation follow the logical execution time (LET) paradigm. The application features two periodic task sets whose physical (i.e., abstraction coefficients) and temporal properties vary with their criticality levels: in the *lo*-criticality mode $\sigma = 1$, the IMU measurement task is omitted, also leading to a smaller WCET for the sensor fusion task in comparison to the *hi*-criticality mode $\sigma = 0$. Consequently, the deadline for the controller task can be relaxed, leaving enough resources to guarantee schedulability of the less critical path-planning application. The latter comprises only a single long-running periodic task altering the controller's set point.

At the beginning of the timeline in Figure 3 (A), the UAV is in good condition, indicated by $v_k \leq v_1^*$. Measurements allow the observer abstraction to detect the mild gust of wind and its state rises in turn. While the *lo*-criticality mode $\sigma = 1$ can reject this disturbance, the strong gust of wind occurring later drives the UAV further off its reference trajectory, eventually leading to a change in criticality when v_k passes v_1^*. Here, the scheduler timely reconfigures the system for the *hi*-criticality mode, which can withstand the strongest specified disturbance for any amount of time. However, the increased computational demand for $\sigma = 0$ implies that guaranteed timeliness for the less critical path planning application must be dropped. Figure 3 (B) illustrates how worst-case timing leads to a deadline violation in the latter. Recapitulating, this signifies the notion of physical mixed-criticality introduced by our approach: in the average case, the whole system works as expected. However, during unusually high disturbances, the criticality changes based on the underlying physical process. If the system becomes overloaded, it still has a defined fault state. In the UAV's case, it will hover around the last known reference point until the disturbance wears off and then continue its flight.

3.2 Blind Abstractions

As a first step, we apply the idea presented in [19] to our system model (4) by restricting it to the linear case and extending it to support ellipsoidal bounds and multiple switching modes. For this, we base the analysis on the $||\cdot||_{PQ}$-norms introduced in Section 2.1.

While the rigorous proofs are detached to Appendix A, we still want to exemplify the derivation here. The key mathematical idea is to express the system's safety outputs under the $||\cdot||_S$-norm, split the resulting equation by applying the triangle inequality, and upper bound the uncertain terms by their specified ellipsoids. Introducing an analysis ellipsoid parametrized by the matrix $P \succ 0$ yields a time-dynamic expression $||x_k||_P$, which we can relate to the specification. Applying the ideas, we get

$$||s_{k+1}||_S \overset{(4a),(4e)}{=} ||C_s(A_{\sigma_k}x_k+G_{\sigma_k}d_k)||_S \overset{(3)}{\leq} ||C_s||_{SP}(||A_{\sigma_k}||_{PP}||x_k||_P+||G_{\sigma_k}||_{PD_{\sigma_k}}) \overset{(4f),!}{\leq} 1. \quad (5)$$

Appendix A gives a more formal derivation. There, we prove why the following linear system, which we call *blind (state) abstraction*, provides a sound upper bound (Theorem 3) on the system's state, i.e., $||x_k||_P \leq v_k \ \forall k \in \mathbb{N}_0$ and how obeying the specification translates to a constant-valued *safety bound* v_{max} on the abstraction's state (6c) (Theorem 4). $\forall k \in \mathbb{N}_0$:

$$\begin{cases} v_{k+1} = \rho_{\sigma_k} v_k + \beta_{\sigma_k} & \textit{(Dynamics)} \quad (6a) \\ v_0 = \alpha & \textit{(Initial State)} \quad (6b) \\ v_k \overset{!}{\leq} v_{max}. & \textit{(Safety Bound)} \quad (6c) \end{cases}$$

All coefficients can be derived from the system model (4) by applying different $||\cdot||_{PQ}$-norms as

$$\rho_\sigma := ||A_\sigma||_{PP} \quad \beta_\sigma := ||G_\sigma||_{PD_\sigma} \quad v_{max} := ||C_s||_{SP}^{-1} \quad \alpha := ||I||_{PX_0}. \quad (7)$$

While α is the abstraction's lowest admissible initial state, ρ_σ and β_σ define the worst-case exponential decays and disturbance influences for each mode $\sigma \in \Sigma$, respectively. Aside from being *one-dimensional*, the abstraction's state v_k inherits the desirable property of non-negativity from the norm-based coefficients. In combination, this allows for easy monitoring of the system with respect to two values with clear interpretations: while for $v_k = 0$ the system is in perfect condition, $v_k = v_{max}$ indicates that the abstraction is on the verge of guaranteeing safety.

3.3 Adding Measurements: Observer Abstractions

We argue that the blind abstraction (6) poses a valuable analysis tool. However, as we will show in Section 4, the pessimistic assumptions make it hard to use in practice. This section presents how uncertain measurements can be fed into the abstraction to improve the average-case behavior. Casually speaking, the measurement allows the arising *observer abstractions* to rule out impossible abstraction values. The underlying assumption is that these impossible cases are more often on the pessimistic side than not, improving the abstraction's performance on average.

The key concept behind incorporating measurements is to add a Luenberger observer, i.e., superimposing the state estimate with some linear combination of the measurements. The observer gains $L_\sigma \in \mathbb{R}^{n_x \times n_y(\sigma)}$ define the weights for every state and measurement component. As shown by Lemma 1, the observer abstractions truly generalize the blind

ones when disregarding the measurements, i.e., choosing $L_\sigma = 0\ \forall\sigma \in \Sigma$. In a slight abuse of notation, we redefine the symbols used before and arrive at the following definition of *observer abstractions.* $\forall k \in \mathbb{N}_0$:

$$\begin{cases} v_{k+1} = \rho_{\sigma_k} v_k + \beta_{\sigma_k} + ||L_{\sigma_k} y_k||_P & \textit{(Dynamics)} \quad (8a) \\ v_0 = \alpha & \textit{(Initial State)} \quad (8b) \\ v_k \overset{!}{\leq} v_{max}. & \textit{(Safety Bound)} \quad (8c) \end{cases}$$

Comparing (6) and (8), their structure differs only in the additional measurement term in the dynamics equation. To discuss how adding measurements alters the abstraction, we first introduce the observer dynamics matrix to abbreviate the results later on:

$$\tilde{A}_\sigma := A_\sigma - L_\sigma C_\sigma. \tag{9}$$

With this in mind, the coefficients for (8) are again $||\cdot||_{PQ}$-norms of the matrices defining the system model (4). They read

$$\begin{cases} \rho_\sigma := ||\tilde{A}_\sigma||_{PP} \quad \beta_\sigma := ||G_\sigma||_{PD_\sigma} + ||L_\sigma H_\sigma||_{PZ_\sigma} \quad v_{max} := ||C_s||_{SP}^{-1} \quad \alpha := ||I||_{PX_0} & (10a) \\ \gamma_\sigma := ||L_\sigma C_\sigma||_{PP} \quad \delta_\sigma := ||L_\sigma H_\sigma||_{PZ_\sigma}. & (10b) \end{cases}$$

Besides the coefficients already introduced for the blind abstractions, assessing the worst-case behavior of the observer abstractions requires the two additional terms γ_σ and δ_σ. Lemma 5 shows that the dynamic system

$$\begin{cases} \bar{v}_{k+1} = (\rho_{\sigma_k} + \gamma_{\sigma_k})\bar{v}_k + \beta_{\sigma_k} + \delta_{\sigma_k} \quad \forall k \in \mathbb{N}_0 & (11a) \\ \bar{v}_0 = \alpha & (11b) \end{cases}$$

provides an upper bound on the observer abstraction, i.e., $v_k \leq \bar{v}_k\ \forall k \in \mathbb{N}_0$, effectively reducing the worst-case behavior to a more pessimistic version of the blind abstraction (6). Interpreting this result, the coefficients γ_σ and δ_σ describe the *additional* decay rate and disturbance input separating the average from the worst case.

Assuming that the pair (A_σ, C_σ) is sufficiently observable,[2] the gains L_σ allow for altering the abstraction dynamics for the better (i.e., reduce ρ_σ and β_σ) on average. This reduction comes at the cost of adding uncertainty originating from the measurements via δ_σ. We want to make clear that we are still concerned with how the system states behave, not the observer's error dynamics. It is impossible to alter the system's dynamic properties via the measurements, even if they are perfect, i.e., $\delta_\sigma = 0$. They only allow for improving upon our knowledge about its state. The triangle inequality reflects this mathematically:

$$||A_\sigma||_{PP} \overset{(9)}{=} ||\tilde{A}_\sigma + L_\sigma C_\sigma||_{PP} \overset{(3c),(10)}{\leq} \rho_\sigma + \gamma_\sigma.$$

As a note, the observer gains shown here are independent of any observer encapsulated in (4). While they allow for adding any measurement of the form (4b), we assume that the abstraction is only fed a subset of those measurements used for the controller as querying additional sensors at run time defies the goal of cheap state monitoring.

[2] While full observability allows for choosing ρ_σ arbitrarily small or even as 0 [33, 40], its value can already be reduced with only subsystems being observable.

3.4 Safe Abstraction-Based Run-Time Switching Policies

If we recapitulate on the previous sections, we have found a safe upper bound for assessing the system's state and how well it obeys its safety specification by introducing state abstractions (8). Based upon the assumption of a *safe mode*, i.e., one which can provably withstand the worst-case disturbance over any amount of time, we derive the set of feasible switching modes for the current abstraction state. Finally, we describe a prototypical algorithm for abstraction-based run-time scheduling.

To find the set of feasible modes and thus determine points at which criticality changes, we first use the results from Lemma 5 to define the *maximum safe value* v^*_σ for every mode σ as the highest abstraction value which guarantees that the abstraction stays within its safety bound for the next time step when choosing $\sigma_k = \sigma$:

$$v^*_\sigma := \frac{v_{max} - \beta_\sigma - \delta_\sigma}{\rho_\sigma + \gamma_\sigma} \quad \forall \sigma \in \Sigma. \tag{12}$$

Note that modes with negative v^*_σ are infeasible, i.e., our analysis provides them with no safety guarantees at any time. In contrast, the *feasible set* $\Sigma_f(v_k)$ at time step k defines all modes for which the abstraction stays safe for $k+1$, i.e.,

$$\Sigma_f(v_k) := \{\sigma \in \Sigma | v_k \leq v^*_\sigma\}. \tag{13}$$

By Theorem 7, this set is never empty if a) the system is feasible and b) there is at least one *safe mode* σ permitting worst-case disturbance for any amount of time, which translates to

$$v_0 \leq v_{max} \quad \text{and} \tag{14}$$

$$\exists \sigma \in \Sigma : \quad v^*_\sigma \geq v_{max}. \tag{15}$$

Notice that the condition (15) requires exponential stability, i.e., $\rho_\sigma + \gamma_\sigma < 1$. If these assumptions hold, the system is guaranteed to remain safe by Corollary 8 when the scheduler chooses the switching sequence from the feasible set (13):

$$\sigma_k \in \Sigma_f(v_k) \quad \forall k \in \mathbb{N}_0 \Rightarrow ||s_k||_S \leq 1 \quad \forall k \in \mathbb{N}_0. \tag{16}$$

We now define a *prototypical switching policy.* It greedily chooses the least critical but feasible mode σ for the timestep k by assessing the abstraction's state v_k. Without loss of generality, we assume that the states are ordered by criticality with $\sigma = 0$ being the highest critical one. Formally:

$$\sigma_k = \max \Sigma_f(v_k) \quad \forall k \in \mathbb{N}_0. \tag{17}$$

3.5 Design-Time Analysis

Aside from the analysis ellipsoid parametrized by the matrix $P \succ 0$, each set of observer gains L_σ introduces additional parameters. While the problem of choosing them looks similar to designing a robust observer for switched systems for which the literature provides optimal solutions [16], we are again not concerned with the observer's error dynamics but the system's safety. Therefore, the problem is more challenging: as shown in Section 3.3, adding an observer will – at best – not increase the abstraction coefficients (10). However, as of Section 3.4, we need only one safe mode to guarantee safety at all times. This allows us to adapt the parameters accordingly and incorporate the other (possibly unstable) modes

to best effort. In this section, we derive a *heuristic* for choosing the parameters for large v^*_σ by formulating a parametric semidefinite program (SDP) and solving it on a grid $i \in \mathcal{G}$. To keep the parameters and variables for the individual optimization problems distinct from the derivations above, we index them using i as, e.g., $P(i)$. Afterward, we choose the actual abstraction parameters as the best feasible solution found on the grid. Note that this part of the paper is not a rigorous derivation but a piece of engineering that works for our purposes. Consequently, we focus on conveying the underlying thoughts instead of detailing the fundamentals of and modeling with semidefinite programming. We refer the interested reader to e.g., [41] and [16].

As a starting point, the abstraction coefficients (10) can be computed for every $P \succ 0$. This allows us to apply any heuristic for choosing P and the L_σ and later assess if the resulting abstraction guarantees safety. With that in mind and for the sake of brevity, we do not give proofs for the derivation of the heuristic and formulate assumptions from which we proceed. Informally, the SDP aims to maximize the smallest $v^*_\sigma(i)$, i.e., make the least critical mode schedulable for the longest amount of time. In the following, we will detail the optimization problem mathematically. Most of the constraints are needed to express different $||\cdot||_{PQ}$-norms, but we also add additional ones to enforce safe modes.

The first set of constraints follows from the static portions of the system model (4): we require that the safety outputs obey the specification for all initial states, i.e., $||x_0||_{X_0} \leq 1 \Rightarrow ||C_s x_0||_S \leq 1$. The following linear matrix inequality (LMI) is mathematically equivalent:

$$X_0 \overset{!}{\succeq} C_s^T S C_s. \tag{18}$$

We want to ensure that the analysis ellipsoid of the i-th problem expressed by the matrix variable $P(i) \succ 0$ is non-degenerate and well-conditioned. For convenience, we confine it between the initial state and the specification ellipsoid, leading to $0 \leq v_0 \leq v_{max} \leq 1$ if the problem is feasible. To avoid a loss of definiteness in case of C_s not being full-ranked, we restrict the smallest eigenvalue to the arbitrary value $\sqrt{10^{-6}}$. The following LMIs express all of this:

$$P(i) \overset{!}{\succeq} C_s^T S C_s \quad \text{and} \quad X_0 \overset{!}{\succeq} P(i), \quad P(i) \overset{!}{\succeq} 10^{-6} \cdot I. \tag{19}$$

To make the optimizer aware of the $v^*_\sigma(i)$, we must provide it with the abstraction coefficients (10), which depend on $P(i)$. We assume there is no convex representation for choosing the $\rho_\sigma(i)$, $\gamma_\sigma(i)$, $L_\sigma(i)$, and $P(i)$ simultaneously, use the former two as parameters for the semidefinite program, and confine them to a grid later. To incorporate the observer gains, we define the optimization variables $W_\sigma(i) := P(i)L_\sigma(i)$. By Schur's complement and after multiplying with $P(i)$, we can rewrite $||\tilde{A}_\sigma(i)||_{P(i)P(i)} \leq \rho_\sigma(i)$ and $||L_\sigma(i)C_\sigma||_{P(i)P(i)} \leq \gamma_\sigma(i)$ as $\forall \sigma \in \Sigma$:

$$\begin{bmatrix} \rho_\sigma(i)P(i) & (P(i)A_\sigma - W_\sigma(i)C_\sigma)^T \\ P(i)A_\sigma - W_\sigma(i)C_\sigma & \rho_\sigma(i)P(i) \end{bmatrix} \overset{!}{\succeq} 0 \text{ and } \begin{bmatrix} \gamma_\sigma(i)P(i) & (W_\sigma(i)C_\sigma)^T \\ W_\sigma(i)C_\sigma & \gamma_\sigma(i)P(i) \end{bmatrix} \overset{!}{\succeq} 0. \tag{20}$$

We want to avoid introducing additional parameters and, therefore, an exponential increase in the search space they span. Thus, we define the optimization variables $\beta_k^2(i)$ and $\delta_k^2(i)$ and (probably falsely) assume that $\beta_k^2(i) \approx \beta_k$ and $\delta_k^2(i) \approx \delta_k$. This makeshift is valuable as we can avoid non-convex products such as $\delta_\sigma(i)P^{-1}(i)$ and instead reformulate $||G_\sigma||^2_{P(i)D_\sigma} \leq \beta^2_\sigma(i)$ and $||L_\sigma(i)H_\sigma||^2_{P(i)Z_\sigma} \leq \delta^2_\sigma(i)$ as the LMI contraints

$$\begin{bmatrix} \beta^2_\sigma(i)D_\sigma & (P(i)G_\sigma)^T \\ P(i)G_\sigma & P(i) \end{bmatrix} \overset{!}{\succeq} 0 \quad \text{and} \quad \begin{bmatrix} \delta^2_\sigma(i)Z_\sigma & (W_\sigma(i)H_\sigma)^T \\ W_\sigma(i)H_\sigma & P(i) \end{bmatrix} \overset{!}{\succeq} 0 \quad \forall \sigma \in \Sigma. \tag{21}$$

To express the maximum safe values, we rewrite (12) as $(\rho_\sigma + \gamma_\sigma)v^*_\sigma + \beta_\sigma + \delta_\sigma \leq v_{max}$ and introduce $v^*_\sigma(i)$ as optimization variables. We have found empirically that this inequality can be approximated as $c_\sigma(i) := (\rho_\sigma(i) + \gamma_\sigma(i))v^*_\sigma(i) + \beta^2_\sigma(i) + 4\delta^2_\sigma(i) \leq v_{max}$. Given the norm-based definition (10) of v_{max}, which is also used to obtain its actual value after optimization, we get

$$\begin{bmatrix} P(i) & c_\sigma(i)C_s^T \\ c_\sigma(i)C_s & S^{-1} \end{bmatrix} \overset{!}{\geq} 0 \quad \forall \sigma \in \Sigma. \tag{22}$$

We require a subset of modes $\emptyset \neq \Sigma_s \subseteq \Sigma$ to be safe which translates to

$$v^*_\sigma \overset{!}{\geq} v_{max} \quad \forall \sigma \in \Sigma_s. \tag{23}$$

As a last set of constraints, we introduce the single variable $v^*_{min}(i)$ as the minimum of all $v^*_\sigma(i)$,

$$v^*_{min}(i) \overset{!}{\leq} v^*_\sigma(i) \quad \forall \sigma \in \Sigma, \tag{24}$$

and use it as the maximization objective. Requirements on the modes' exponential decays can then be imposed using the optimization parameters $\rho_\sigma(i)$ and $\gamma_\sigma(i)$. The next step is to define a grid for their values on which to solve the SDP. The parameter space grows exponentially in the number of modes. We consider this to be undesired and consequentially constrain all values $\rho_\sigma(i)$ and $\gamma_\sigma(i)$ onto a one-dimensional grid $i \in \mathcal{G} := \{0, \ldots, n_g - 1\}$ as

$$\rho_\sigma(i) = \underline{r}_\sigma + \frac{\bar{r}_\sigma - \underline{r}_\sigma}{n_g - 1} i \quad \text{and} \quad \gamma_\sigma(i) = l_\sigma - \rho_\sigma(i) \quad \forall \sigma \in \Sigma. \tag{25}$$

With the spectral radius $\mathcal{R}(A_\sigma)$, i.e., A_σ's largest absolute eigenvalue, the hyperparameters for our heuristic are given by Σ_s, $\mathcal{R}(A_\sigma) \leq l_\sigma, 0 \leq \underline{r}_\sigma \leq \bar{r}_\sigma \leq l_\sigma$. Again without proof, we conceptualize their meaning: l_σ sets the maximum admissible decay rate. Bounding it from below makes sense, as a sound abstraction must not decay faster than the underlying system. Imposing $\underline{r}_\sigma \leq \rho_\sigma(i)$ avoids high observer gains which amplify noise, i.e., they increase $\gamma_\sigma(i)$ and $\delta_\sigma(i)$. Lastly, $\rho_\sigma(i) \leq \bar{r}_\sigma$ sets the desired decay rate in the optimistic case.

The parameterization works as follows: for every grid point $i \in \mathcal{G}$, solve the SDP $\max v^*_{min}(i)$ in $P(i), W_\sigma(i), \beta^2_\sigma(i), \delta^2_\sigma(i)$, $v^*_\sigma(i)$ and $v^*_{min}(i)$ subject to (18)–(24) with the parameters set by (25), then compute $L_\sigma(i) = P^{-1}(i)W_\sigma(i)$. For all feasible SDPs, determine the actual abstraction coefficients (10) as the optimization variables do not reflect their actual values. Of all safe abstraction parametrizations, choose the one which maximizes the actual $v^*_{min} := \min_{\sigma\in\Sigma} v^*_\sigma$ obtained from (10).

4 Evaluation

In this section, we evaluate how introducing measurements lessens the blind abstractions' inherent pessimism and that the run-time policy (17) adapts well in a weakly-hard setting. Further, we demonstrate that mere stability is insufficient for guaranteeing safety and that correlated disturbances justify using a set-valued disturbance model. While this was carried out in a custom simulator, we complement the results with execution time measurements, which compare the abstractions' run-time overheads to those of set-valued estimators from the literature on an ARM Cortex-M4F processor.

4.1 Benchmark System

We use a double integrator as our benchmark system. The two modes are defined as *control* ($\sigma = 0$) and *skip* ($\sigma = 1$), i.e., measure both states and apply a state-space controller as the control signal or skip measurements and zero the controller output. They represent extreme examples for high and low criticality. We do not consider that measurement uncertainties enter the system via the controller and instead specify a single process disturbance affecting the states, which is the same mathematically. Further, we subject the measurements passed to the state abstraction to uncertainties. When executed, the controller places both closed-loop eigenvalues at 0.9. The safety specification is set arbitrarily to constrain both states and the control signal to a sphere of radius 1.25. The initial state is chosen as one tenth of this radius. For the lack of an analytical bound, we empirically chose the disturbances as high as possible such that our heuristic is still able to verify that $\sigma = 0$ is a safe mode, i.e., $v_{max} \le v_0^*$. The benchmark's system model reads

$$\begin{cases} A_0 = \begin{bmatrix} 0.9950 & 8.205 \cdot 10^{-2} \\ -0.1100 & 0.8050 \end{bmatrix}, A_1 = \begin{bmatrix} 1 & 9.091 \cdot 10^{-2} \\ 0 & 1 \end{bmatrix} \\ G_0 = G_1 = \begin{bmatrix} -5.051 \cdot 10^{-3} \\ -0.1111 \end{bmatrix}, D_0 = D_1 = 3.516 \cdot 10^{-4} \\ C_0 = H_0 = I, Z_0 = 2.384 \cdot 10^{-5} \cdot I, \quad C_1, H_1, Z_1 \text{ empty} \\ X_0 = 1.563 \cdot 10^{-2} \cdot I, C_s = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0.99 & 1.755 \end{bmatrix}, S = 1.563 \cdot I. \end{cases} \tag{26}$$

We draw the initial states independently and identically (i.i.d.) uniformly distributed from the surface of $\mathcal{E}(X_0)$.

4.2 Disturbance Scenarios

According to their elliptical bounds, we specify two types of disturbance scenarios for the simulation runs: in the *benign* case, we draw both disturbances uniformly and i.i.d. from their ellipsoids' volumes resulting in uncorrelated noise. In contrast, the *malicious* scenario aims to approximate the worst-case disturbance. We achieve this by drawing 100 points uniformly and i.i.d. from the corresponding ellipsoids' surfaces in each timestep. For the process disturbance, we then select the point d_k which maximizes $||C_s(x_k + G_{\sigma_k} d_k)||_S$ given the current state x_k. As the measurement uncertainty only affects the abstraction, we select z_k to maximize $||x_k + L_{\sigma_k} H_{\sigma_k} z_k||_P$. This procedure yields time-correlated values.

4.3 Effect of Measurements

In a first simulative experiment, we want to assess the effect of adding measurements as described in Section 3.3 by parametrizing three different state abstractions: the first one performs a *full measurement*, i.e., it utilizes both measurements via C_0 and H_0. The second abstraction uses *partial measurements* only, i.e., it disregards the measurement y_2 by omitting the second row of C_0 and H_0, respectively. Finally, the third abstraction is blind and thus disregards both y_1 and y_2, allowing it only to use the worst-case assumptions. Note that no measurements are used when skipping the controller.

Parametrizing each abstraction using our heuristic on an Intel Core i7-8565U took less than five seconds. Both observer abstractions were parametrized for $\underline{r}_0 = 0.3$, $\bar{r}_0 = 0.9$, $\lambda_0 = 0.94$, $\underline{r}_1 = 1.05$ and $\bar{r}_1 = \lambda_1 = 1.15$ on $n_g = 101$ grid points. For the blind abstraction, we

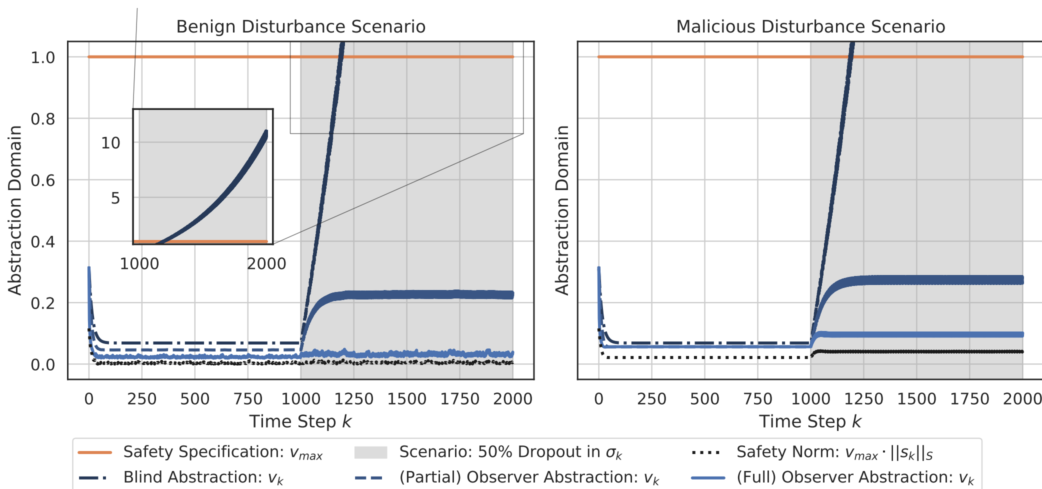


Figure 4 Effect of measurements. A double integrator is subjected to the benign (left) and malicious (right) disturbance scenario. Its state (dotted black) is compared to three state abstractions (blue), either running blindly (dash-dotted), measuring its partial (dashed), or full state (solid). The controller is always executed until time step 1000 and then subjected to 50% dropout, i.e., the modes are alternated deterministically.

Table 1 State abstraction coefficients for the double integrator and different measurement outputs computed by (10) and rounded to three significant digits.

Measurement	α	v_{max}	σ	ρ_σ	γ_σ	β_σ	δ_σ	v^*_σ
Full	0.314	1.00	0	0.396	0.554	0.0104	0.00616	1.05
			1	1.07	0.00	0.00425	0.00	0.934
Partial	0.317	1.00	0	0.900	0.0400	0.00448	0.000235	1.06
			1	1.07	0.00	0.00425	0.00	0.933
Blind	0.310	1.00	0	0.940	0.00	0.00411	0.00	1.06
			1	1.07	0.00	0.00411	0.00	0.933

matched $\bar{r}_0 = \lambda_0 = 0.94$ to make the SDP feasible. Table 1 shows the resulting abstraction coefficients computed via (10). The observer gains read $L_0 \approx \begin{bmatrix} 0.577 & 0.0487 \\ -0.0687 & 0.458 \end{bmatrix}$ for the full and $L_0 \approx \begin{bmatrix} 0.0393 \\ -0.0233 \end{bmatrix}$ for the partial measurement. The condition numbers of the shape matrices P^{-1} range between ~6.64 and ~7.48 with ~0.155 being the smallest of all their eigenvalues, i.e., the analysis ellipsoids are well-defined.

Figure 4 depicts two simulations in which the controller always runs at first and is then subjected to a deterministic 50% dropout sequence, i.e., the switching signal alternates between both modes starting at time step 1000. The benign or malicious disturbances perturb the system during the whole simulation, respectively. After the initial values have decayed, the blind abstraction's state is strictly higher than those of both observer abstractions. In the malicious scenario, the latter are nearly indistinguishable until the dropouts occur. There, the full measurement improves its abstraction noticeably. As we show later, the additional run-time overhead is negligible if the measurement is already available. Quantifying the overapproximation over 100 simulation runs by comparing the mean quotient of the individual v_k and $v_{max} \cdot ||s_k||_S$ yields factors between ~2.37 (full measurement, malicious disturbance,

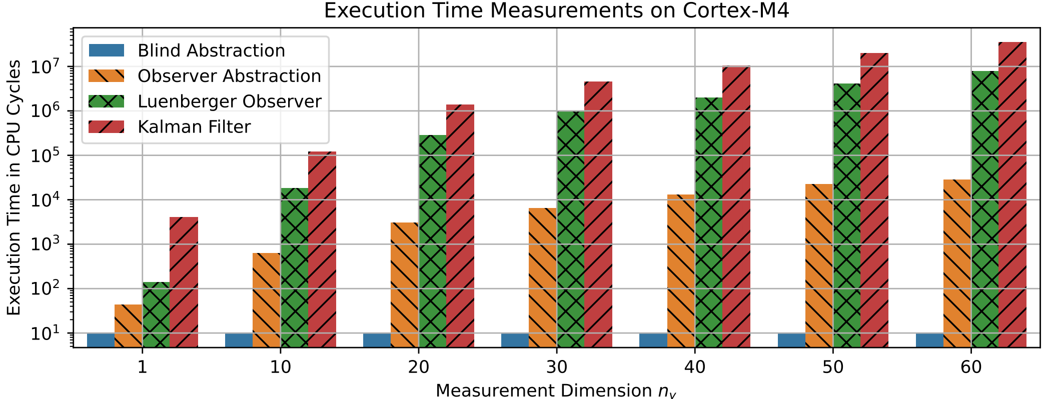


Figure 5 Execution times for updating a blind and observer abstraction by (6a) and (8a) as well as the update steps for a set-valued Luenberger observer [16] and Kalman filter [30] for different measurement sizes n_y. Times are given in CPU cycles on an ARM Cortex-M4F processor and plotted in logarithmic scales.

during dropout) and ~71.6 (partial measurement, benign disturbance, during dropout). This comparison leaves out the blind abstraction during dropout as – opposed to the observer abstractions – it appears to diverge. The large discrepancy signifies the benefit of separating the average case from the worst case at run time by reusing the controller's measurements. While our analysis cannot give guarantees about the steady states in case of dropouts, the simulations indicate that the observer abstractions average out well below their safety bound. The run-time algorithm (8) can predict upcoming violations in any case, such as for the blind abstraction around timestep 1200, allowing the scheduler to timely reconfigure the system for higher criticality.

4.4 Run-Time Overhead

While Section 4.3 focuses on the benefits of adding measurements, this section addresses their run-time overhead. For this, we measured the execution times of updating a blind and an observer abstraction by (6a) and (8a) given different measurement sizes $n_y \in \{1, \ldots, 60\}$. To contrast the results, we also measured the cost of updating a set-valued Luenberger observer [16] and Kalman filter [30]. Most of the time, only a subset of the states is measured, i.e., $n_y < n_x$. We set $n_x = n_y$ for both observers, effectively reducing their execution time to a minimum. This does not affect the observer abstractions as the measurement term is independent of n_x, i.e., $||L_{\sigma_k} y_k||_P = \sqrt{y_k^T L_{\sigma_k}^T P L_{\sigma_k} y_k}$. We further reduced the cost for the Kalman filter by omitting the proposed line search for ω in [30, (13c)] in favor of a constant value while choosing ω by the trace criterion given in [16, (29)] for [30, (11c)].

Note that an additional online reachability analysis is required in all cases. While for state abstractions this reduces to trivial real-valued comparisions as outlined in Section 3.4, both set-valued observers from the literature require elaborate techniques as their ellipsoids' shapes are time-varying. As the references do not provide the respective analyses, we neither investigated this cost here nor their pessimism in Section 4.3. In all cases, we do not consider the acquisition of measurements as we assume that a state abstraction reuses the ones passed to the controller.

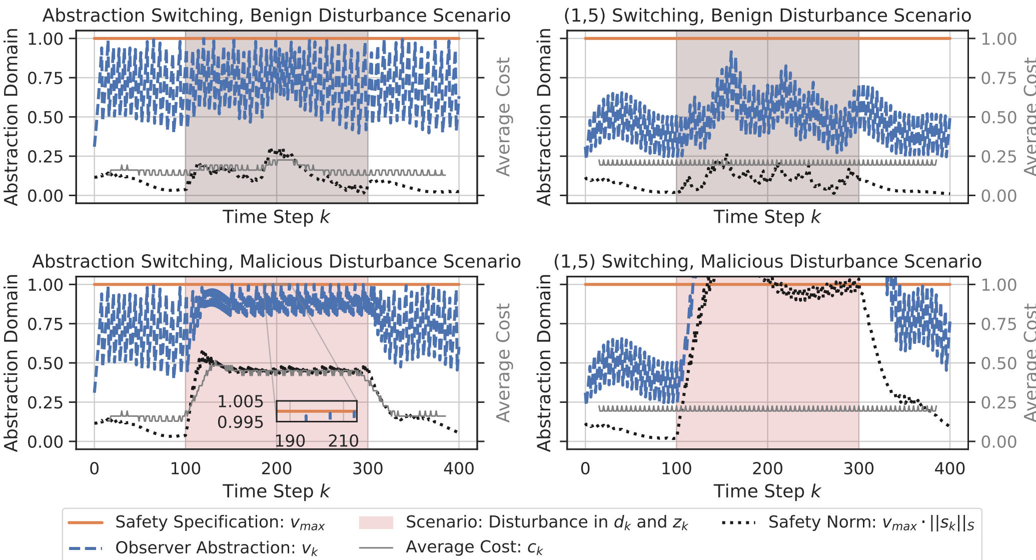


Figure 6 Comparison between the abstraction switching policy (17) and a minimal (1, 5) sequence (columns) subject to both disturbance scenarios (rows) when applied to the double integrator (26).

We conducted the experiment on an STM32F411E-Discovery board [37] with an ARM Cortex-M4F processor using `gcc` with `-O3 -ffp-contract=off`. Utilizing the DWT cycle counter, the measurement overhead amounted to one cycle consistently. We drew all coefficient values at random and computed the corresponding update steps in single precision for ten iterations utilizing some optimizations (e.g., take into account positive definiteness). We repeated this procedure for ten sets of coefficients, yielding 100 measurements per update step and n_y. Note that the parameters are stored in RAM. Placing them in flash adds additional wait states [37, p. 44]. This, however, affects all candidates. Figure 5 depicts the measurement results.

While the blind abstraction used 10 cycles consistently, the execution times for the other update steps rise for increasing measurement sizes. Note that this rise is not always monotonic as of compiler optimizations. The cycle counts for each size n_y coincide for all 100 execution time measurements of every individual update step (except for the Kalman filter for which the variations were negligible), which is to be expected as all are constant-time algorithms. The observer abstraction's maximum of 29874 cycles was attained at $n_y = 58$. In comparison, the Luenberger observer and Kalman filter took ∼198 and ∼1057 times longer for an update of this size. Considering the 96 MHz clock, updating the observer abstraction took around $300\mu s$ for a system with $n_y = 58$ measurement signals. Looking at the data, we conclude that the additional overhead from adding measurements is still negligible compared to the complexity imposed by a control system of this size.

4.5 Abstraction-Based Run-Time Switching and Insufficiency of Stability

Sections 4.3 and 4.4 evaluate the abstraction itself, i.e., how well it can track the physical state and how expensive the assessment is at run time. This is possible for any known switching sequence. In contrast, here we use the state abstraction as an inexpensive way to construct the switching sequence such that the system stays safe at all times by applying our prototypical policy (17). Further, we exemplify that stability alone is an insufficient criterion for safety when dealing with disturbances. These experiments were again carried out in simulation.

We only consider an abstraction utilizing both measurements. Without the more pessimistic configurations detailed in Section 4.3, we were able to increase the process uncertainty by a factor of 10 (i.e., $100 \cdot D_\sigma$) while keeping the heuristic working. To achieve this, we decreased $\bar{r}_0$ to 0.7, which was previously disregarded to keep the full and partial measurements comparable.

As the system is feasible and $\sigma = 0$ is safe, the policy guarantees safety at all time steps by Corollary 8, which was validated over 100 simulation runs. For the experiment, we compare it to a minimal $(1, 5)$-switching sequence, i.e., execute the controller only every fifth time. We decided on this specific sequence as its execution ratio is close to but higher than the average generated by the abstraction switching in the benign case. As shown by Theorem 9, the system is exponentially stable even under the more general $(1, 5)$ weakly-hard execution, i.e., skip no more than four consecutive controller executions. Even though our approach cannot guarantee safety under this sequence, the abstraction again remains a sound upper bound on the specification output as of Theorem 3. Subjecting each switching policy to both disturbance scenarios yields four combinations. Figure 6 depicts one of the 100 simulation runs obtained this way. As a visual guide for the switching sequence, the plots are overlaid with the average cost from controller execution defined by $c_k := 1 - \text{avg}(\sigma_k)$ where $\text{avg}(\cdot)$ is the moving average over 31 time steps.

While the abstraction stays below v_{max} for $(1, 5)$-switching under the benign disturbance, even the system itself violates its specification in the malicious case for all 100 runs indicating a critical condition. This shows that mere stability is insufficient in the presence of disturbances and that correlated disturbances require more pessimistic approaches, such as ellipsoidal models.

By design, the switching policy (17) keeps the abstraction and, by extension, the system within safe bounds at all time steps. While the average cost stays nearly constant at 15% to 20% of the controller executions during the benign disturbance, it adapts to the malicious case by increasing the average cost to around 45% after a brief period of overshoot between timesteps 100 and 150. From this, we conclude that our state abstractions are a pessimistic yet cheap and simple enough tool for monitoring and influencing run-time adaptive switched linear systems at the operating system level.

5 Related Work

Adaptive scheduling and selective verification of CPSs is a thriving field of research. Our approach shares its objective correspondingly with a wide area of related work that we inherently expand upon as well. Yet, we are not aware of any approach that provides (1) sound yet very cost-effective state abstraction, (2) guarantees not only stability but also adhering to safety constraints, and (3) uses observer-based feedback of measured values to mitigate pessimism. The following papers each differ from our approach in one or more respects.

We are not the first to note that CPSs are a combination of criticalities in time and quality of control (QoC), thus sharing parallels with MCS. For example, in [34], the QoC is maximized while preserving guarantees for cyber tasks, while [26] maps high-level hazards to the criticality of tasks.

The co-design of the controller and real-time system was introduced by Seto et al. [36] to improve utilization by relaxing timing requirements. Realizing the impact on QoC [4], [31] proposed to adjust control parameters to counteract deadline misses. For example, by adapting the control period and deadline adherence [10, 11, 17, 20]. A popular variant of

mitigated timing constraints is (m, K)-firm scheduling introduced by [21]. Subsequently, more flexible task models and switching between safe and optimistic control were researched [15, 47, 43, 5, 35]. Likewise, for sporadic bursts of deadline misses, control parameters can be adapted [32, 14] and [44, 29] provide stability analysis of closed-loop systems.

Conversely, optimal sampling period approaches [8, 12] infer a sampling period to maximize QoC and minimize the quadratic cost function regardless of other tasks.

Even more radical are approaches on self-triggered control [3, 39, 22, 23], which analyze the system state to predict the next control instant. These approaches offer superior average-case performance – unfortunately – to the detriment of overall schedulability.

The field of model predictive control emerged from the requirement to impose bounds on control systems. Naturally, our approach reuses its fundamental concepts, e.g. bounding the reachable sets of a perturbed system by ellipsoids as in Tube Model Predictive Control [9]. In fact, it can be interpreted as a stripped down version of MPC. While utilizing information about the full system state promises to be less pessimistic and there are even variants available which incorporate scheduling decisions [24, 25, 28], they require solving potentially large optimization problems at run time. Further, determining a discrete switching sequence can yield a problem growing exponentially in the prediction horizon [1]. This poses a high burden on their implementation in embedded control systems [27]. We argue that while superior in performance, MPC is only viable if the control system requires it anyway.

6 Conclusion

The physical state, or rather its distance from the safety specification, is a crucial criterion for the actual criticality of control applications in many real-world applications. In this paper, we proved that careful abstraction allows for state estimation efficiently enough to serve as run-time monitoring. Therefore, we extended our previously defined one-dimensional convergence rate abstractions described in [19] to linear systems which feature multiple modes of controller execution and are subject to ellipsoidally bound disturbances. Further, we introduced the concept of observer abstractions that allow for the feedback of uncertain measurements to lessen overapproximation, making the abstractions much more practical. Given a well-posed specification, we then developed a design-time analysis for their parameterization. Finally, as part of a case study, we validated our disturbance model, showed that the benefits of introducing measurements outweighs their run-time overhead, and illustrated that our prototypical run-time policy for optimistically choosing the mode of controller execution adapts well to changing disturbances while obeying safety specifications even in the worst case. We consider our work a foundation for designing mixed-criticality systems and scheduling approaches that leverage a system's physical state for control tasks as an additional criticality criterion.

References

1 Ricardo P. Aguilera and Daniel E. Quevedo. On the stability of MPC with a Finite Input Alphabet. *IFAC Proceedings Volumes*, 44(1):7975–7980, 2011. doi:10.3182/20110828-6-IT-1002.02705.

2 Anayo K. Akametalu, Claire J. Tomlin, and Mo Chen. Reachability-Based Forced Landing System. *Journal of Guidance, Control, and Dynamics*, 41(12):2529–2542, 2018. doi:10/gfpcbn.

3 Shigeru Akashi, Hideaki Ishii, and Ahmet Cetinkaya. Self-triggered control with tradeoffs in communication and computation. *Automatica*, 94:373–380, 2018. doi:10.1016/j.automatica.2018.04.028.

4 K.-E. Årzén, A. Cervin, J. Eker, and L. Sha. An introduction to control and scheduling co-design. In *Proceedings of the 39th IEEE Conference on Decision and Control*, volume 5, pages 4865–4870, 2000. `doi:10/ck4mjj`.

5 Stanley Bak, Deepti K. Chivukula, Olugbemiga Adekunle, Mu Sun, Marco Caccamo, and Lui Sha. The system-level simplex architecture for improved real-time embedded system safety. In *Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 99–107, 2009. `doi:10.1109/RTAS.2009.20`.

6 André Benine-Neto, Stefano Scalzi, Saïd Mammar, and Mariana Netto. Dynamic controller for lane keeping and obstacle avoidance assistance system. In *13th International IEEE Conference on Intelligent Transportation Systems*, pages 1363–1368, 2010. `doi:10/bj4xpb`.

7 Guillem Bernat, Alan Burns, and Alberto Liamosi. Weakly Hard Real-time Systems. *IEEE Transactions on Computers*, 50(4):308–321, 2001. `doi:10/cqd6d3`.

8 Enrico Bini and Giuseppe M. Buttazzo. The optimal sampling pattern for linear control systems. *IEEE Transactions on Automatic Control*, 59(1):78–90, 2014. `doi:10/f3nxws`.

9 Mark Cannon, Johannes Buerger, Basil Kouvaritakis, and Saša Rakovic. Robust Tubes in Nonlinear Model Predictive Control. *IEEE Transactions on Automatic Control*, 56(8):1942–1947, 2011. `doi:10/dhg386`.

10 Rosa Castañé, Pau Marti, Manel Velasco, Anton Cervin, and Daniel Henriksson. Resource management for control tasks based on the transient dynamics of closed-loop systems. In *Proceedings of the 18th Euromicro Conf. on Real-Time Systems (ECRTS '06)*, pages 172–182, Los Alamitos, CA, USA, 2006. `doi:10/bx9rps`.

11 Anton Cervin, Johan Eker, Bo Bernhardsson, and Karl-Erik Årzén. Feedback–feedforward scheduling of control tasks. *Real-Time Systems*, 23(1-2):25–53, 2002. `doi:10/fnb5nk`.

12 Anton Cervin, Manel Velasco, Pau Martí, and Antonio Camacho. Optimal online sampling period assignment: Theory and experiments. *IEEE Trans. on Control Systems Technology*, 19(4):902–910, 2011. `doi:10/d38qtf`.

13 Long Cheng, Kai Huang, Gang Chen, Biao Hu, and Alois Knoll. Mixed-criticality control system with performance and robustness guarantees. In *Proceedings of the IEEE Trustcom/BigDataSE/ICESS*, pages 767–775, 2017. `doi:10/gr65zt`.

14 Hoon Sung Chwa, Kang G. Shin, and Jinkyu Lee. Closing the gap between stability and schedulability: A new task model for cyber-physical systems. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 327–337, 2018. `doi:10/gqz7sd`.

15 Xiaotian Dai, Wanli Chang, Shuai Zhao, and Alan Burns. A Dual-Mode Strategy for Performance-Maximisation and Resource-Efficient CPS Design. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):85:1–85:20, October 2019. `doi:10/gq5wfw`.

16 Monia Dkhil, Thach Ngoc Dinh, Zhenhua Wang, Tarek Raïssi, and Messaoud Amairi. Interval Estimation for Discrete-Time Switched Linear Systems Based on L_∞ Observer and Ellipsoid Analysis. *IEEE Control Systems Letters*, 5(1):13–18, 2021. `doi:10/gq5wdm`.

17 Daniele Fontanelli, Luca Greco, and Luigi Palopoli. Soft real-time scheduling for embedded control systems. *Automatica*, 49(8):2330–2338, 2013. `doi:10/gq5wfr`.

18 Maximilian Gaukler, Andreas Michalka, Peter Ulbrich, and Tobias Klaus. A New Perspective on Quality Evaluation for Control Systems with Stochastic Timing. In *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (HSCC '18)*, pages 91–100, New York, NY, USA, April 2018. ACM. `doi:10/gq5wdx`.

19 Maximilian Gaukler, Tim Rheinfels, Peter Ulbrich, and Günter Roppenecker. Convergence Rate Abstractions for Weakly-Hard Real-Time Control, 2019. `doi:10/gq5wdt`.

20 Luca Greco, Daniele Fontanelli, and Antonio Bicchi. Design and Stability Analysis for Anytime Control via Stochastic Scheduling. *IEEE Transactions on Automatic Control*, 56(3):571–585, March 2011. `doi:10/bgwcfj`.

21 Moncef Hamdaoui and Parameswaran Ramanathan. A Dynamic Priority Assignment Technique for Streams with (m, K)-firm Deadlines. *IEEE Transactions on Computers*, 44(12):1443–1451, 1995. doi:10/cq995j.

22 W. P. M. H. Heemels and M. C. F. Donkers. Model-based periodic event-triggered control for linear systems. *Automatica*, 49(3):698–711, 2013. doi:10/f4sp65.

23 W. P. M. H. Heemels, Karl Henrik Johansson, and Paulo Tabuada. An introduction to event-triggered and self-triggered control. In *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, pages 3270–3285, 2012. doi:10/f22pf7.

24 Dan Henriksson, Anton Cervin, Johan Åkesson, and Karl-Erik Årzén. Feedback scheduling of model predictive controllers. In *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 207–216, 2002. doi:10/frxj6d.

25 Erik Henriksson, Daniel E. Quevedo, Henrik Sandberg, and Karl Henrik Johansson. Self-Triggered Model Predictive Control for Network Scheduling and Control. *IFAC Proceedings Volumes*, 45(15):432–438, 2012. doi:10/f24cvt.

26 Viacheslav Izosimov and Erik Levholt. Mixed criticality metric for safety-critical cyber-physical systems on multi-core architectures. *Methods*, 2:8, 2015.

27 Tor Arne Johansen. Toward dependable embedded model predictive control. *IEEE Systems Journal*, 11(2):1208–1219, 2017. doi:10.1109/JSYST.2014.2368129.

28 Yingzhao Lian, Yuning Jiang, Naomi Stricker, Lothar Thiele, and Colin N. Jones. Robust resource-aware self-triggered model predictive control. *IEEE Control Systems Letters*, 6:1724–1729, 2022. doi:10/gr65zv.

29 Martina Maggio, Arne Hamann, Eckart Mayer-John, and Dirk Ziegenbein. Control-System Stability Under Consecutive Deadline Misses Constraints. In Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:24, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10/ghqcvc.

30 Benjamin Noack, Marcus Baum, and Uwe D. Hanebeck. State estimation for ellipsoidally constrained dynamic systems with set-membership pseudo measurements. In *Proceedings of the IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*, pages 297–302, 2015. doi:10/gq5wfg.

31 Paolo Pazzaglia, Arne Hamann, Dirk Ziegenbein, and Martina Maggio. Adaptive design of real-time control systems subject to sporadic overruns. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1887–1892, 2021. doi:10/gq5whh.

32 Paolo Pazzaglia, Claudio Mandrioli, Martina Maggio, and Anton Cervin. DMAC: Deadline-Miss-Aware Control. In Sophie Quinton, editor, *Proceedings of the 31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:24, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10/gf6j4p.

33 Gilberto Pin, Peng Li, Giuseppe Fedele, and Thomas Parisini. A deadbeat observer for LTI systems by time/output-dependent state mapping. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 4795–4800, 2017. doi:10.1109/CDC.2017.8264367.

34 Reinhard Schneider, Dip Goswami, Alejandro Masrur, Martin Becker, and Samarjit Chakraborty. Multi-layered Scheduling of Mixed-criticality Cyber-physical Systems. *Journal of System Architecture*, 59(10):1215–1230, November 2013. doi:10/f5qdjc.

35 Danbing Seto, Bruce H. Krogh, Lui Sha, and Alongkarn Chutinan. Dynamic control system upgrade using the simplex architecture. *IEEE Control Systems*, 18(4):72–80, August 1998. doi:10/fk87g6.

36 Danbing Seto, John P. Lehoczky, Lui Sha, and Kang G. Shin. On Task Schedulability in Real-time Control Systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, pages 13–21, Los Alamitos, CA, USA, December 1996. doi:10.1109/REAL.1996.563693.

37 STMicroelectronics. *Reference manual RM0383, Rev 3*, 2018.

38 Mohammad M. Sultan, Daniel Biediger, Bernard Li, and Aaron T. Becker. The Reachable Set of a Drone: Exploring the Position Isochrones for a Quadcopter. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7679–7685, 2021. doi:10.1109/ICRA48506.2021.9561715.

39 Sebastian Trimpe and Raffaello D'Andrea. Event-based state estimation with variance-based triggering. *IEEE Transactions on Automatic Control*, 59(12):3266–3281, 2014. doi:10.1109/TAC.2014.2351951.

40 Sezai Emre Tuna. Deadbeat Observer: Construction via Sets. *IEEE Transactions on Automatic Control*, 57(9):2333–2337, 2012. doi:10.1109/TAC.2012.2183197.

41 Lieven Vandenberghe and Stephen Boyd. Semidefinite Programming. *SIAM Review*, 38(1):49–95, 1996. doi:10.1137/1038003.

42 Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS '07)*, pages 239–243, 2007. doi:10/cg89kh.

43 Prasanth Vivekanandan, Gonzalo Andres Garcia, Heechul Yun, and Shawn Shahriar Keshmiri. A simplex architecture for intelligent and safe unmanned aerial vehicles. In *Proceedings of the IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 69–75, 2016. doi:10.1109/RTCSA.2016.17.

44 Nils Vreman, Anton Cervin, and Martina Maggio. Stability and Performance Analysis of Control Systems Subject to Bursts of Deadline Misses. In Björn B. Brandenburg, editor, *Proceedings of the 33rd Euromicro Conference on Real-Time Systems*, volume 196 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:23, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECRTS.2021.15.

45 Nils Vreman, Paolo Pazzaglia, Victor Magron, Jie Wang, and Martina Maggio. Stability of Linear Systems Under Extended Weakly-Hard Constraints. *IEEE Control Systems Letters*, 6:2900–2905, 2022. doi:10/gqnxtm.

46 Guorong Wang, Yimin Wei, and Sanzheng Qiao. *Generalized Inverses: Theory and Computations.* Springer Singapore, first edition, 2018. doi:10/gq5wd4.

47 Xiaofeng Wang, Naira Hovakimyan, and Lui Sha. Rsimplex: A robust control architecture for cyber and physical failures. *ACM Transactions on Cyber-Physical Systems*, 2(4), July 2018. doi:10.1145/3121428.

A Proofs

While in Section 3 we focused on conveying the ideas and design decisions behind our approach, here we formalize and prove the state abstractions' properties.

▶ **Lemma 1** (Generalization). *By setting* $L_\sigma = 0 \quad \forall \sigma \in \Sigma$*, the observer abstraction* (8) *is a true generalization of the blind abstraction* (6)*.*

Proof. When applying norm homogeneity (3a), the observer abstraction's dynamics (8a) and their coefficients (10) are reduced to the blind counterparts (6a) and (7) by setting $L_\sigma = 0 \quad \forall \sigma \in \Sigma$. ◀

▶ **Lemma 2** (Disturbance Bound). *The highest impact of an ellipsoidally constrained disturbance in terms of the* P*-norm is attained at the ellipsoid's surface:*

$$\max_{||x||_Q \leq 1} ||Mx||_P = \max_{||x||_Q = 1} ||Mx||_P \overset{(1c)}{=} ||M||_{PQ}. \tag{27}$$

Proof. We prove destructively: assume some $\tilde{x}$ with $||\tilde{x}||_Q < 1$ were a maximizer on the ellipsoid's inside. Then $x := ||\tilde{x}||_Q^{-1}\tilde{x}$ is of unit length (i.e., it lies on the surface of Q) and by the norm properties

$$||Mx||_P \overset{(3a)}{=} \underbrace{||\tilde{x}||_Q^{-1}}_{>1} ||M\tilde{x}||_P \overset{(3b)}{>} ||M\tilde{x}||_P$$

is greater than the assumed maximizer. This contradiction concludes (27). ◀

▶ **Theorem 3** (Soundness). *For any switching sequence $\sigma_k \in \Sigma, k \in \mathbb{N}_0$, an (observer) abstraction* (8) *poses a sound upper bound for the underlying system's state* (4)*, i.e.,*

$$||x_k||_P \leq v_k \quad \forall k \in \mathbb{N}_0. \tag{28}$$

Proof. We prove (28) by induction.

Base Case. By the ellipsoidal constraint on the initial state, the norm properties, and the definition of blind abstractions

$$||x_0||_P \overset{(4d)}{\leq} \max_{||x||_{X_0} \leq 1} ||x||_P \overset{(27)}{=} \underbrace{||I||_{PX_0}}_{\overset{(10)}{=} \alpha} \overset{(8b)}{=} v_0.$$

Inductive Assumption (I.A.). $||x_k||_P \leq v_k$.

Inductive Step ($k \to k+1$). We incorporate the observer by adding a zero and substituting the measurement. Applying the triangle inequality and bounding the summands by the norm properties yields the abstraction's coefficients and dynamics by definition:

$$||x_{k+1}||_P \overset{(4a)}{=} ||A_{\sigma_k} x_k + G_{\sigma_k} d_k||_P \overset{(9),+0}{=} ||\tilde{A}_{\sigma_k} x_k + G_{\sigma_k} d_k + L_{\sigma_k} C_{\sigma_k} x_k||_P \overset{(4b)}{=}$$

$$||\tilde{A}_{\sigma_k} x_k + G_{\sigma_k} d_k - L_{\sigma_k} H_{\sigma_k} z_k + L_{\sigma_k} y_k||_P \overset{(3c)}{\leq} ||\tilde{A}_{\sigma_k} x_k||_P + ||G_{\sigma_k} d_k||_P + ||L_{\sigma_k} H_{\sigma_k} z_k||_P$$

$$+ ||L_{\sigma_k} y_k||_P \overset{(1c),(4c),(27)}{\leq} \underbrace{||\tilde{A}_{\sigma_k}||_{PP}}_{\overset{(10)}{=} \rho_{\sigma_k}} ||x_k||_P + \underbrace{||G_{\sigma_k}||_{PD_{\sigma_k}} + ||L_{\sigma_k} H_{\sigma_k}||_{PZ_{\sigma_k}}}_{\overset{(10)}{=} \beta_{\sigma_k}} + ||L_{\sigma_k} y_k||_P \overset{I.A.}{\leq}$$

$$\rho_{\sigma_k} v_k + \beta_{\sigma_k} + ||L_{\sigma_k} y_k||_P \overset{(8a)}{=} v_{k+1}.$$ ◀

▶ **Theorem 4** (Safety Bound). *If an abstraction's state is within its safety bound, the system is guaranteed to obey its specification, i.e.,*

$$v_k \leq v_{max} \Rightarrow ||s_k||_S \leq 1 \quad \forall k \in \mathbb{N}_0. \tag{29}$$

Proof. After expanding the definitions, we apply Theorem 3 and the norm consistency. Assume $v_k \leq v_{max} (*)$, then

$$||s_k||_S \overset{(4e)}{=} ||C_s x_k||_S \overset{(1c)}{\leq} \underbrace{||C_s||_{SP}}_{\overset{(10)}{=} v_{max}^{-1}} ||x_k||_P \overset{(28)}{\leq} v_{max}^{-1} v_k \overset{(*)}{\leq} 1.$$ ◀

▶ **Lemma 5** (Worst-case Behavior). *The dynamic system* (11) *provides an upper bound on the observer abstraction* (8), *i.e.,*

$$v_k \leq \bar{v}_k \quad \forall k \in \mathbb{N}_0. \tag{30}$$

Proof. First, we decompose the measurement term and apply Theorem 3:

$$||L_{\sigma_k} y_k||_P \overset{(4b)}{\leq} ||L_{\sigma_k} C_{\sigma_k} x_k + L_{\sigma_k} H_{\sigma_k} z_k||_P \overset{(1c),(3c),(4c),(27)}{\leq}$$
$$\underbrace{||L_{\sigma_k} C_{\sigma_k}||_{PP}}_{\overset{(10)}{=}\gamma_{\sigma_k}} ||x_k||_P + \underbrace{||L_{\sigma_k} H_{\sigma_k}||_{PZ_{\sigma_k}}}_{\overset{(10)}{=}\delta_{\sigma_k}} \overset{(28)}{\leq} \gamma_{\sigma_k} v_k + \delta_{\sigma_k}. \tag{31}$$

We then prove (30) by induction:
Base Case: By definition, $v_0 \overset{(8b),(11b)}{=} \bar{v}_0$.
Inductive Assumption (I.A.): $v_k \leq \bar{v}_k$.
Inductive Step ($k \to k+1$):

$$v_{k+1} \overset{(8a),(31)}{\leq} (\rho_{\sigma_k} + \gamma_{\sigma_k}) v_k + \beta_{\sigma_k} + \delta_{\sigma_k} \overset{I.A.}{\leq} (\rho_{\sigma_k} + \gamma_{\sigma_k}) \bar{v}_k + \beta_{\sigma_k} + \delta_{\sigma_k} \overset{(11a)}{=} \bar{v}_{k+1}. \quad ◀$$

▶ **Lemma 6** (An Inclusion Condition). *The feasible set* (13) *obeys the following inclusion condition:*

$$a \leq b \Rightarrow \Sigma_f(b) \subseteq \Sigma_f(a). \tag{32}$$

Proof.

$$\Sigma_f(b) \overset{(13)}{=} \{\sigma \in \Sigma | b \leq v^*_\sigma\} \overset{a \leq b}{=} \{\sigma \in \Sigma | a \leq v^*_\sigma \vee b \leq v^*_\sigma\} \subseteq \{\sigma \in \Sigma | a \leq v^*_\sigma\} \overset{(13)}{=} \Sigma_f(a). \quad ◀$$

▶ **Theorem 7** (Recursive Feasibility). *Under the conditions* (14) *and* (15), *the switching policy* (16) *is recursivly feasible, i.e.,* $\Sigma_f(v_k) \neq \emptyset \; \forall k \in \mathbb{N}_0$. *Assume (w.l.o.g.) that* $v^*_0 \geq v_{max}$. *Then,*

$$0 \in \Sigma_f(v_k) \neq \emptyset \quad \forall k \in \mathbb{N}_0. \tag{33}$$

Proof. In preparation, observe that

$$\sigma \in \Sigma_f(v_k) \overset{(13)}{\Leftrightarrow} v_k \leq v^*_\sigma \overset{(12)}{=} \frac{v_{max} - \beta_\sigma - \delta_\sigma}{\rho_\sigma + \gamma_\sigma} \Leftrightarrow (\rho_\sigma + \gamma_\sigma) v_k + \beta_\sigma + \delta_\sigma \leq v_{max}. \tag{34}$$

We prove (33) by induction using the results from Lemma 6:
Base Case: By assuming (14) and (15), $v_0 \leq v_{max} \leq v^*_0$. Therefore:

$$0 \overset{(13)}{\in} \Sigma_f(v^*_0) \overset{(32)}{\subseteq} \Sigma_f(v_0).$$

Inductive Assumption (I.A.): $0 \in \Sigma_f(v_k)$.
Inductive Step ($k \to k+1$): Assuming the I.A. holds, we know that $\Sigma_f(v_k) \neq \emptyset$, therefore, using any policy $\sigma_k \in \Sigma_f(v_k)$ leads to

$$v_{k+1} \overset{(8a),(31)}{\leq} (\rho_{\sigma_k} + \gamma_{\sigma_k}) v_k + \beta_{\sigma_k} + \delta_{\sigma_k} \leq \max_{\sigma \in \Sigma_f(v_k)} ((\rho_\sigma + \gamma_\sigma) v_k + \beta_\sigma + \delta_\sigma) \overset{(34)}{\leq} v_{max} \overset{(15)}{\leq} v^*_0. \tag{35}$$

Using the definition of feasible sets, we arrive at $v_{k+1} \leq v^*_0 \overset{(13)}{\Leftrightarrow} 0 \in \Sigma_f(v_{k+1})$. ◀

▶ **Corollary 8** (Guaranteed Safety). *Under the conditions of Theorem 7, any switching policy obeying* (16) *guarantees the safety specification* (4f) $\forall k \in \mathbb{N}_0$.

Proof. $v_k \leq v_{max}$ holds for $k = 0$ due to (14) and for $k \in \mathbb{N}$ due to (15) and (33)–(35). Theorem 4 concludes (4f) $\forall k \in \mathbb{N}_0$. ◀

▶ **Theorem 9** (Double Integrator Stability). *In the absence of disturbances ($d_k = 0, z_k = 0 \forall k \in \mathbb{N}_0$), the double integrator benchmark* (26) *is exponentially stable under* (1, 5)*-switching, i.e., there exist some $C > 0, 0 < \lambda < 1$ such that*

$$||x_k||_P \leq C\lambda^k ||x_0||_P. \tag{36}$$

Proof. Given $d_k = 0, z_k = 0 \forall k \in \mathbb{N}_0$, the system dynamics (4a) have the explicit solution

$$||x_k||_P = ||\left(\prod_{i=0}^{k-1} A_{\sigma_i}\right) x_0||_P. \tag{37}$$

For any switching sequence satisfying the (1, 5) constraint, we can decompose the product (37) into n shorter sequences of the shape $M_m = A_0 A_1^m$ and some open loop part A_1^l with $m, l \in \{0, \ldots, 4\}$. Let $f(i)$ be the mapping for the concrete sequence $(**)$. Applying the consistency properties yields

$$\begin{aligned} ||x_k||_P &\overset{(37),(**)}{=} ||A_1^l \left(\prod_{i=1}^{n} M_{f(i)}\right) x_0||_P \overset{(1c),(1d)}{\leq} ||A_1^l||_{PP} \left(\prod_{i=1}^{n} ||M_{f(i)}||_{PP}\right) ||x_0||_P \\ &\overset{(3b),n\geq 0}{\leq} \underbrace{\max_{l\in\{0,\ldots,4\}} ||A_1^l||_{PP}}_{=:\tilde{C}} (\underbrace{\max_{m\in\{0,\ldots,4\}} ||M_m||_{PP}}_{=:\tilde{\lambda}})^n ||x_0||_P. \end{aligned}$$

Given that the M_m describe between one and five timesteps, we can bound n from below as $\lfloor \frac{k}{5} \rfloor \leq n$. Assuming $0 \leq \tilde{\lambda} \leq 1$, the exponential decay can be upper bounded as

$$\tilde{C}\tilde{\lambda}^n \leq \tilde{C}\tilde{\lambda}^{\lfloor \frac{k}{5} \rfloor} \overset{\lfloor \frac{k}{5} \rfloor \geq \frac{k}{5}-1}{=} \tilde{C}\tilde{\lambda}^{\frac{k}{5}-1} = \underbrace{\tilde{C}\tilde{\lambda}^{-1}}_{=:C} (\underbrace{\tilde{\lambda}^{\frac{1}{5}}}_{=:\lambda})^k.$$

Combining the above yields the stability condition (36), i.e., $||x_k||_P \leq \tilde{C}\tilde{\lambda}^n ||x_0||_P \leq C\lambda^k ||x_0||_P$.

The last step is to prove that $0 \leq \tilde{\lambda} \leq 1$ which can be shown for $P := \begin{bmatrix} 5.849 & 5.045 \\ 5.045 & 13.45 \end{bmatrix} > 0$ and (1b) yielding $\tilde{\lambda} = 0.9712$ and therefore stability with $C = 1.191$ and $\lambda = 0.9942$. ◀

Isospeed: Improving (min,+) Convolution by Exploiting (min,+)/(max,+) Isomorphism

Raffaele Zippo
Dipartimento di Ingegneria dell'Informazione, University of Firenze, Italy
Dipartimento di Ingegneria dell'Informazione, University of Pisa, Italy
Distributed Computer Systems Lab (DISCO), TU Kaiserslautern, Germany

Paul Nikolaus
Distributed Computer Systems Lab (DISCO), TU Kaiserslautern, Germany

Giovanni Stea
Dipartimento di Ingegneria dell'Informazione, University of Pisa, Italy

Abstract

(min,+) convolution is the key operation in (min,+) algebra, a theory often used to compute performance bounds in real-time systems. As already observed in many works, its algorithm can be computationally expensive, due to the fact that: i) its complexity is superquadratic with respect to the size of the operands; ii) operands must be extended *before* starting its computation, and iii) said extension is tied to the least common multiple of the operand periods.

In this paper, we leverage the isomorphism between (min,+) and (max,+) algebras to devise a new algorithm for (min,+) convolution, in which the need for operand extension is minimized. This algorithm is considerably faster than the ones known so far, and it allows us to reduce the computation times of (min,+) convolution by orders of magnitude.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Networks → Network performance analysis; Mathematics of computing → Mathematical software performance

Keywords and phrases Deterministic Network Calculus, min-plus algebra, max-plus algebra, performance, algorithms

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.12

Supplementary Material *Software (ECRTS 2023 Artifact Evaluation approved artifact)*: https://doi.org/10.4230/DARTS.9.1.3

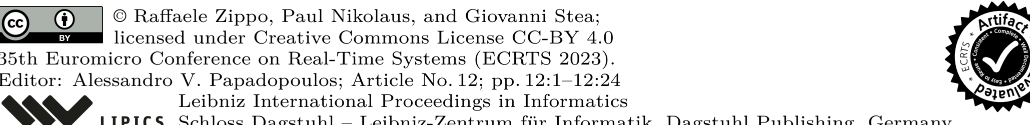
Funding This work was supported in part by the Italian Ministry of Education and Research (MIUR) in the framework of the FoReLab project (Departments of Excellence).

Acknowledgements This work is inspired by the results in [20] – we wish to thank Steffen Bondorf for pointing out this paper to us, as well as Raul-Paul Epure for suggestions with respect to some proofs.

1 Introduction

(min,+) and (max,+) algebras [2, 17] lie at the core of theories for the analysis of worst-case performance bounds.[1] More specifically, Deterministic Network Calculus (DNC) [10, 9, 8, 15, 4] – devised for network traffic – and Real-Time Calculus (RTC) [25] – devised for event-triggered systems – are both based on (min,+) and (max,+) algebra. Some recent papers analyzing real-time systems using DNC or RTC are [24, 19, 7, 3]. In both theories, flows of traffic (in DNC) or events (in RTC) are represented as cumulative functions of time,

[1] While commonly called *algebras* in DNC jargon, (min,+) and (max,+) are *semirings* [4, Ch. 2].



© Raffaele Zippo, Paul Nikolaus, and Giovanni Stea;
licensed under Creative Commons License CC-BY 4.0

35th Euromicro Conference on Real-Time Systems (ECRTS 2023).
Editor: Alessandro V. Papadopoulos; Article No. 12; pp. 12:1–12:24
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

counting the traffic (or the events) arrived up to time t. The service guarantees offered by elements where resource contention may occur (e.g., packet schedulers at the output of a network) are also represented as functions of time, called *curves*.[2] I/O relationships at a node, represented as operations in (min,+) and (max,+) algebra, allow one to derive worst-case performance guarantees, e.g., bounds on the transit delay of a flow, or the backlog at a node. These theories are *compositional*, i.e., they allow one to model complex systems by first modeling their elements in isolation, and then composing their models, again using operations in (min,+) or (max,+) algebra. For instance, in DNC, the minimum service that a packet scheduler guarantees to a flow traversing it is represented by a *service curve* β. Accordingly, if a flow traverses two such schedulers, having service curves β_1 and β_2, the minimum service that the latter is guaranteed on an end-to-end basis can be obtained by computing the *(min,+) convolution* of β_1 and β_2. A similar property exists in RTC. In this paper, we concentrate on properties of (min,+) algebra. However, given the strong similarities between the two theories, we will often use $(\cdot,+)$ when the discussion applies to both (min,+) and (max,+) algebra.

The issue of *automated computation* of $(\cdot,+)$ algebra expressions is relevant: algebraic expressions which look simple on paper can in fact require lengthy computations. The research community has therefore developed several software packages to automate this task. In doing so, they have addressed the problem of finding efficient data structures to represent functions and curves, and efficient algorithms to implement basic $(\cdot,+)$ algebra operations. Works [6, 4] provided an "algorithmic toolbox" for DNC: they showed that piecewise-affine functions that are ultimately pseudo-periodic (UPP) represent suitable models for both traffic and service guarantees, and provided algorithms for most $(\cdot,+)$ algebra operations. The toolbox was first implemented in the COINC free library [5], which is not available anymore, and later by the commercial library RTaW-Pegase [21] and the open-source library Nancy [28]. A very similar model [4, p. 95], called *variability characterization curves* (VCCs), was implemented by the RTC toolbox [26]. Broadly speaking, in both UPP and VCC models functions and curves are represented as *sequences of segments*, and periodicity is leveraged to allow functions defined in $[0, +\infty)$ to be represented with a finite amount of information.

One of the most important operations in (min,+) algebra is (min,+) convolution. The latter has a complexity which is superquadratic with the size of its operands (i.e., the number of segments in their sequences), which makes it computationally expensive. What is worse, the algorithms that compute these operations require that the sequences of operands must be *extended* beforehand, which reflects on the overall complexity. Operand extension, in turn, depends heavily on numerical properties of the operands themselves, and is often related to the hyperperiod, i.e., the least common multiple (lcm) of the period lengths of the operands. The impact of the above issue grows exponentially when operations are chained together, which limits the scope of the studies that one can do in practice. Different techniques have been proposed in the literature to mitigate or avoid this issue, such as using containers and inclusion functions [16], avoiding the periodic parts altogether by bounding the study a priori [12, 13, 14], using a posteriori representation minimization to mitigate the impact on chained operations [29], devising more efficient algorithms for specific subclasses of operands and operations [29], namely subadditive functions and (min,+) convolution.

In this paper, we provide a novel technique to reduce the computation cost of (min,+) convolution, under general hypotheses on the operands, which we call *isospeed*. Our technique relies on exploiting the isomorphism between (min,+) and (max,+) algebra, thoroughly

[2] To be precise, in a curve an abscissa τ describes what may happen in any interval of length τ.

described in [17]. It is shown therein that the result of a (min,+) convolution can be obtained by computing the *isomorphic* operation, i.e., the (max,+) convolution, if one applies a simple transformation to the operands beforehand, and to the result afterwards. This transformation is called *pseudoinversion*. This implies that there are always two ways to compute a (min,+) convolution: the *direct* one, e.g., using the algorithm described in [4], and the *inverse* one – i.e., the one based on pseudoinversion and isomorphism. This was first observed in [20]: the authors found that – in their specific use cases – the inverse algorithm for (min,+) convolution was considerably faster than the direct one. However, they did not provide an explanation as to why. Lacking the above, one cannot know whether this is a general result or just a stroke of luck. In this paper, we investigate the above in depth, providing novel results of both theoretical and practical significance. First, we offer a cogent explanation for the empirical observation in [20], which allows us to understand when and why one algorithm will be faster than the other *a priori*, and when instead no conclusion can be drawn. This is because the complexity of (min,+) convolution depends on numerical properties of the operands, and pseudoinversion modifies them, making the computational cost of the inverse algorithm different from the direct one's. Then, we build on the above observation to devise a new algorithm, that reduces the operand extension and the runtime of (min,+) convolution to a minimum. This makes it no worse (barring negligible overhead), and often much better, than the *fastest* between the direct algorithm and inverse one. Our algorithm improves the feasibility of performance studies that cannot benefit from the other methods mentioned above. For instance, it applies regardless of the shape of the operands, unlike the algorithms in [29], which requires operands to be subadditive, or the method described in [12, 13, 14], which requires service curves to be superadditive. Moreover, it optimizes single (min,+) convolutions, unlike the representation minimization in [29], which is only beneficial when *chaining* operations and does nothing to optimize individual ones. While the two methods can work in conjunction, individual (min,+) convolutions may take minutes or more using the standard algorithm. Due to space limitations, we can only focus on (min,+) convolution in this paper. However, our results can be generalized to (max,+) convolution as well, which can be improved in the same way, with the same performance benefits. This is because the isomorphism, as the name implies, works both ways.

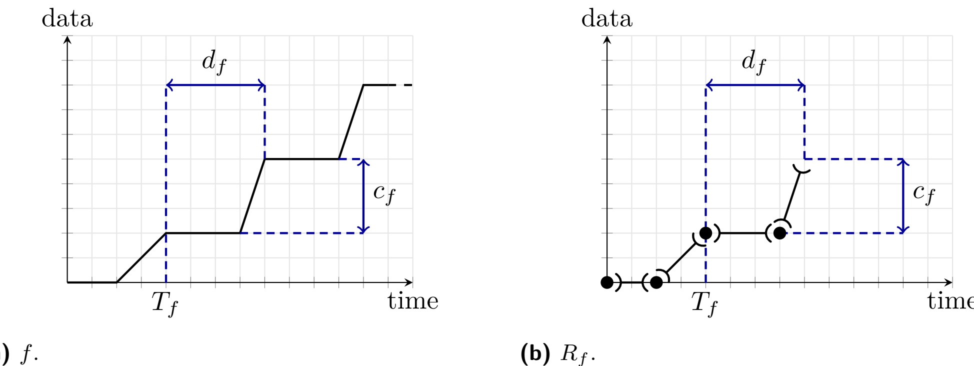
The rest of this paper is organized as follows: In Section 2, we introduce the mathematical background and state of the art. Then, in Section 3, we present our isospeed algorithm, which is evaluated in Section 4. Section 5 concludes the paper and highlights future work.

2 Background and Notation

In this section, we provide the mathematical and algorithmic background required for this paper. We define the types of functions that we use and the operations that we aim to improve. We use $a \wedge b = \min(a, b)$ and $a \vee b = \max(a, b)$. We also use $\mathbb{N}_0$ to denote the set of non-negative integers $\{0, 1, 2, 3, \dots\}$, $\mathbb{N}$ for the set of strictly positive integers $\{1, 2, 3, \dots\}$, and $\mathbb{Q}_+$ the set of non-negative rationals (including 0).

2.1 The UPP Model

To implement DNC and RTC computations in software, one needs to provide finite representations of functions and well-formed algorithms for (·,+) operations. In this paper, we use the model discussed in [29, 27] and implemented in Nancy [28], which we also use to implement our optimization. According to the widely accepted approach described in [6, 4],



(a) f.

(b) R_f.

Figure 1 An ultimately pseudo-periodic piecewise-affine function f and its representation R_f.

a sufficiently generic class of functions useful for $(\cdot,+)$ computations is the set $\mathcal{U}$ of (i) ultimately pseudo-periodic $\mathbb{Q}_+ \to \mathbb{Q} \cup \{+\infty, -\infty\}$ (ii) piecewise affine functions. We define both properties (i) and (ii) separately:

▶ **Definition 1** (Ultimately Pseudo-Periodic Function [6, p. 8]). *Let f be a function $\mathbb{Q}_+ \to \mathbb{Q} \cup \{+\infty, -\infty\}$. Then, f is ultimately pseudo-periodic (UPP) if exists $T_f \in \mathbb{Q}_+, d_f \in \mathbb{Q}_+ \setminus \{0\}, c_f \in \mathbb{Q} \cup \{+\infty, -\infty\}$ such that*

$$f(t + k \cdot d_f) = f(t) + k \cdot c_f, \qquad \forall t \geq T_f, \forall k \in \mathbb{N}. \tag{1}$$

We call T_f the pseudo-periodic start*, or length of the initial transient, d_f the* period length*, and c_f the* period height*. We also say that f is UPP* from T_f.

▶ **Definition 2** (Piecewise Affine Function [6, p. 9]). *We say that a function f is* piecewise affine *(PA) if there exists an increasing sequence $(a_i), i \in \mathbb{N}_0$ which tends to $+\infty$, such that $a_0 = 0$ and $\forall i \in \mathbb{N}_0$, it either holds that $f(t) = b_i + \rho_i t$ for some $b_i, \rho_i \in \mathbb{Q}$, or $f(t) = +\infty$, or $f(t) = -\infty$ for all $t \in \,]a_i, a_{i+1}[$.*

We remark that functions in $\mathcal{U}$ are not necessarily non-decreasing, and can assume infinite values. Both these properties are useful for algebraic manipulations. Among the functions in $\mathcal{U}$, we distinguish *Ultimately Constant* (UC) ones. A function is UC if there exists a $T \in \mathbb{Q}_+$ such that $f(t) = f(T)\, \forall t \geq T$. Similarly, an *Ultimately Infinite* (UI) function is one such that $f(t) = +\infty$, or $f(t) = -\infty$ for all $t \geq T$. Typical cases of UI curves in DNC are the service curves of *delay elements*. Throughout this paper, we will exclude UC and UI functions, because some properties do not hold otherwise. This limitation is of negligible impact, since (min,+) convolution is computationally trivial when an operand is UC/UI.

For functions in $\mathcal{U}$, it is enough to store a representation of the initial transient part and of one period, which is a finite amount of information. This is exemplified in Figure 1. Accordingly, we call a *representation* R_f of a function f the tuple (S, T, d, c), where T, d, c are the values described above, and S is a sequence of points and open segments describing f in $[0, T + d[$. We use both points and open segments in order to easily model discontinuities. We will use the umbrella term *elements* to encompass both when convenient. We denote with $n(S)$ the cardinality of a sequence S, i.e., the number of its elements. Note that, given R_f, one can compute $f(t)$ for all $t \geq 0$, and also S_f^I, i.e., a sequence describing f in the finite interval I for any $I \subset \mathbb{Q}_0^+$. Furthermore, being finite, R_f can be used as a data structure to represent f in code. As it will be useful in the following, we define *Cut* to be an (obvious) algorithm that, given R_f and an interval I, computes S_f^I. With a little abuse of notation,

we will use $(\cdot,+)$ operators directly on finite sequences such as S_f^I. For instance, given the (min,+) convolution (formally defined later), we will write $S_f^{I_f} \otimes S_g^{I_g}$ to express that we are computing the (min,+) convolution $f \otimes g$, limited to the values of f in interval I_f and those of g in interval I_g. It will be useful in the following to consider a function $f \in \mathcal{U}$ in a *restricted support* D.[3] This is done as follows:

▶ **Definition 3** (Min and Max Restrictions). *Let $f \in \mathcal{U}$ and $D \subseteq \mathbb{Q}_+$. Then, its* min restriction *over a support D is defined as*

$$f|_D^\wedge := \begin{cases} f(t), & \text{if } t \in D, \\ +\infty, & \text{otherwise.} \end{cases}$$

Moreover, its max restriction *over a support D is defined as*

$$f|_D^\vee := \begin{cases} f(t), & \text{if } t \in D, \\ -\infty, & \text{otherwise.} \end{cases}$$

In this work, we will often consider D to be an interval I of the form $[0, a[$ or $[a, +\infty[$. In many cases, it will be useful to restrict a function to its *transient part*, i.e, to interval $I = [0, T_f[$, or to its *periodic part*, i.e., $I = [T_f, +\infty[$, using shorthands $f_t^\wedge$ and $f_t^\vee$, as well as $f_p^\wedge$ and $f_p^\vee$, respectively. Accordingly, one can decompose f as $f = f_t^\wedge \wedge f_p^\wedge$ or $f = f_t^\vee \vee f_p^\vee$.

A $(\cdot,+)$ operator can be defined computationally as an algorithm that takes UPP representations of its input functions and yields a UPP representation of the result. Considering a generic binary operator[4] $[\cdot] * [\cdot]$, in order to compute $f * g$ we need an algorithm that computes R_{f*g} from R_f and R_g, i.e., $R_f, R_g \rightarrow R_{f*g}$. We call this *by-curve* algorithm. Such an algorithm consists of the following steps:

1. compute valid parameters T_{f*g}, d_{f*g} and c_{f*g} for the result.
2. compute the intervals I_f and I_g, for the sequences $S_f^{I_f} = \text{Cut}(R_f, I_f)$ and, likewise, $S_g^{I_g}$;
3. compute $S_f^{I_f}, S_g^{I_g} \rightarrow S_{f*g}^{I_{f*g}}$ where $I_{f*g} = [0, T_{f*g} + d_{f*g}[$, i.e., use an algorithm that computes the resulting sequence from the sequences of the operands. We call this *by-sequence* algorithm for operator $[\cdot] * [\cdot]$;
4. return $R_{f*g} = (S_{f*g}, T_{f*g}, d_{f*g}, c_{f*g})$.

The *by-curve* algorithm for operator $[\cdot] * [\cdot]$ allows us to compute the result with any operands. Works [6, 4] provide such computational descriptions for most DNC operators, such as $(\cdot,+)$ convolution and deconvolution, while [27] provides the same for pseudoinverses (formally defined in the next section).

▶ Remark 4. Parameters T_{f*g}, d_{f*g} and c_{f*g}, as well as intervals I_f and I_g, are *sufficient* to compute a representation R_{f*g}. There may be, in general, more than one way to compute them for an algorithm, resulting in different performance.

Intuitively, dealing with shorter sequences leads to faster *by-sequence* algorithms. Optimized parameters and intervals can be found either by making restrictive assumptions on the shape of the operands, or – as we do in this paper – by exploiting algebraic properties.

[3] Inspired by [6, p. 7], we use *support* D to assign a subset outside of which the function is constantly $-\infty$ or $+\infty$. Note that this does not necessarily mean, in general, that f is finite on D. We mostly use it to define a set within which the properties of f are observed.

[4] The same process applies also, with minor adjustments, to unitary operators.

2.2 Upper and Lower Pseudoinverses

It was first shown in [17] that (min,+) and (max,+) algebra can be regarded as specular images of each other. In fact, results in one algebra can be mapped to the other via *pseudoinversion* of operands and results. This *isomorphism* will be exploited throughout this paper. Hereafter, we discuss the essential definitions and properties of pseudoinverses, taken from [27].[5]

▶ **Definition 5** (Lower and Upper Pseudoinverse). *Let $f \in \mathcal{U}$ be non-decreasing. Then its* lower pseudoinverse $f_\downarrow^{-1}$ *and its* upper pseudoinverse $f_\uparrow^{-1}$ *are defined as*

$$f_\downarrow^{-1}(y) := \inf \{t \geq 0 \mid f(t) \geq y\},$$
$$f_\uparrow^{-1}(y) := \sup \{t \geq 0 \mid f(t) \leq y\}.$$

The lower pseudoinverse is always left-continuous and the upper pseudoinverse is right-continuous. Moreover, upper and lower pseudoinverses can be combined to yield something close to an involutive property. Consider a non-decreasing function f. Then, if f is left-continuous, $f(t) = \left(f_\uparrow^{-1}\right)_\downarrow^{-1}(t)$. If f is right-continuous, $f(t) = \left(f_\downarrow^{-1}\right)_\uparrow^{-1}(t)$. UPP properties and algorithms for the pseudoinverses are summarized here.

▶ **Theorem 6** ([27], Theorem 9, Theorem 10). *Let f be a non-decreasing UPP function that is neither UC nor UI. Then, $f_\downarrow^{-1}$ and $f_\uparrow^{-1}$ are function of $\mathcal{U}$ with*

$$T_{f_\downarrow^{-1}} = f\left(T_f + d_f\right), \tag{2}$$
$$T_{f_\uparrow^{-1}} = f\left(T_f\right) \tag{3}$$
$$d_{f_\downarrow^{-1}} = d_{f_\uparrow^{-1}} = c_f, \tag{4}$$
$$c_{f_\downarrow^{-1}} = c_{f_\uparrow^{-1}} = d_f. \tag{5}$$

The exact algorithm for upper/lower pseudoinverses is reported in [27]. In both cases, it can be computed in linear time with the operand's sequence size, i.e., it is $\mathcal{O}\left(n(S)\right)$. This makes pseudoinversion considerably less complex than (min,+) convolution, which – as we will discuss below – is superquadratic. Both operations yield a result whose sequence has a cardinality *similar* to its operand's. The two cardinalities are not exactly *equal* because constant segments in the operand map to discontinuities in the pseudoinverse and vice versa. Segments count as elements in the cardinality, whereas discontinuities do not. This will be recalled later on, when we discuss performance.

2.3 (min,+) and (max,+) Convolution

Convolution is one of the most common operations in (·,+) algebra. We introduce here both (min,+) and (max,+) convolution.

▶ **Definition 7** (Convolution in (min,+) / (max,+) Algebra). *Let f, g be non-decreasing. Their (min,+) convolution is defined for all $t \geq 0$ as*

$$f \otimes g(t) := \inf_{0 \leq s \leq t} \left\{f(s) + g(t-s)\right\}.$$

[5] These definitions differ from the ones in [17, p. 60]. In fact, [17] considers functions defined in $\mathbb{R}$, hence having no boundaries, whereas functions in $\mathcal{U}$ are defined in $\mathbb{Q}_+$, hence 0 and $f(0)$ constitute a boundary.

Their (max,+) convolution is defined for all $t \geq 0$ as

$$f \overline{\otimes} g(t) \coloneqq \sup_{0 \leq s \leq t} \{f(s) + g(t-s)\} .$$

A fundamental result, which uses the above properties, is the isomorphism between (min,+) and (max,+) convolution, which enables us to replace one with the other, via pseudoinversion of operands and results.

▶ **Theorem 8** (Isomorphism of Convolution For Left-Continuous Functions $\in \mathcal{U}$)**.** *Let $f, g \in \mathcal{U}$ be left-continuous and non-decreasing. Then,*

$$(f \otimes g)_{\uparrow}^{-1} = \left(f_{\uparrow}^{-1}\right) \overline{\otimes} \left(g_{\uparrow}^{-1}\right) . \tag{6}$$

A proof can be derived by following along the lines of [20, Theorem 1], [17, Theorem 10.3b], but adapting to the fact that we consider functions $\in \mathcal{U}$. As a consequence, since the (min,+) convolution of left-continuous functions is itself left-continuous, we obtain:

$$f \otimes g = \left((f \otimes g)_{\uparrow}^{-1}\right)_{\downarrow}^{-1} = \left(f_{\uparrow}^{-1} \overline{\otimes} g_{\uparrow}^{-1}\right)_{\downarrow}^{-1} . \tag{7}$$

The algorithms for $(\cdot, +)$ convolution of UPP curves require one to specialize the generic steps described in Section 2.1. Due to space limitations, we discuss in depth (min,+) convolution only. (max,+) convolution can be presented along the same lines.

It was proved in [6] that (min,+) convolution $f \otimes g$ can be computed if one decomposes its operands into their transient and periodic parts, according to Definition 3. More specifically, the procedure is as follows:

1. Decompose the operands as $f = f_t^{\wedge} \wedge f_p^{\wedge}$ and $g = g_t^{\wedge} \wedge g_p^{\wedge}$.
2. Compute partial convolutions involving at least one transient part: $h_{tt} \coloneqq f_t^{\wedge} \otimes g_t^{\wedge}$, $h_{tp} \coloneqq f_t^{\wedge} \otimes g_p^{\wedge}$, $h_{pt} \coloneqq f_p^{\wedge} \otimes g_t^{\wedge}$. These can be computed using the algorithms described in [6]. These computations are not particularly complex, since at least one of the operands is defined in a finite interval.
3. Compute the partial convolution of the *periodic* parts, $h_{pp} \coloneqq f_p^{\wedge} \otimes g_p^{\wedge}$. This is the computationally complex part, as we detail below.
4. Compute $f \otimes g = h_{tt} \wedge h_{tp} \wedge h_{pt} \wedge h_{pp}$.

In [6, Proposition 4.5], UPP properties are derived for all these parts (h_{tt}, h_{tp}, h_{pt}, h_{pp}) and their minimum. We summarize this result here as Proposition 9.

▶ **Proposition 9** ([6], Proposition 4.5)**.** *Let $f, g \in \mathcal{U}$. Then their (min,+) convolution $f \otimes g$ is again $\in \mathcal{U}$. Moreover,*

$$d_{f \otimes g} = \operatorname{lcm}(d_f, d_g) , \tag{8}$$

$$c_{f \otimes g} = d_{f \otimes g} \cdot \min\left(\frac{c_f}{d_f}, \frac{c_g}{d_g}\right) = \operatorname{lcm}(d_f, d_g) \cdot \min\left(\frac{c_f}{d_f}, \frac{c_g}{d_g}\right) \tag{9}$$

are a period length and height for $f \otimes g$.

From the proof of [6, Proposition 4.5] we can extract the following result for, in particular, the convolution of periodic parts.

▶ **Corollary 10** ((min,+) convolution of periodic parts.). *Let f and $g \in \mathcal{U}$. Then, $h_{pp} = f_p^{\wedge} \otimes g_p^{\wedge}$ is again a function of $\mathcal{U}$ with*

$$d_{h_{pp}} = d_{f \otimes g} = \operatorname{lcm}(d_f, d_g)\,, \tag{10}$$

$$c_{h_{pp}} = c_{f \otimes g} = d_{h_{pp}} \cdot \min\left(\frac{c_f}{d_f}, \frac{c_g}{d_g}\right), \tag{11}$$

$$T_{h_{pp}} = T_f + T_g + \operatorname{lcm}(d_f, d_g)\,. \tag{12}$$

And that, in order to compute $f_p^{\wedge} \otimes g_p^{\wedge}$, it is sufficient to use

$$\begin{aligned} I_{f_p^{\wedge}} &= \left[T_f, T_f + 2 \cdot d_{h_{pp}}\right[\,, \\ I_{g_p^{\wedge}} &= \left[T_g, T_g + 2 \cdot d_{h_{pp}}\right[\,, \\ I_{h_{pp}} &= \left[T_f + T_g, T_f + T_g + 2 \cdot d_{h_{pp}}\right[\,. \end{aligned} \tag{13}$$

Corollary 10 shows that the period of the convolution of the periodic parts depends on the lcm of the periods of the operands. Algorithm 1 reports the pseudocode for the (min,+) convolution algorithm, based on the above decomposition [6]. When computing the convolution of the periodic parts, operands have to be *extended*, i.e., computed in intervals $I_{f_p^{\wedge}}$, $I_{g_p^{\wedge}}$. The complexity of the by-sequence algorithm for the convolution (line 6) is [6, p. 43]

$$\mathcal{O}\left(n\left(S_{f_p^{\wedge}}^{I_{f_p^{\wedge}}}\right) \cdot n\left(S_{g_p^{\wedge}}^{I_{g_p^{\wedge}}}\right) \cdot \log\left(n\left(S_{f_p^{\wedge}}^{I_{f_p^{\wedge}}}\right) \cdot n\left(S_{g_p^{\wedge}}^{I_{g_p^{\wedge}}}\right)\right)\right). \tag{14}$$

However, domains $I_{f_p^{\wedge}}$, $I_{g_p^{\wedge}}$ depend on $\operatorname{lcm}(d_f, d_g)$. On one hand, this corroborates the observation that computing h_{pp} is the most complex task. On the other hand, the number of operations required may vary considerably depending on *numerical properties* of the operands. In fact, $\operatorname{lcm}(d_f, d_g)$ ranges from $\max(d_f, d_g)$ to the product of the numerators of d_f and d_g[6]. Therefore, the runtime of the convolution of the periodic parts may vary a lot.

We briefly discuss (max,+) convolution, to highlight that it can be computed via the same decomposition, at similar big-O complexity, as the (min,+) convolution [6]. The main difference is that, since a *supremum* is used in place of an *infimum*, the decomposition is based on the maximum, rather than the minimum, of the parts. The procedure is as follows:

1. Decompose the operands as $f = f_t^{\vee} \vee f_p^{\vee}$ and $g = g_t^{\vee} \vee g_p^{\vee}$.
2. Compute partial convolutions involving at least one transient part: $\overline{h}_{tt} := f_t^{\vee} \overline{\otimes} g_t^{\vee}$, $\overline{h}_{tp} := f_t^{\vee} \overline{\otimes} g_p^{\vee}$, $\overline{h}_{pt} := f_p^{\vee} \overline{\otimes} g_t^{\vee}$. These can be computed by adapting the algorithms for the (min,+) convolution described in [6]. Again, these computations are not particularly complex, since at least one of the operands is defined in a finite interval.
3. Compute the partial convolution of the *periodic* parts, $\overline{h}_{pp} := f_p^{\vee} \overline{\otimes} g_p^{\vee}$.
4. Compute $f \overline{\otimes} g = \overline{h}_{tt} \vee \overline{h}_{tp} \vee \overline{h}_{pt} \vee \overline{h}_{pp}$.

When computing the convolution of the periodic parts, operands have to be computed in intervals $I_{f_p^{\vee}}$, $I_{g_p^{\vee}}$, which do depend on $d_{\overline{h}_{pp}} = \operatorname{lcm}(d_f, d_g)$. The worst-case complexity of the by-sequence algorithm for the (max,+) convolution can be derived following the same steps as for the (min,+) convolution in [6, p. 43], and is the same as in (14), provided that one substitutes $S_{x_p^{\vee}}^{I_{x_p^{\vee}}}$ for $S_{x_p^{\wedge}}^{I_{x_p^{\wedge}}}$ for both operands x. Like with (min,+) convolution, domains $I_{f_p^{\vee}}$, $I_{g_p^{\vee}}$ depend on $\operatorname{lcm}(d_f, d_g)$. Therefore, the same observations already discussed apply here as well: computing $\overline{h}_{pp}$ is the most complex task, and the number of operations it requires may vary considerably depending on *numerical properties* of the operands.

[6] We recall that the lcm of two fractions is the lcm of their numerators divided by the greatest common divisor of their denominators.

Algorithm 1 Pseudocode for (min,+) convolution.

Input Functions f and g.
Return Their (min,+) convolution $f \otimes g$.

1: Decompose the operands as $f = \min\left(f_t^\wedge, f_p^\wedge\right)$ and $g = \min\left(g_t^\wedge, g_p^\wedge\right)$
2: Compute $h_{tt} \coloneqq f_t^\wedge \otimes g_t^\wedge$, $h_{tp} \coloneqq f_t^\wedge \otimes g_p^\wedge$, $h_{pt} \coloneqq f_p^\wedge \otimes g_t^\wedge$ as described in [6]
3: Compute $h_{pp} \coloneqq f_p^\wedge \otimes g_p^\wedge$ as follows:
4: Let $d_{h_{pp}} = \operatorname{lcm}(d_f, d_g)$; $c_{h_{pp}} = d_{h_{pp}} \cdot \min\left(\frac{c_f}{d_f}, \frac{c_g}{d_g}\right)$; $T_{h_{pp}} = T_f + T_g + d_{h_{pp}}$.
5: Let

$$I_{f_p^\wedge} = \left[T_f, T_f + 2 \cdot d_{h_{pp}}\right[;$$
$$I_{g_p^\wedge} = \left[T_g, T_g + 2 \cdot d_{h_{pp}}\right[;$$
$$I_{h_{pp}} = \left[T_f + T_g, T_f + T_g + 2 \cdot d_{h_{pp}}\right[.$$

6: Compute $S_{h_{pp}}^{I_{h_{pp}}} = S_{f_p^\wedge}^{I_{f_p^\wedge}} \otimes S_{g_p^\wedge}^{I_{g_p^\wedge}}$
7: $R_{h_{pp}} = \left(S_{h_{pp}}^{I_{h_{pp}}}, T_{h_{pp}}, d_{h_{pp}}, c_{h_{pp}}\right)$
8: $f \otimes g = \min\left(h_{tt}, h_{tp}, h_{pt}, h_{pp}\right)$

3 Improving the Runtime of (min,+) Convolution

This section reports our contributions. First, we observe that there are always two algorithms to compute the (min,+) convolution, and discuss *why* one can be faster than the other. Our observations motivate our improved (min,+) convolution algorithm, which outperforms both the above.

3.1 Alternative Algorithms for (min,+) Convolution

The algorithms for (min,+) and (max,+) convolution are very similar, and they have the same complexity *on the same operands*. However, a (min,+) convolution $f \otimes g$ can be computed via a (max,+) convolution of *pseudoinverse operands* $f_\uparrow^{-1}$, $g_\uparrow^{-1}$, as per (7) [20].

This means that one can *always* compute a (min,+) convolution in two different ways:

1. the *direct* method, using Algorithm 1;
2. the *inverse* method, using pseudoinversion of the operands, (max,+) convolution, and pseudoinversion of the result, via (7).

Now, both methods have the same big-O complexity. In fact, pseudoinversion is linear, and – as explained in the previous section – both (min,+) and (max,+) convolutions are superquadratic. Despite this, it was observed in [20] that the inverse algorithm was significantly *faster* than the direct one, which is counterintuitive, since pseudoinversion adds overhead.[7]

We present here a sound explanation of the above phenomenon, which is missing in [20], and serves as a basis for our improved method. In both the (min,+) and (max,+) convolutions, the dominant factor for the complexity is the length of the *extended* sequences (as per (14)), which depends on the *number of periods that each operand must be extended*. For the

[7] In [20], this result was presented within the context of VCC curves using RTC Toolbox [26] and event-based service curves, which differs from the one referenced here, which is instead based on the UPP model implemented in Nancy [28]. However, this distinction does not affect the following discussion.

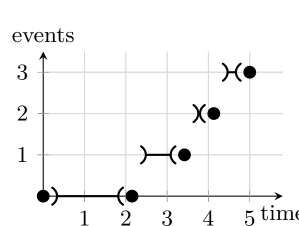


Figure 2 Example of event-based service curve used in [20], having abscissas in $\mathbb{R}_+$ and ordinates in $\mathbb{N}_0$.

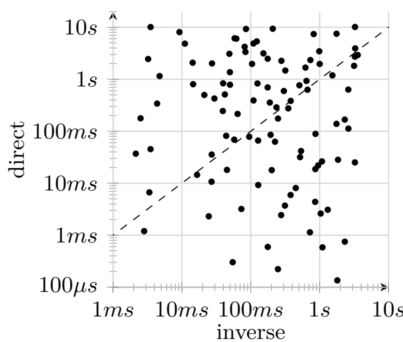


Figure 3 runtimes of the *direct* vs. *inverse* algorithms for (min,+) convolution.

direct algorithm, such extension occurs on hyperperiod $\mathrm{lcm}(d_f, d_g)$. Hence, we can write $\mathrm{lcm}(d_f, d_g) = k_{d_f} \cdot d_f = k_{d_g} \cdot d_g$. We call k_{d_f} and k_{d_g} *extension multipliers.* Computing the cut of f in $\left[T_f, T_f + 2 \cdot d_{h_{pp}}\right[$ (13) entails extending f by $2 \cdot k_{d_f}$ periods – and g by $2 \cdot k_{d_g}$.

On the other hand, the *inverse* algorithm uses pseudoinversion of the operands, which *swaps their period lengths* d_f, d_g *with their period heights* c_f, c_g. Consider in fact computing $f \otimes g$ via (7). We obtain for the inner function $f_\uparrow^{-1} \overline{\otimes} g_\uparrow^{-1}$, using Theorem 6,

$$
\begin{aligned}
d_{f_\uparrow^{-1} \overline{\otimes} g_\uparrow^{-1}} &= \mathrm{lcm}\left(d_{f_\uparrow^{-1}}, d_{g_\uparrow^{-1}}\right) = \mathrm{lcm}(c_f, c_g)\,, \\
c_{f_\uparrow^{-1} \overline{\otimes} g_\uparrow^{-1}} &= \max\left(\frac{c_{f_\uparrow^{-1}}}{d_{f_\uparrow^{-1}}}, \frac{c_{g_\uparrow^{-1}}}{d_{g_\uparrow^{-1}}}\right) \cdot d_{f_\uparrow^{-1} \overline{\otimes} g_\uparrow^{-1}} = \max\left(\frac{d_f}{c_f}, \frac{d_g}{c_g}\right) \cdot \mathrm{lcm}(c_f, c_g)\,.
\end{aligned}
\tag{15}
$$

Thus, in the *inverse* algorithm, the hyperperiod is $\mathrm{lcm}(c_f, c_g) = k_{c_f} \cdot c_f = k_{c_g} \cdot c_g$, and the extension multipliers are instead k_{c_f} and k_{c_g}.

Both algorithms have the same complexity, but the operands they work on may have considerably different size (i.e., the cardinalities of their extended sequences), hence their runtime can be vastly different. For instance, if $k_{d_f} > k_{c_f}$ and $k_{d_g} > k_{c_g}$, the inverse algorithm will be faster. This is likely the case in the experiments of work [20], which uses event-based service curves, exemplified in Figure 2. However, depending on the parameters of the operands, two more cases can be given, i.e.:

- $k_{d_f} < k_{c_f}$ and $k_{d_g} < k_{c_g}$, in which case the direct algorithm will generally be faster;
- $k_{d_f} < k_{c_f}$ and $k_{d_g} > k_{c_g}$ (or vice versa), in which case the comparison is inconclusive.

Figure 3 compares the direct and the inverse approach. We generated 100 pairs of operands randomly, and reported the runtimes of each (min,+) convolution as coordinates of points on the cartesian plane, with the direct algorithm on the ordinates and the inverse one on the abscissas. The horizontal (or vertical) distance between a point and the bisector indicates the difference (in orders of magnitude) between choosing one algorithm or the other. The above figure clearly shows that the comparison may swing either way, *and* that there is a lot to be gained in choosing wisely.

Hereafter, we follow up on the above observations. We show that the (min,+)/(max,+) isomorphism holds in more general settings than those described in Section 2, and that this, in turn, can be leveraged to define an improved algorithm for (min,+) convolution.

3.2 Exploiting Isomorphism to Speed up (min,+) Convolution

As discussed above, the most expensive part of (min,+) convolution is computing $h_{pp} = f_p^\wedge \otimes g_p^\wedge$, and this is due to the problem of operand extension. We have observed in the previous section that pseudoinversion may considerably alter the way operands are extended. Our intuition is that we can *limit the extension of each operand individually* to the minimum of what the direct and inverse algorithms would do, i.e., we can always choose *the smallest extension multiplier* operand by operand, independently. This will allow h_{pp} to be computed using smaller sequences, in considerably less time. We obtain this result via incremental steps. First, we show that isomorphism allows us to find *another set of parameters* for h_{pp}.

▶ **Theorem 11.** *Let $f, g \in \mathcal{U}$ be left-continuous and non-decreasing functions. Then, $h_{pp} = f_p^\wedge \otimes g_p^\wedge$ is again a function of $\mathcal{U}$ with*

$$d_{h_{pp}} = \max\left(\frac{d_f}{c_f}, \frac{d_g}{c_g}\right) \cdot \operatorname{lcm}(c_f, c_g) = \max\left(k_{c_g} \cdot d_g, k_{c_f} \cdot d_f\right), \tag{16}$$

$$c_{h_{pp}} = \operatorname{lcm}(c_f, c_g), \tag{17}$$

$$T_{h_{pp}} = \sup\left\{t \geq T_f + T_g \mid f_p^\wedge \otimes g_p^\wedge(t) \leq f(T_f) + g(T_g) + \operatorname{lcm}(c_f, c_g)\right\}. \tag{18}$$

The proof is reported in Appendix A.2. Now, since *both* Corollary 10 and Theorem 11 compute valid parameters for h_{pp}, we can always use the *minimum* of each:

$$d_{h_{pp}} = \min\left(\operatorname{lcm}(d_f, d_g), \max\left(k_{c_g} \cdot d_g, k_{c_f} \cdot d_f\right)\right), \tag{19}$$

$$c_{h_{pp}} = d_{h_{pp}} \cdot \min\left(\frac{c_f}{d_f}, \frac{c_g}{d_g}\right), \tag{20}$$

$$\begin{aligned} T_{h_{pp}} = \min(&T_f + T_g + \operatorname{lcm}(d_f, d_g), \\ &\sup\left\{t \geq T_f + T_g \mid f_p^\wedge \otimes g_p^\wedge(t) \leq f(T_f) + g(T_g) + \operatorname{lcm}(c_f, c_g)\right\}). \end{aligned} \tag{21}$$

Then, we show that we can find alternative *cuts* of $f_p^\wedge$ and $g_p^\wedge$ required to compute h_{pp}.

▶ **Corollary 12.** *Given f and $g \in \mathcal{U}$ which are left-continuous and non-decreasing in $[T_f, +\infty[$ and $[T_g, +\infty[$, respectively, and are neither UC nor UI. Then, to compute $f_p^\wedge \otimes g_p^\wedge$ via $(\max, +)$ isomorphism of restricted functions, it is sufficient to use sequences $S_{f_p^\wedge}^{I'_{f_p^\wedge}}$ and $S_{g_p^\wedge}^{I'_{g_p^\wedge}}$, with*

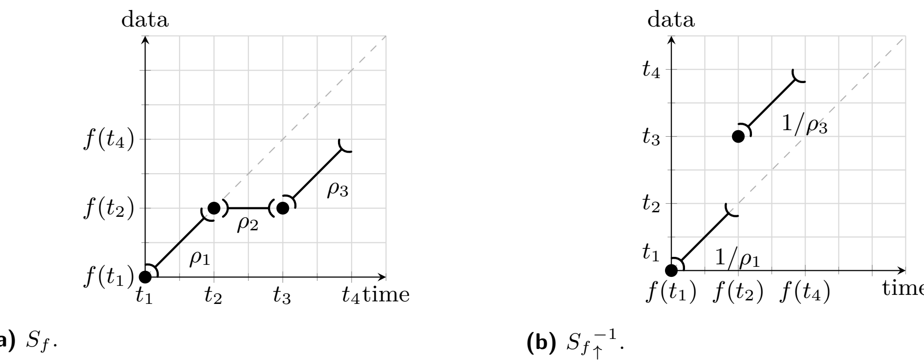
$$I'_{f_p^\wedge} = \left[T_f, T'_f + 2 \cdot k_{c_f} \cdot d_f\right], I'_{g_p^\wedge} = \left[T_g, T'_g + 2 \cdot k_{c_g} \cdot d_g\right]. \tag{22}$$

where we used $T'_f = \sup\{t \geq T_f \mid f(t) = f(T_f)\}$ and $T'_g = \sup\{t \geq T_g \mid g(t) = g(T_g)\}$.[8]

The proof is reported in Appendix A.2. Corollary 12 states that, instead of computing $f_p^\wedge \otimes g_p^\wedge$ using domains $I_{f_p^\wedge}$ and $I_{g_p^\wedge}$ defined in (13), we can use *both* $I'_{f_p^\wedge}$ and $I'_{g_p^\wedge}$, whose size depends on k_{c_f}, k_{c_g} instead of k_{d_f}, k_{d_g}. Finally, we show that we can *mix and match* the above intervals, to minimize the extension of *each* operand, independently.

▶ **Theorem 13** (Mix and Match ((min,+) Convolution))**.** *Let f and $g \in \mathcal{U}$ which are neither UC nor UI, and are left-continuous and non-decreasing in $[T_f, +\infty[$ and $[T_g, +\infty[$, respectively. Let $I_{f_p^\wedge}, I_{g_p^\wedge}$ be the intervals to compute $f_p^\wedge \otimes g_p^\wedge$ according to (13), and let $I'_{f_p^\wedge}, I'_{g_p^\wedge}$ be the intervals to compute the same through Corollary 12. Then $I_{f_p^\wedge} \cap I'_{f_p^\wedge}, I_{g_p^\wedge} \cap I'_{g_p^\wedge}$ are valid intervals to compute $f_p^\wedge \otimes g_p^\wedge$.*

[8] The suprema are attainable since the functions are left-continuous over the respective intervals.



(a) S_f.

(b) $S_{f\uparrow}^{-1}$.

Figure 4 Example of upper pseudoinverse of a sequence S_f.

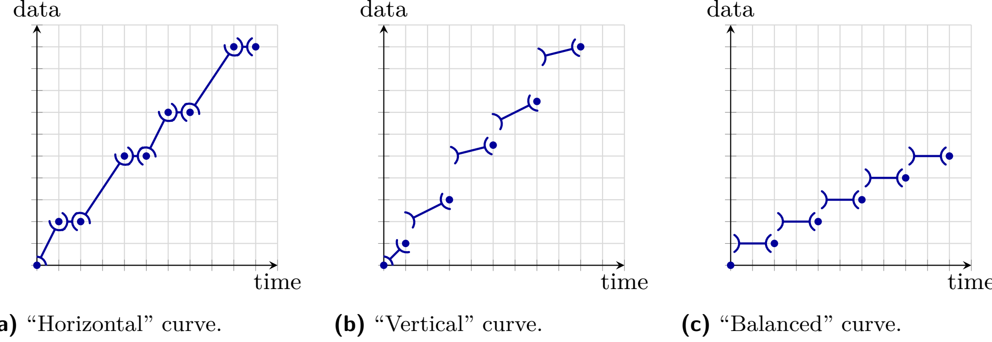
The proof is reported in Appendix A.2. This theorem allows us to compute h_{pp} using extended sequences that have the minimum cardinality between those that the *direct* and the *inverse* methods would compute. We exemplify this through a simple example.

Consider f and g with $d_f = 2$, $c_f = 13$, $d_g = 11$ and $c_g = 3$. Then, using the *direct* method we will compute $2 \cdot k_{d_f} = 22$ period extensions of f and $2 \cdot k_{d_g} = 4$ period extensions of g. On the other hand, using the *inverse* method, we will compute the same result with $2 \cdot k_{c_f} = 6$ period extensions of f and $2 \cdot k_{c_g} = 26$ period extensions of g. Which of the two will be faster depends both on the number of elements contained in each period of f and g, but also on the topological properties of these elements, which are difficult to understand *ex ante* [29]. Using the above theorem, though, we can just take the best option for each independently: $2 \cdot k_{c_f} = 6$ period extensions of f and $2 \cdot k_{d_g} = 4$ period extensions of g – which is clearly better than both the previous options.

Based on the above, we can define a new algorithm, called *isospeed*, which outperforms both the *direct* and the *inverse* ones. It consists in modifying in Algorithm 1 including the new, optimized values for the parameters, i.e., (19), (20) and (21) at line 4, and Theorem 13 at line 5. Moreover, we also optimize line 6, i.e., the *by-sequence* convolution, still leveraging isomorphism. We start from the algorithm in [6], summarized below.

Given two sequences S_a and S_b, consider their elements $e^a_1, \dots, e^a_n$ and $e^b_1, \dots, e^b_m$, where $n = n(S_a)$, $m = n(S_b)$. For each pair e^a_i, e^b_j, we can then compute the *elementary* (min,+) convolution $e^a_i \otimes e^b_j$, i.e., $n(S_a) \cdot n(S_b)$ elementary convolutions. Then, $S_a \otimes S_b$ is computed as the lower envelope of these elementary convolutions. However, it is easy to see that not all pairs e^a_i, e^b_j will contribute to the end result. Indeed, the convolution result is relevant only for a given interval (e.g., in Algorithm 1 $I_{h_{pp}} = [T_f + T_g, T_f + T_g + 2 \cdot \mathrm{lcm}(d_f, d_g)[$), thus any elementary convolution whose abscissas fall outside such interval can be safely skipped. We call this *horizontal filtering*. Similarly, when applying the optimizations described in the previous section, we can ignore elementary convolutions whose *ordinates* fall outside $[f(T_f) + g(T_g), f(T_f) + g(T_g) + 2 \cdot \mathrm{lcm}(c_f, c_g)]$ (*vertical filtering*). Horizontal and vertical filtering further reduce the computation time.

Moreover, we recall that – under pseudoinversion – constant segments become discontinuities and vice versa, leading to different cardinalities for the sequences of an operand. This is exemplified by Figure 4, where S_f has six elements, while $S_{f\uparrow}^{-1}$ has four. While constant segments do contribute to the complexity of computing the convolution, discontinuities do not, being only a difference in value between two elements. Thus, also within the by-sequence convolution there may be a runtime difference between the direct and inverse approach.



(a) “Horizontal” curve. **(b)** “Vertical” curve. **(c)** “Balanced” curve.

Figure 5 Shapes of curves used in our experiments.

Accordingly, we optimize the by-sequence convolution via the following heuristic: we count the total number of constant segments and discontinuities of the two operands, call them C and D, respectively. Then, if $D > C$, we perform the by-sequence convolution of the operands as they are. If, instead $D < C$, we pseudo-invert the sequences first, perform a (max,+) by-sequence convolution, and then pseudo-invert the result again. This heuristic is cheap, being $\mathcal{O}\left(n(S_f) + n(S_g)\right)$. However, it may not have 100% accuracy. In fact, work [29] discusses that the topological properties of the elements, which are difficult to understand *ex ante*, may also influence the runtime of by-sequence convolution.

Finally, we discuss the algorithmic cost of the isospeed algorithm. Applying the mix-and-match theorem requires computing the extension multipliers of both operands and comparing constants, which is $\mathcal{O}\left(1\right)$. However, testing the hypotheses that each operand must be left-continuous and non-decreasing is – strictly speaking – $\mathcal{O}\left(n(S)\right)$, where S is the base sequence, not the extended one. Note that the same cost has to be paid in the *inverse* algorithm as well. However, we observe that such a cost can easily be amortized by testing these properties *once* per operand and caching the result. Moreover, computing C and D for the heuristic also has a linear cost.

4 Performance Evaluation

The isospeed algorithm has been implemented by extending Nancy [28], an open-source library implementing the algorithms from [6, 29, 27].We compared it against two baselines, i.e., the *direct* algorithm, [6], recalled in Algorithm 1, and the *inverse* one [20]. To highlight the impact of the by-sequence convolution and its heuristic, we run the experiments using three different shapes of operands, shown in Figure 5. *Horizontal* curves have constant segments but no discontinuities, whereas *vertical* curves have discontinuities but no constant segments. *Balanced* curves are similar to the type of curves studied in [20], and have an equal number of constant segments and discontinuities.

We run the experiments on a cloud Virtual Machine (Intel Xeon Processors (CascadeLake) cores @2.2 GHz, 32 GB of DRAM, Ubuntu 22.04), using randomly generated parameters for the shapes discussed above. We run all algorithms in serial mode (rather than *parallel*, which is the default in Nancy). To make the comparison more challenging, *horizontal* filtering is included in the baseline algorithms as well, since it does not depend on the results of this paper, whereas vertical filtering – which is a consequence of isomorphism – is used only in the isospeed algorithm. Moreover, we include the cost of testing operand properties in the *isospeed* and *inverse* algorithms (there is nothing to test for the *direct* one). We measured the time to compute the convolution using the three methods.

Our results are shown in Figures 6–8. The results of Figures 6a, 7a, and 8a and Figures 6b, 7b, and 8b clearly show that the *isospeed* algorithm outperforms blindly choosing either the *direct* or the *inverse* approach, reducing runtimes often by several orders of magnitude. Moreover, Figures 6c, 7c, and 8c show that *isospeed* performs at least as well as the best between *direct* and *inverse* in most cases, and sometimes even better, being up to one order of magnitude faster. The improvements occur whenever both baseline algorithms extend one operand more than necessary, whereas the isospeed does not, due of its mix-and-match approach. At the risk of stating the obvious, we remark that you do not know which of the two baseline algorithms is the best *beforehand*.

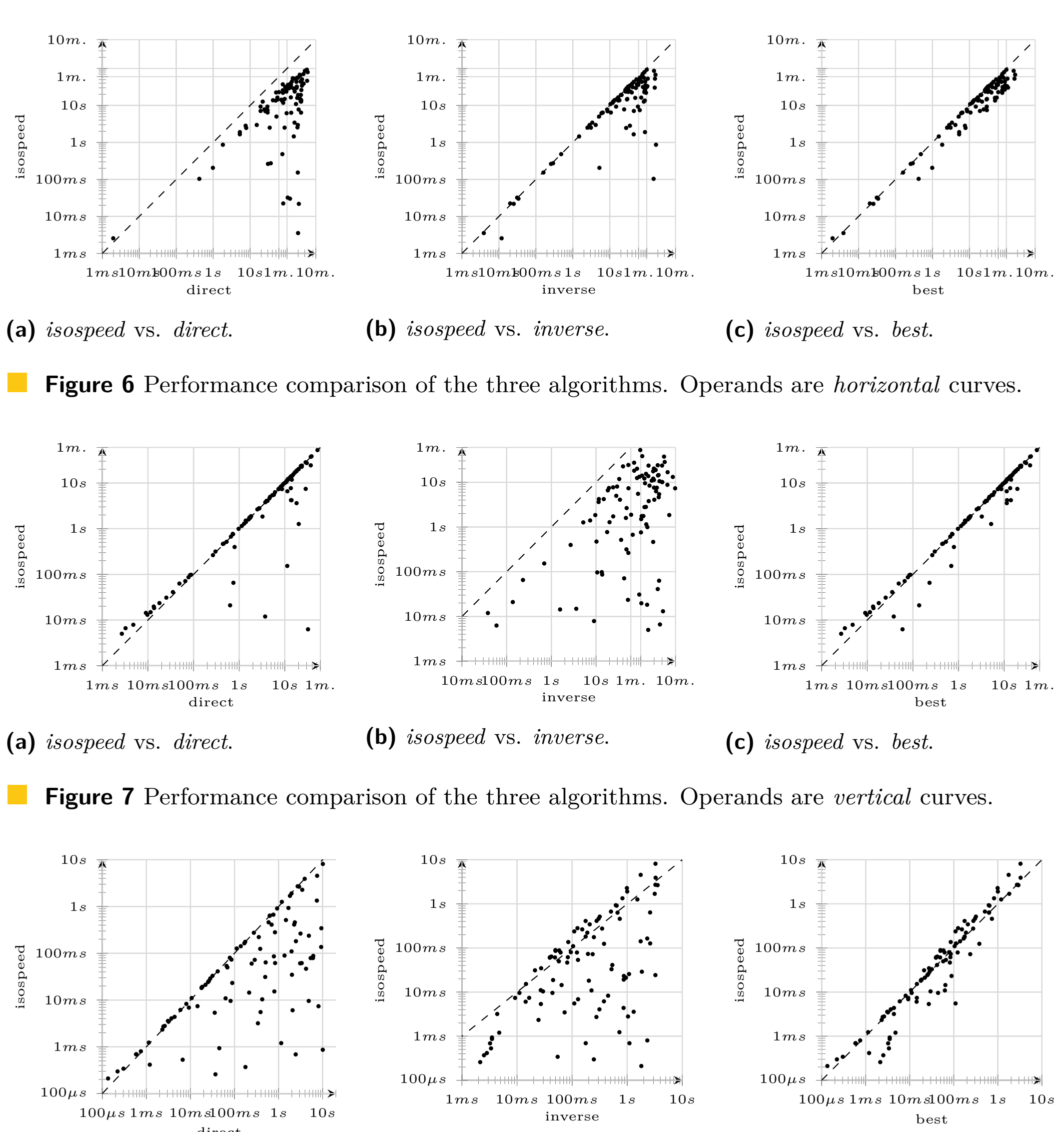
There are indeed some cases when isospeed adds a modicum of overhead – see the few points above the bisector in the figures. This is due to two different reasons: the points in the bottom-left corner of, e.g., Figures 7a and 7c are experiments where runtimes are in the order of milliseconds, and the overhead of testing hypotheses is significant against such a short timespan. We do not see this as a relevant shortcoming – there is little to optimize if the baseline is already that fast. The points above the bisector in the top-right region of Figures 8b and 8c are instead experiments when our heuristic fails to select the most efficient way to perform the by-sequence convolution. This only occurs with *balanced* curves, and for a reason: with *horizontal* and *vertical* curves, the choice is quite clear-cut – it is either $D \gg C$ or $C \ll D$, respectively, and the heuristic always selects the best approach. *Balanced* curves, instead, are designed to thwart our heuristic – they have, in fact, $D \approx C$.

All the above experiments highlighted speedups of up to one order of magnitude against the (clairvoyant) best baseline. Such speedups depend on numerical properties of the operands, and random generation of operand parameters seldom hits on the most interesting cases. As a last set of experiments, we generate horizontal operands in such a way that $k_{d_f} > k_{c_f}$ and $k_{d_g} < k_{c_g}$ (or vice versa), as in the example reported at the end of Section 3.2, so that each baseline algorithm will always extend one operand more than necessary. The results of these experiments are reported in Figure 9, and they show more frequent and significant speedups against the best baseline (e.g., compared with Figure 6) – reaching two orders of magnitude.

As a last remark, we observe that the absolute magnitude of the runtimes involved in these experiments is a few tens of seconds at most. This was done on purpose to keep experiments manageable, and it certainly does not imply that (min,+) convolutions are always that fast. One can always devise cases where runtimes are in the order of hours or more – all it takes is period lengths and heights that are products of large primes. Moreover, it is well known that chaining convolutions leads to exponentially increasing runtimes, much like chaining lcms does, a phenomenon called *state explosion* [12, 13, 29]. In these cases, *isospeed* may act as an enabler of otherwise unfeasible performance studies.

5 Conclusions

In this paper, we have investigated what is perhaps the most common operation in (min,+) algebra, i.e., (min,+) convolution. Starting from the observation that – due to (min,+)/(max,+) isomorphism – there are always two ways to compute a (min,+) convolution, the *direct* and *inverse* algorithm, we provide a technically sound explanation of an observation first appeared in [20], i.e., that one algorithm can be considerably faster than the other. The reason lies in the way the two algorithm extend operands, which is the key factor in the complexity of said algorithms. Based on the above observation, we prove algebraic properties that allow one to minimize operand extension, for each operand independently. This allows



(a) *isospeed* vs. *direct.* **(b)** *isospeed* vs. *inverse.* **(c)** *isospeed* vs. *best.*

Figure 6 Performance comparison of the three algorithms. Operands are *horizontal* curves.

(a) *isospeed* vs. *direct.* **(b)** *isospeed* vs. *inverse.* **(c)** *isospeed* vs. *best.*

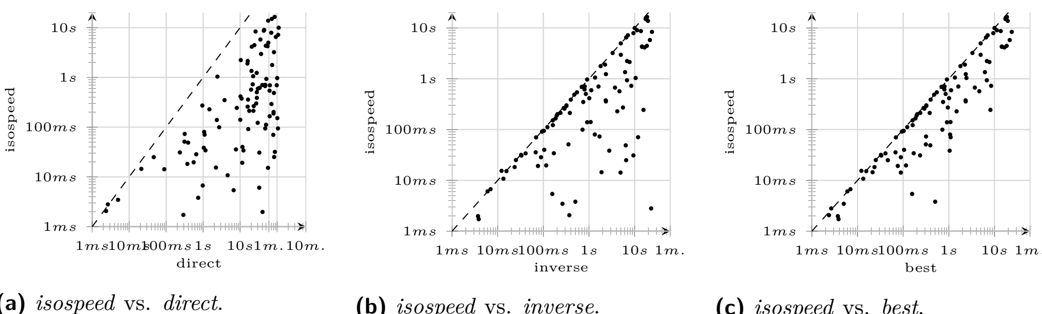
Figure 7 Performance comparison of the three algorithms. Operands are *vertical* curves.

(a) *isospeed* vs. *direct.* **(b)** *isospeed* vs. *inverse.* **(c)** *isospeed* vs. *best.*

Figure 8 Performance comparison of the three algorithms. Operands are *balanced* curves.

us to devise a novel algorithm, called *isospeed*, that outperforms both the *direct* and *inverse* algorithms, reducing runtimes often by several orders of magnitude. More interestingly, our isospeed algorithm also beats a clairvoyant heuristic that guesses the best of the above two baselines: except in few cases when it adds a modicum of overhead, isospeed is at least as fast as that, and can be one or two orders of magnitude faster.

Abating the cost of (min,+) convolution by orders of magnitude is not just a performance improvement: it may also enable performance studies that were previously considered to be beyond the realm of doable, e.g., because of state explosion. Some examples of this problem are reported in [29]. Our findings allow us to reduce this problem as much as possible, in the most general settings: unlike the techniques described in [12, 13, 14, 29], which only apply to specific classes of operands, isospeed only requires operands to be left-continuous and non-decreasing.



(a) *isospeed* vs. *direct*. **(b)** *isospeed* vs. *inverse*. **(c)** *isospeed* vs. *best*.

Figure 9 Performance comparison of the three algorithms. Operands are *horizontal* curves, and their parameters are set so that comparing extension multipliers is inconclusive.

Due to space limitations, we were unable to discuss (max,+) convolution in this paper. However, all the results in this paper apply – via minor, straightforward changes – to that as well, because isomorphism works both ways. More to the point, the performance improvements for (max,+) convolution are exactly the same, since – as we discussed briefly in Section 2 – the algorithm for the (max,+) convolution is not different from that of (min,+) convolution. This increases the significance of our findings.

As a future work, we plan to devise more precise heuristics, that allow one to identify the most efficient by-sequence convolution more effectively. Moreover, we believe that the same process highlighted in this paper could be used to find alternative, improved algorithms for other operations, e.g., the $(\cdot, +)$ *deconvolution*.

Finally, many works in real-time literature deal with *supply functions* and *demand bound functions*, which are similar to the curves discussed here [23, 11, 18, 22, 1]. A further avenue of research is then to explore the possibility to express these results using Real-Time Calculus, hence making the computational improvements discussed in this paper available to speed up the resolution of these problems.

References

1 Luis Almeida and Paulo Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *Proceedings of the 4th ACM international conference on Embedded software*, pages 95–103, 2004.

2 François Baccelli, Guy Cohen, Geert Jan Olsder, and Jean-Pierre Quadrat. *Synchronization and linearity: an algebra for discrete event systems.* John Wiley & Sons Ltd, 1992.

3 Mahmoud Bazzal, Lukas Krawczyk, and Carsten Wolff. RTCAnalysis: Practical Modular Performance Analysis of Automotive Systems with RTC. In Audrius Lopata, Daina Gudonienė, and Rita Butkienė, editors, *Information and Software Technologies*, pages 209–223, Cham, 2021. Springer International Publishing.

4 Anne Bouillard, Marc Boyer, and Euriell Le Corronc. *Deterministic Network Calculus: From Theory to Practical Implementation.* Wiley, Hoboken, NJ, 2018.

5 Anne Bouillard, Bertrand Cottenceau, Bruno Gaujal, Laurent Hardouin, Sébastien Lagrange, Mehdi Lhommeau, and Éric Thierry. COINC library: a toolbox for the Network Calculus. In *VALUETOOLS'09*, 2009.

6 Anne Bouillard and Éric Thierry. An algorithmic toolbox for network calculus. *Discrete Event Dynamic Systems*, 18(1):3–49, 2008.

7 Marc Boyer, Amaury Graillat, Benoît Dupont de Dinechin, and Jörn Migge. Bounding the delays of the MPPA network-on-chip with network calculus: Models and benchmarks. *Performance Evaluation*, 143:102124, 2020. doi:10.1016/j.peva.2020.102124.

8 Cheng-Shang Chang. *Performance guarantees in communication networks.* Springer-Verlang, New York, USA, 2000.

9 Rene L Cruz. A calculus for network delay, part II: Network analysis. *IEEE Transactions on information theory*, 37(1):132–141, 1991.

10 Rene L Cruz et al. A calculus for network delay, part I: Network elements in isolation. *IEEE Transactions on information theory*, 37(1):114–131, 1991.

11 Xiang Feng and Aloysius K Mok. A model of hierarchical real-time virtual resources. In *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.*, pages 26–35. IEEE, 2002.

12 Nan Guan and Wang Yi. Finitary real-time calculus: Efficient performance analysis of distributed embedded systems. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 330–339, 2013.

13 Kai Lampka, Steffen Bondorf, and Jens Schmitt. Achieving efficiency without sacrificing model accuracy: Network calculus on compact domains. In *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 313–318. IEEE, 2016.

14 Kai Lampka, Steffen Bondorf, Jens B. Schmitt, Nan Guan, and Wang Yi. Generalized finitary Real-Time calculus. In *Proc. of the 36th IEEE International Conference on Computer Communications (INFOCOM 2017)*, 2017.

15 Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet.* Springer Science & Business Media, Berlin, Germany, 2001.

16 Euriell Le Corronc, Bertrand Cottenceau, and Laurent Hardouin. Container of (min,+)-linear systems. *Discrete Event Dynamic Systems*, 24(1):15–52, 2014.

17 Jörg Liebeherr. Duality of the Max-Plus and Min-Plus Network Calculus. *Foundations and Trends in Networking*, 11(3-4):139–282, 2017. `doi:10.1561/1300000059`.

18 Giuseppe Lipari and Enrico Bini. Resource partitioning among real-time applications. In *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings.*, pages 151–158. IEEE, 2003.

19 David A. Nascimento, Steffen Bondorf, and Divanilson R. Campelo. Modeling and Analysis of Time-Aware Shaper on Half-Duplex Ethernet PLCA Multidrop. *IEEE Transactions on Communications*, 71(4):2216–2229, 2023. `doi:10.1109/TCOMM.2023.3246080`.

20 Victor Pollex, Henrik Lipskoch, Frank Slomka, and Steffen Kollmann. Runtime Improved Computation of Path Latencies with the Real-Time Calculus. In *Proceedings of the 1st International Workshop on Worst-Case Traversal Time*, pages 58–65, 2011.

21 RealTime-at-Work. RTaW-Pegase (min,+) library. `https://www.realtimeatwork.com/rtaw-pegase-libraries/`. Accessed: 2022-04-05.

22 Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*, pages 2–13. IEEE, 2003.

23 Martin Stigge, Pontus Ekberg, Nan Guan, and Wang Yi. The digraph real-time task model. In *2011 17th IEEE real-time and embedded technology and applications symposium*, pages 71–80. IEEE, 2011.

24 Deepak Vedha Raj Sudhakar, Karsten Albers, and Frank Slomka. Generalized and scalable offset-based response time analysis of fixed priority systems. *Journal of Systems Architecture*, 112:101856, 2021. `doi:10.1016/j.sysarc.2020.101856`.

25 L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *2000 IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 101–104 vol.4, 2000. `doi:10.1109/ISCAS.2000.858698`.

26 Ernesto Wandeler and Lothar Thiele. Real-Time Calculus (RTC) Toolbox. `http://www.mpa.ethz.ch/Rtctoolbox`. URL: `http://www.mpa.ethz.ch/Rtctoolbox`.

27 Raffaele Zippo, Paul Nikolaus, and Giovanni Stea. Extending the Network Calculus Algorithmic Toolbox for Ultimately Pseudo-Periodic Functions: Pseudo-Inverse and Composition, 2022. `doi:10.48550/arXiv.2205.12139`.

28 Raffaele Zippo and Giovanni Stea. Nancy: An efficient parallel Network Calculus library. *SoftwareX*, 19:101178, 2022. `doi:10.1016/j.softx.2022.101178`.

29 Raffaele Zippo and Giovanni Stea. Computationally efficient worst-case analysis of flow-controlled networks with Network Calculus. *IEEE Transactions on Information Theory*, 2023. `doi:10.1109/TIT.2023.3244276`.

A Appendix

This appendix reports the proofs of the main results of this paper, i.e., Theorem 11, Corollary 12 and Theorem 13. In order to make these proofs rigorous, we need a few preliminary technical clarifications. These preliminaries state that the (min,+)/(max,+) isomorphism stated in [20, 17] holds for the convolution of periodic parts as well. This is fairly intuitive, but requires to be rigorously stated nonetheless. Said preliminaries re-state existing results from [6, 4, 17, 27], tweaking the existing proofs to accommodate restricted functions in $\mathcal{U}$. The preliminary results are reported in the first part of this appendix, and the proof of our main results follow in the second part.

A.1 Preliminary results

In the main results of this work, we exploit properties analogue to the isomorphisms proved in [17], such as Theorem 8, which need to be applied to functions restricted over a support, e.g., $f_p^{\wedge}$. Now, pseudoinverses – and, consequently, isomorphism – are defined for *non-decreasing* functions, and restricted functions are not. Thus, in this section we generalize pseudoinversion and isomorphism to include restricted functions.

▶ **Definition 14** (Lower and Upper Pseudoinverse over an Interval). *Let $f \in \mathcal{U}$ be non-decreasing over I, where $I = [a, +\infty[\subset \mathbb{Q}$. Then, its* lower pseudoinverse over (the interval) *I is defined as*

$$f^{-1}_{\downarrow,I}(y) := \begin{cases} \inf \{t \in I \mid f(t) \geq y\}, & \text{if } y \geq f(a), \\ +\infty, & \text{otherwise,} \end{cases} \tag{23}$$

and its upper pseudoinverse over (the interval) *I is defined as*

$$f^{-1}_{\uparrow,I}(y) := \begin{cases} \sup \{t \in I \mid f(t) \leq y\}, & \text{if } y \geq f(a), \\ -\infty, & \text{otherwise.} \end{cases} \tag{24}$$

Note that it does not hold in general that $\sup \{t \in I \mid f(t) < y\} = \inf \{t \in I \mid f(t) \geq y\}$, for the lower pseudoinverse, as well as $\sup \{t \in I \mid f(t) \leq y\} = \inf \{t \in I \mid f(t) > y\}$, for the upper pseudoinverse. However, if $y > f(a)$ (for $I = [a, +\infty[$), the two equations hold. We also note that, since these pseudoinverses consider only values of $f(t)$ for $t \in I$, it follows that $f^{-1}_{\downarrow,I} = \left(f|_I^{\wedge}\right)^{-1}_{\downarrow,I} = \left(f|_I^{\vee}\right)^{-1}_{\downarrow,I}$, and similarly for $f^{-1}_{\uparrow,I}$. Finally, given $f(I) := [f(a), +\infty[$, the lower pseudoinverse over I is left-continuous over $f(I)$, and the upper pseudoinverse over I is right-continuous over $f(I)$.

▶ **Theorem 15.** *Let I be an interval of the form $[a, +\infty[$. Let $f \in \mathcal{U}$ be neither UC nor UI, and is non-decreasing over I. Then, its lower pseudoinverse over I, $f^{-1}_{\downarrow,I}$, is again a function of $\mathcal{U}$ with*

$$T_{f^{-1}_{\downarrow,I}} = \begin{cases} f\left(T_f + d_f\right), & \text{if } a \leq T_f, \\ f\left(a + d_f\right), & \text{if } a > T_f, \end{cases} \tag{25}$$

$$d_{f^{-1}_{\downarrow,I}} = c_f, \tag{26}$$

$$c_{f^{-1}_{\downarrow,I}} = d_f, \tag{27}$$

and its upper pseudoinverse over I, $f_{\uparrow,I}^{-1}$, is again a function of $\mathcal{U}$ with

$$T_{f_{\uparrow,I}^{-1}} = \begin{cases} f\left(T_f\right), & \text{if } a \leq T_f, \\ f\left(a\right), & \text{if } a > T_f, \end{cases} \tag{28}$$

$$d_{f_{\uparrow,I}^{-1}} = c_f, \tag{29}$$

$$c_{f_{\uparrow,I}^{-1}} = d_f. \tag{30}$$

The proofs are easily derived following the steps of those in Theorem 9 and Theorem 10 in [27]. The only difference is that, since we are considering values of $f(t)$ only for $t \in [a, +\infty[$, we can only consider $f(t)$ to be UPP from $\max\left(T_f, a\right)$. Note that, in the rest of this paper, we will always use $a \leq T_f$, thus only the first branch of (25) and (28) apply. As briefly mentioned in [27], one can improve the result of (25) using additional assumptions on the shape of f. As this will be useful in this work, we derive these results explicitly.

▶ **Lemma 16.** *Let $f \in \mathcal{U}$ be neither UC nor UI, right-continuous, and non-decreasing over the interval $I = [a, +\infty[$, where $a \leq T_f$. Let*

$$T_f^* := f_{\downarrow,I}^{-1}(f(T_f)) = \inf\left\{t \geq a \mid f(t) = f(T_f)\right\}, \tag{31}$$

$$T_f^{**} := f_{\downarrow,I}^{-1}(f(T_1^* + d_f)) = \inf\left\{t \geq a \mid f(t) = f(T_1^* + d_f)\right\}. \tag{32}$$

Then, if f is UPP from T_f^* and if $T_f^{**} = T_f^* + d_f$, the pseudo-periodic start $T_{f_{\uparrow}^{-1}}$ in (25) can be improved into

$$T_{f_{\downarrow,I}^{-1}} = f(T_f). \tag{33}$$

The properties addressed by Lemma 16 have to do with constant segments at the start and the end of the pseudo-period which, as discussed in [27], can lead to $T_{f_{\downarrow,I}^{-1}} > f(T_f)$. The conditions of Lemma 16 ensure that this does not happen, by checking that either a) there are no constant segments before T_f and $T_f + d$, or b) these constant segments are of equal length, such that $f_{\downarrow,I}^{-1}$ is UPP from $f(T_f)$. The proof consists, in fact, in verifying that due to right-continuity of f and the definitions of T_f^*, T_f^{**}, it follows that a) or b) are verified and thus $T_{f_{\downarrow,I}^{-1}} = f(T_f)$.

▶ **Lemma 17.** *Let $f \in \mathcal{U}$ be neither UC nor UI, left-continuous, and non-decreasing over the interval $I = [a, +\infty[$, where $a \leq T_f$. Moreover, let $f_{\uparrow,I}^{-1}$ be its upper pseudoinverse over I. Then, $f_{\uparrow,I}^{-1}$ satisfies the conditions of Lemma 16, thus its lower pseudoinverse $\left(f_{\uparrow,I}^{-1}\right)_{\downarrow,[f(a),+\infty[}^{-1}$ is UPP from $f_{\uparrow,I}^{-1}(T_{f_{\uparrow,I}^{-1}}) = T_f$.*

The proof is easily derived by observing that, in order for $f_{\uparrow,I}^{-1}$ to violate the conditions of Lemma 16, f would need to have left-discontinuities, which is a contradiction.

▶ **Lemma 18** (Sufficient cut for Upper Pseudoinverse over Interval)**.** *Let $f \in \mathcal{U}$ be neither UC nor UI, and is left-continuous and non-decreasing over $I = [a, +\infty[$. Then, in order to compute $f_{\uparrow,I}^{-1}(x)$ and $x \in [x_1, x_2] \subset [f(a), +\infty[$ with $x_1 < x_2$, it is sufficient to use $f(t)$ with $t \in [t_1, t_2]$, where $t_1 := f_{\uparrow,I}^{-1}(x_1)$ and $t_2 := f_{\uparrow,I}^{-1}(x_2)$.*

The proof is easily derived by observing that, for any $x \in [x_1, x_2]$, $\sup\left\{t \geq a \mid f(t) \leq x\right\} = \sup\left\{t_1 \leq t \leq t_2 \mid f(t) \leq x\right\}$.

▶ **Lemma 19** (Sufficient cut for Lower Pseudoinverse over Interval). *Let $f \in \mathcal{U}$ be neither UC nor UI, and is right-continuous and non-decreasing over $I = [a, +\infty[$. Then, in order to compute $f^{-1}_{\downarrow,I}(x)$ with $x \in [x_1, x_2] \subset [f(a), +\infty[$ and $x_1 < x_2$, it is sufficient to use $f(t)$ with $t \in [t_1, t_2]$, where $t_1 \coloneqq f^{-1}_{\downarrow,I}(x_1)$ and $t_2 \coloneqq f^{-1}_{\downarrow,I}(x_2)$.*

The proof is similar to the one for Lemma 18.

▶ **Lemma 20.** *Let $f \in \mathcal{U}$ be non-decreasing and $I = [a, +\infty[\subset \mathbb{Q}_+$. Let $x \in I$. If $f(x) \leq y$, then $f^{-1}_{\uparrow,I}(y) \geq x$.*

The proof follows along the lines of [17, pp. 62], since it follows from $x \in I$ that we are in the supremum case of (24). Next, we generalize Proposition 3.10 in [4, pp. 48].

▶ **Proposition 21.** *Let $f, g \in \mathcal{U}$ be left-continuous and non-decreasing. Then, for any $t \in [T_f + T_g, +\infty[$ it exists $s^* \in [T_f, t - T_g]$ such that*

$$(f^{\wedge}_p \otimes g^{\wedge}_p)(t) = \inf_{T_f \leq s \leq t - T_g} \left\{ f^{\wedge}_p(s) + g^{\wedge}_p(t - s) \right\} = f(s^*) + g(t - s^*).$$

In other words, the infimum is attainable.

The proof follows along the lines of [4, pp. 48]. The only difference is related to the use of Bolzano-Weierstrass theorem for real sequences, obtaining $s^* \in \mathbb{R}_+$. However, it is easy to show that, since functions of $\mathcal{U}$ are piecewise affine $\mathbb{Q}_+ \to \mathbb{Q} \cup \{+\infty, -\infty\}$, given any convergent sequence $f(s_n)$, $s_n \in \mathbb{Q}_+$ and s^* attains the limit of this sequence, then $s^* \in \mathbb{Q}_+$.

▶ **Proposition 22.** *Let f and g be non-decreasing and right-continuous functions of $\mathcal{U}$. Then, for any $t \in [T_f + T_g, +\infty[$ it exists $s^* \in [T_f, t - T_g]$ such that*

$$(f^{\vee}_p \overline{\otimes} g^{\vee}_p)(t) = \sup_{T_f \leq s \leq t - T_g} \left\{ f^{\vee}_p(s) + g^{\vee}_p(t - s) \right\} = f(s^*) + g(t - s^*)$$

In other words, the supremum is attainable.

The proof follows along the same lines as Proposition 21. Next, we provide a generalization of Theorem 8 for functions restricted to their periodic part.

▶ **Theorem 23.** *Let $f, g \in \mathcal{U}$ be left-continuous and non-decreasing. Then*

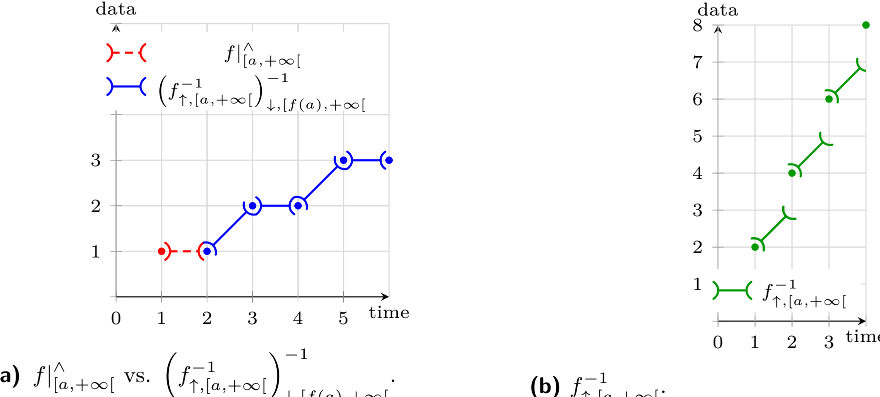
$$\left(f^{\wedge}_p \otimes g^{\wedge}_p\right)^{-1}_{\uparrow,[T_f+T_g,+\infty[} = \left(f^{-1}_{\uparrow,p} \overline{\otimes} g^{-1}_{\uparrow,p}\right). \tag{34}$$

The proof follows along the same lines of [17, Theorem 10.3, p. 69]. The only difference is that, for $x < f(T_f) + g(T_g)$, we derive that both sides are $-\infty$. Next, we also generalize the involutive property of pseudoinverses.

▶ **Proposition 24.** *Let $f \in \mathcal{U}$ be left-continuous and non-decreasing on the interval $I = [a, +\infty[$. Let $a' \coloneqq \sup \{t \geq a \mid f(t) = f(a)\} \geq a$. Then*

$$\left(f^{-1}_{\uparrow,[a,+\infty[}\right)^{-1}_{\downarrow,[f(a),+\infty[} = f|^{\wedge}_{[a',+\infty[}.$$

The proof follows along the same lines of [17, Lemma 10.1 (c), pp. 64-6]. The only difference is that, for $t < a'$, we derive that for both sides are $+\infty$. Note that Proposition 24 implies that performing the lower pseudoinverse (over an interval) of an upper pseudoinverse (over an interval) does not reconstitute f over that same interval, but only a subset of it: in fact, we obtain $f|^{\wedge}_{[a',+\infty[}$ instead of $f|^{\wedge}_{[a,+\infty[}$, where $a' \geq a$. This information loss



(a) $f|^{\wedge}_{[a,+\infty[}$ vs. $\left(f^{-1}_{\uparrow,[a,+\infty[}\right)^{-1}_{\downarrow,[f(a),+\infty[}$. **(b)** $f^{-1}_{\uparrow,[a,+\infty[}$.

Figure 10 Example of loss of information when computing pseudoinverses over an interval.

is exemplified in Figure 10, where we show that the values of f between 1 and 2 are lost. However, if the value a is known, it is easy to reconstitute $f|^{\wedge}_{[a,+\infty[}$ by observing that the missing values of f in $[a, a'[$ are all $f(a')$, which is part of the result. Again referencing Figure 10, we can see how the constant segment in red is the missing piece that can be reconstituted by knowing $a = 1$. We formalize this process through the *reconstruction operator*, $[f]_a$. Given a function f that is either $+\infty$ or $-\infty$ in $[0, a'[$, and finite in $[a', +\infty[$, then

$$[f]_a(t) \coloneqq \begin{cases} f(t), & \text{if } t \in [0, a[, \\ f(a'), & \text{if } t \in [a, a'[, \\ f(t), & \text{if } t \in [a', +\infty[. \end{cases} \tag{35}$$

Using the reconstruction operator, we can state a stronger version of Proposition 24.

▶ **Proposition 25.** *Let $f \in \mathcal{U}$ be left-continuous and non-decreasing on the interval $I = [a, +\infty[$. Let $a' \coloneqq \sup\{t \geq a \mid f(t) = f(a)\} \geq a$. Then*

$$\left[\left(f^{-1}_{\uparrow,[a,+\infty[}\right)^{-1}_{\downarrow,[f(a),+\infty[}\right]_a = f|^{\wedge}_{[a,+\infty[}. \tag{36}$$

The proof is easily derived by applying (35). Combining these results, we can derive an alternative expression for $f^{\wedge}_p \otimes g^{\wedge}_p$, which is the analogous to (7) for restricted functions.

$$f^{\wedge}_p \otimes g^{\wedge}_p = \left[\left(f^{-1}_{\uparrow,p} \,\overline{\otimes}\, g^{-1}_{\uparrow,p}\right)^{-1}_{\downarrow,[f(T_f)+g(T_g),+\infty[}\right]_{T_f+T_g}. \tag{37}$$

A.2 Proofs of the results in Section 3.2

▶ **Theorem 11.** *Let $f, g \in \mathcal{U}$ be left-continuous and non-decreasing functions. Then, $h_{pp} = f^{\wedge}_p \otimes g^{\wedge}_p$ is again a function of $\mathcal{U}$ with*

$$d_{h_{pp}} = \max\left(\frac{d_f}{c_f}, \frac{d_g}{c_g}\right) \cdot \operatorname{lcm}(c_f, c_g) = \max\left(k_{c_g} \cdot d_g, k_{c_f} \cdot d_f\right), \tag{16}$$

$$c_{h_{pp}} = \operatorname{lcm}(c_f, c_g), \tag{17}$$

$$T_{h_{pp}} = \sup\left\{t \geq T_f + T_g \mid f^{\wedge}_p \otimes g^{\wedge}_p(t) \leq f(T_f) + g(T_g) + \operatorname{lcm}(c_f, c_g)\right\}. \tag{18}$$

Proof. Using Proposition 25, we have that

$$f_p^\wedge \otimes g_p^\wedge \overset{(36)}{=} \left[\left(\left(f_p^\wedge \otimes g_p^\wedge\right)^{-1}_{\uparrow,[T_f+T_g,+\infty[}\right)^{-1}_{\downarrow,[f(T_f)+g(T_g),+\infty[}\right]_{(T_f+T_g)}$$
$$\overset{(34)}{=} \left[\left(f^{-1}_{\uparrow,p} \,\overline{\otimes}\, g^{-1}_{\uparrow,p}\right)^{-1}_{\downarrow,[f(T_f)+g(T_g),+\infty[}\right]_{(T_f+T_g)},$$

where we used Theorem 23 in the second line. For the inner part (the (max,+) convolution), we obtain for all $x \geq f(T_f) + g(T_g) + c_{h_{pp}} = f(T_f) + g(T_g) + \mathrm{lcm}(c_f, c_g)$ that

$$\left(f^{-1}_{\uparrow,p} \,\overline{\otimes}\, g^{-1}_{\uparrow,p}\right)(x + c_{h_{pp}})$$
$$= \sup_{f(T_f) \leq u \leq x + c_{h_{pp}} - g(T_g)} \left\{ f^{-1}_{\uparrow,p}(u) + g^{-1}_{\uparrow,p}(x + c_{h_{pp}} - u) \right\}$$
$$\overset{(x - g(T_g) \geq f(T_f) + c_{h_{pp}})}{=} \sup_{f(T_f) \leq u \leq x - g(T_g)} \left\{ f^{-1}_{\uparrow,p}(u) + g^{-1}_{\uparrow,p}(x + c_{h_{pp}} - u) \right\}$$
$$\vee \sup_{f(T_f) + c_{h_{pp}} \leq u \leq x + c_{h_{pp}} - g(T_g)} \left\{ f^{-1}_{\uparrow,p}(u) + g^{-1}_{\uparrow,p}(x + c_{h_{pp}} - u) \right\},$$

We continue by substituting $v \coloneqq x + c_{h_{pp}} - u$:

$$\left(f^{-1}_{\uparrow,p} \,\overline{\otimes}\, g^{-1}_{\uparrow,p}\right)(x + c_{h_{pp}})$$
$$\overset{(v := x + c_{h_{pp}} - u)}{=} \sup_{f(T_f) \leq u \leq x - g(T_g)} \left\{ f^{-1}_{\uparrow,p}(u) + g^{-1}_{\uparrow,p}(x + \underbrace{\mathrm{lcm}(c_f, c_g)}_{= k_{c_g} c_g} - u) \right\}$$
$$\vee \sup_{g(T_g) \leq v \leq x - f(T_f)} \left\{ f^{-1}_{\uparrow,p}(x + \underbrace{\mathrm{lcm}(c_f, c_g)}_{= k_{c_f} c_f} - v) + g^{-1}_{\uparrow,p}(v) \right\}$$
$$\overset{(x - u \geq g(T_g), x - v \geq f(T_f))}{=} \sup_{f(T_f) \leq u \leq x - g(T_g)} \left\{ f^{-1}_{\uparrow,p}(u) + g^{-1}_{\uparrow,p}(x - u) \right\} + k_{c_g} d_g$$
$$\vee \sup_{g(T_g) \leq v \leq x - f(T_f)} \left\{ f^{-1}_{\uparrow,p}(x - v) + g^{-1}_{\uparrow,p}(v) \right\} + k_{c_f} d_f$$
$$= \left(f^{-1}_{\uparrow,p} \,\overline{\otimes}\, g^{-1}_{\uparrow,p}\right)(x) + \max\left(k_{c_g} \cdot d_g, k_{c_f} \cdot d_f\right),$$

where we used Theorem 6 in the fourth line. It follows then that

$$T_{\overline{\otimes}_p^{-1}} = f(T_f) + g(T_g) + \mathrm{lcm}(c_f, c_g), \tag{38}$$
$$d_{\overline{\otimes}_p^{-1}} = c_{h_{pp}} = \mathrm{lcm}(c_f, c_g), \tag{39}$$
$$c_{\overline{\otimes}_p^{-1}} = \max\left(k_{c_g} \cdot d_g, k_{c_f} \cdot d_f\right). \tag{40}$$

Next, for the outer part we consider the lower pseudoinverse of the above result, over the interval $[f(T_f) + g(T_g), +\infty[$. From Lemmas 16 and 17, it follows that

$$T_{h_{pp}} \overset{(33)}{=} \left(f_p^\wedge \otimes g_p^\wedge\right)^{-1}_{\uparrow,[T_f+T_g,+\infty[}\left(T_{\overline{\otimes}_p^{-1}}\right)$$
$$\overset{(24)}{=} \sup\left\{ t \geq T_f + T_g \mid f_p^\wedge \otimes g_p^\wedge(t) \leq T_{\overline{\otimes}_p^{-1}} \right\}.$$

From Theorem 15 it follows also that

$$d_{h_{pp}} = c_{\overline{\otimes}_p^{-1}} = \max\left(k_{c_g} \cdot d_g, k_{c_f} \cdot d_f\right),$$
$$c_{h_{pp}} = d_{\overline{\otimes}_p^{-1}} = \mathrm{lcm}(c_f, c_g).$$

This finishes the proof. ◀

▶ **Corollary 12.** *Given f and $g \in \mathcal{U}$ which are left-continuous and non-decreasing in $[T_f, +\infty[$ and $[T_g, +\infty[$, respectively, and are neither UC nor UI. Then, to compute $f_p^{\wedge} \otimes g_p^{\wedge}$ via $(\max, +)$ isomorphism of restricted functions, it is sufficient to use sequences $S_{f_p^{\wedge}}^{I'_{f_p^{\wedge}}}$ and $S_{g_p^{\wedge}}^{I'_{g_p^{\wedge}}}$, with*

$$I'_{f_p^{\wedge}} = \left[T_f, T'_f + 2 \cdot k_{c_f} \cdot d_f\right], I'_{g_p^{\wedge}} = \left[T_g, T'_g + 2 \cdot k_{c_g} \cdot d_g\right]. \tag{22}$$

where we used $T'_f = \sup\{t \geq T_f \mid f(t) = f(T_f)\}$ and $T'_g = \sup\{t \geq T_g \mid g(t) = g(T_g)\}$.[9]

Proof. The proof is based on using Proposition 25, as we did in the proof of Theorem 11. We thus compute $f_p^{\wedge} \otimes g_p^{\wedge}$ through $f_{\uparrow,p}^{-1} \overline{\otimes} g_{\uparrow,p}^{-1}$. In the proof of Theorem 11, we derived the UPP properties of the latter as (38), (39) and (40). Thus, we need to compute $S_{\overline{\otimes}_p^{-1}}^{I_{\overline{\otimes}_p^{-1}}} = S_q^{I_q} \overline{\otimes} S_r^{I_r}$, where $q \coloneqq f_{\uparrow,p}^{-1}$ and $r \coloneqq g_{\uparrow,p}^{-1}$ with

$$I_{\overline{\otimes}_p^{-1}} = [f(T_f) + g(T_g), f(T_f) + g(T_g) + 2 \cdot \mathrm{lcm}(c_f, c_g)[, \tag{41}$$
$$I_{f_{\uparrow,p}^{-1}} = [f(T_f), f(T_f) + 2 \cdot \mathrm{lcm}(c_f, c_g)] = \left[f(T_f), f(T_f) + 2 \cdot k_{c_f} \cdot c_f\right], \tag{42}$$
$$I_{g_{\uparrow,p}^{-1}} = [g(T_g), g(T_g) + 2 \cdot \mathrm{lcm}(c_f, c_g)] = \left[g(T_g), g(T_g) + 2 \cdot k_{c_g} \cdot c_g\right]. \tag{43}$$

Next, we derive which values of $f_p^{\wedge}$ and $g_p^{\wedge}$ are needed in order to compute the values of $f_{\uparrow,p}^{-1}$ in $I_{f_{\uparrow,p}^{-1}}$ and $g_{\uparrow,p}^{-1}$ in $I_{g_{\uparrow,p}^{-1}}$. We focus, without loss of generality, on $f_p^{\wedge}$, and obtain via Lemma 18 that

$$f_{\uparrow,p}^{-1}(f(T_f)) = \sup\{t \geq T_f \mid f(t) \leq f(T_f)\} = T'_f,$$

and using Theorem 6

$$f_{\uparrow,p}^{-1}(f(T_f) + 2 \cdot \mathrm{lcm}(c_f, c_g)) = f_{\uparrow,p}^{-1}(f(T_f)) + 2 \cdot k_{c_f} \cdot d_f = T'_f + 2 \cdot k_{c_f} \cdot d_f.$$

Hence, it is sufficient to use $\left[T'_f, T'_f + 2 \cdot k_{c_f} \cdot d_f\right]$ for $f_{\uparrow,p}^{-1}$, and $\left[T'_g, T'_g + 2 \cdot k_{c_g} \cdot d_g\right]$ for $g_{\uparrow,p}^{-1}$, to compute $f_{\uparrow,p}^{-1} \overline{\otimes} g_{\uparrow,p}^{-1}$. Then, to compute the lower pseudoinverse, we do not require any additional value from f and g. We do so however for the last step, due to the loss of information implied by having T'_f and T'_g as left boundaries. Using Proposition 25, the reconstruction operator requires us to know that $f(t) = f(T_f)\ \forall T_f \leq t \leq T'_f$, we obtain

$$I'_{f_p^{\wedge}} = \left[T_f, T'_f + 2 \cdot k_{c_f} \cdot d_f\right], I'_{g_p^{\wedge}} = \left[T_g, T'_g + 2 \cdot k_{c_g} \cdot d_g\right].$$ ◀

▶ **Theorem 13** (Mix and Match ((min,+) Convolution))**.** *Let f and $g \in \mathcal{U}$ which are neither UC nor UI, and are left-continuous and non-decreasing in $[T_f, +\infty[$ and $[T_g, +\infty[$, respectively. Let $I_{f_p^{\wedge}}, I_{g_p^{\wedge}}$ be the intervals to compute $f_p^{\wedge} \otimes g_p^{\wedge}$ according to (13), and let $I'_{f_p^{\wedge}}, I'_{g_p^{\wedge}}$ be the intervals to compute the same through Corollary 12. Then $I_{f_p^{\wedge}} \cap I'_{f_p^{\wedge}}, I_{g_p^{\wedge}} \cap I'_{g_p^{\wedge}}$ are valid intervals to compute $f_p^{\wedge} \otimes g_p^{\wedge}$.*

[9] The suprema are attainable since the functions are left-continuous over the respective intervals.

Proof. We prove the result for $I_{f_p^\wedge} \supset I'_{f_p^\wedge}$, $I_{g_p^\wedge} = I'_{g_p^\wedge}$, and $T_{h_{pp}}$ as given by (18). The remaining cases follow by commutativity and / or along the same lines. Let $I_{f_p^\wedge} = [T_f, b]$, $I'_{f_p^\wedge} = [T_f, a]$ with $a < b$. Moreover, we define $d_{h_{pp}}$ according to (19). We show now that the values of f in $]a, b]$ are not necessary for the computation. Therefore, assume that this is not the case, i.e., there exists some $t^* \in \left[T_f + T_g, T_{h_{pp}} + d_{h_{pp}}\right]$, $s^* \in\,]a, b]$ such that

$$\inf_{0 \leq s \leq t^*,\, s \notin]a,b]} \left\{ f_p^\wedge(s) + g_p^\wedge(t^* - s) \right\} > f_p^\wedge \otimes g_p^\wedge(t^*) = f_p^\wedge(s^*) + g_p^\wedge\,(t^* - s^*) =: z^*.$$

From $I'_{f_p^\wedge} = [T_f, a]$ and Lemma 18 it follows that $I_{f^{-1}_{\uparrow,p}} = [f(T_f), f(a)]$. Let us consider now $\left(f_p^\wedge \otimes g_p^\wedge\right)^{-1}_{\uparrow}(z^*) = f^{-1}_{\uparrow,p} \,\overline{\otimes}\, g^{-1}_{\uparrow,p}(z^*)$. We distinguish two cases: either $z^* \in I_{\overline{\otimes}_p^{-1}}$, computed according to (41), or it is larger than the upper boundary of $I_{\overline{\otimes}_p^{-1}}$. In the first case, i.e., $z^* < f(T_f) + g(T_g) + 2 \cdot \operatorname{lcm}(c_f, c_g)$, we have

$$f^{-1}_{\uparrow,p} \,\overline{\otimes}\, g^{-1}_{\uparrow,p}(z^*) = \sup_{0 \leq v \leq z^*} \left\{ f^{-1}_{\uparrow,p}(v) + g^{-1}_{\uparrow,p}(z^* - v) \right\} = f^{-1}_{\uparrow,p}(v^*) + g^{-1}_{\uparrow,p}\,(z^* - v^*)\,,$$

where $v^* \in I'_{f^{-1}_{\uparrow,p}} = [f(T_f), f(a)]$ such that the supremum is attained (which exists being the upper pseudoinverses right-continuous and due to Proposition 22). Moreover, since $I'_{f^{-1}_{\uparrow,p}}$ and $I_{g^{-1}_{\uparrow,p}}$ are sufficient to compute $f^{-1}_{\uparrow,p} \,\overline{\otimes}\, g^{-1}_{\uparrow,p}(z)$ for any $z \in I_{\overline{\otimes}_p^{-1}}$ (Corollary 12), it follows that $v^* \in I'_{f^{-1}_{\uparrow,p}}$ and $z^* - v^* \in I_{g^{-1}_{\uparrow,p}}$, hence we do not need any value of f and g outside of these intervals to perform the computation for z^* in particular. But, since the interval $]a, b]$ was not used to compute $I'_{f^{-1}_{\uparrow,p}}$, this is a contradiction to $s^* \in\,]a, b]$ being needed for $f_p^\wedge \otimes g_p^\wedge$.

In the second case ($z^* \geq f(T_f) + g(T_g) + 2 \cdot \operatorname{lcm}(c_f, c_g)$), $f^{-1}_{\uparrow,p} \,\overline{\otimes}\, g^{-1}_{\uparrow,p}(z^*)$ can be computed by applying the UPP property meaning that

$$f^{-1}_{\uparrow,p} \,\overline{\otimes}\, g^{-1}_{\uparrow,p}(z^*) = f^{-1}_{\uparrow,p} \,\overline{\otimes}\, g^{-1}_{\uparrow,p} \left(z^* - k \cdot d_{\overline{\otimes}_p^{-1}} \right) + k \cdot c_{\overline{\otimes}_p^{-1}}$$

for $d_{\overline{\otimes}_p^{-1}}$ and $c_{\overline{\otimes}_p^{-1}}$ described in (39) and (40), respectively, and some $k \in \mathbb{N}$ such that $z^* - k \cdot d_{\overline{\otimes}_p^{-1}} \in I_{\overline{\otimes}_p^{-1}}$. We can follow for the latter the same reasoning as in the first case, thus having again a contradiction to the assumption that $s^* \in\,]a, b]$ is needed for the computation of $f_p^\wedge \otimes g_p^\wedge$. ◀

Low-Overhead Online Assessment of Timely Progress as a System Commodity

Weifan Chen
Boston University, MA, USA

Ivan Izhbirdeev
Boston University, MA, USA

Denis Hoornaert
Technische Universität München, Germany

Shahin Roozkhosh
Boston University, MA, USA

Patrick Carpanedo
Boston University, MA, USA

Sanskriti Sharma
Boston University, MA, USA

Renato Mancuso
Boston University, MA, USA

Abstract

The correctness of safety-critical systems depends on both their logical and temporal behavior. Control-flow integrity (CFI) is a well-established and understood technique to safeguard the logical flow of safety-critical applications. But unfortunately, no established methodologies exist for the complementary problem of detecting violations of control flow timeliness. Worse yet, the latter dimension, which we term *Timely Progress Integrity* (TPI), is increasingly more jeopardized as the complexity of our embedded systems continues to soar. As key resources of the memory hierarchy become shared by several CPUs and accelerators, they become hard-to-analyze performance bottlenecks. And the precise interplay between software and hardware components becomes hard to predict and reason about. *How to restore control over timely progress integrity?* We postulate that the first stepping stone toward TPI is to develop methodologies for Timely Progress Assessment (TPA). TPA refers to the ability of a system to live-monitor the positive/negative slack – with respect to a known reference – at key milestones throughout an application's lifespan. In this paper, we propose one such methodology that goes under the name of *Milestone-Based Timely Progress Assessment* or MB-TPA, for short. Among the key design principles of MB-TPA is the ability to operate on black-box binary executables with near-zero time overhead and implementable on commercial platforms. To prove its feasibility and effectiveness, we propose and evaluate a full-stack implementation called *Timely Progress Assessment with 0 Overhead* (TPAw0v). We demonstrate its capability in providing live TPA for complex vision applications while introducing less than 0.6% time overhead for applications under test. Finally, we demonstrate one use case where TPA information is used to restore TPI in the presence of temporal interference over shared memory resources.

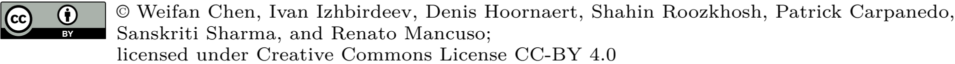
2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases progress-aware regulation, hardware assisted runtime monitoring, timing annotation, control flow graph

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.13

Supplementary Material *Software (Source Code)*: `https://github.com/wchen258/TPAw0v`
archived at `swh:1:dir:94e4198f133a2fb5eca90f45a5875eef2157ccee`

Funding *Denis Hoornaert*: Denis Hoornaert was supported by the Chair for Cyber-Physical Systems in Production Engineering at TUM and the Alexander von Humboldt Foundation.
Renato Mancuso: The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grants number CCF-2008799 and CNS-2238476.



© Weifan Chen, Ivan Izhbirdeev, Denis Hoornaert, Shahin Roozkhosh, Patrick Carpanedo, Sanskriti Sharma, and Renato Mancuso;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

Prompted by the proliferation of cyber-physical, safety-critical, and human-in-the-loop systems, the notion of *timeliness* in computing has gained growing interest. The accompanying demand for complex, robust, and computationally demanding control algorithms has led the real-time community to shift its focus away from simpler hardware platforms to high-complexity and high-performance platforms. As the complexity increases in platforms, many challenges have surfaced at all the software/hardware stack layers. It is well understood that the logic of an application can be hardened against control-flow attacks via Control Flow Integrity (CFI) [39] methods. But no established methodologies exist for the dual problem in the temporal domain, for which we coin the name *Timely Progress Integrity* (TPI).

The introduction of heterogeneous multi-core System-on-Chip (SoC) along with complex memory subsystem mechanisms at the hardware level has complicated the problem of ensuring TPI. In particular, memory subsystem hierarchy such as shared [49], non-blocking caches [62], shared memory controller [66], and DRAM organization [65] are among noteworthy sources of interference. The interplay of each element mentioned above renders the task of guaranteeing timeliness an open challenge. In turn, the introduced complexity in SoCs and their ongoing proliferation have prompted the need for more complex operating systems and OS-level scheduling strategies, which exacerbate the problem.

The real-time community has achieved important milestones towards restoring predictability [45, 48]. But traditional methods – e.g. static WCET analysis, memory resource partitioning – have largely focused on respecting end-to-end constraints in the worst case, as opposed to reason on the current (timely) rate of progress of live applications. Solutions that leverage code instrumentation have been proposed to checkpoint the progress of applications at runtime [37, 38, 58], but a system-level solution that can operate on black-box binaries and *inform* a rich OS of the expected/detected progress of its applications for it to make informed management decisions has not been studied. We propose one such solution.

Timely progress assessment as a system commodity. Reasoning about, controlling, and reacting to changes in the progress of safety-critical applications is the goal. Thus, the ability to assess an application's progress must become a system commodity. In referring to this capability, we coin the term *Timely Progress Assessment* (TPA). With TPA, a system is capable of detecting deviations in the timely progress of an application well before a deadline is missed, providing the ability to enact corrective measures toward ensuring TPI early on. On the other hand, when faster-than-expected progress is detected, the accumulated slack can be redistributed to other workloads. Thus, TPA is an enabling capability towards Timely Progress Integrity (TPI).

This article presents a system design and methodology called Milestone-Based Timely Progress Assessment (MB-TPA) to perform TPA on live black-box applications. MB-TPA relies on binary analysis and widely available on-chip tracing subsystems to detect the timely completion of intermediate progress *milestones* for an application under analysis. We discuss a full-stack implementation of MB-TPA on commercial hardware. The implemented TPA subsystem was termed Timely Progress Assessment with 0 Overhead (TPAw0v), which we describe and evaluate. We show that MB-TPA (1) introduces negligible ($< 0.6\%$) overhead to the monitored applications under test. MB-TPA is able to provide live progress assessment even if a low-power CPU is used to monitor a high-performance CPU. In light of the discussion above, we make the following contributions:

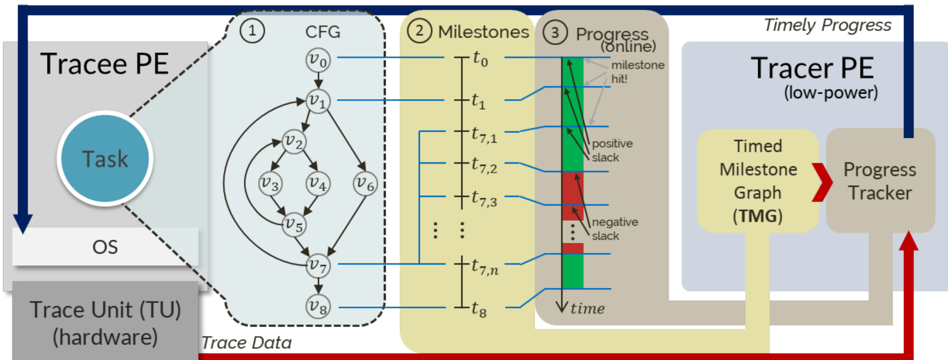


Figure 1 High-level overview of the proposed system design. The CFG of the target application is analyzed to produce a Timed Milestone Graph (TMG). Together with the online data produced by the Trace Unit (TU), a progress tracker assesses timely progress and reports to the OS. The OS can take corrective measures accordingly.

1. We propose the concept of TPI as a requirement that is complementary to CFI to marry logical and temporal integrity.
2. We demonstrate for the first time that online progress assessment without source code instrumentation for black-box applications is feasible in commercial platforms.
3. We present a method called MB-TPA, that solves key challenges with offline milestone identification and online progress assessment.
4. Provide a full-stack proof-of-concept implementation and evaluation of MB-TPA for multi-core Arm Aarch64 SoCs. We refer to our implementation as TPAw0v.
5. Showcase three use cases focusing on real-world vision applications. We leverage TPA to (1) enforce the WCET of a target application; (2) achieve controlled performance degradation of the target application by modulating the degree of contention over shared memory resources; and (3) retrieve live progress-aware profiles of the microarchitectural resources used by the target application.

1.1 Overview of Proposed System Design for MB-TPA

The goal of making TPA a system commodity imposes two main design constraints. First and foremost, it must be possible for a system to enact TPA on potentially unknown (black-box) applications that cannot be recompiled from sources. At the same time, TPA shall be carried out with negligible temporal overhead. An overview of the proposed system design is provided in Figure 1. The design involves the use of a *Tracee* PE (Processing Element) where a target application (Task) runs unmodified. A second low-power/low-performance PE, the *Tracer*, controls the TU to generate trace data transparently to the application under analysis. Section 4 discusses the system assumptions that enable instantiating the proposed system.

Initially, the unmodified binary of the target application is analyzed to construct its Control Flow Graph (CFG) – (1) in Figure 1. Through a sequence of refinement steps, a Timed Milestone Graph (TMG) is derived from the original CFG. An in-depth description of the methodology proposed to produce a TMG from a CFG is provided in Section 5. The TMG is a graph of milestones, each corresponding to some vertex in the original CFG, with associated time information – (2) in Figure 1. At runtime, the tracer uses the input TMG and the data received from the TU and detects (un)timely completion of the milestones –

(3) in Figure 1. The detected positive/negative progress slack is reported back to the OS to enact management decisions. The tracer was implemented as bare-metal firmware running on a low-power CPU. The details of our implementation are provided in Section 7.

2 Related Works

Our work finds context in the broad literature concerned with ensuring that the timeliness of a (set of) critical task(s) can be controlled. In modern platforms, the progress of application workload can be impacted by many factors. These include scheduling decisions, overheads introduced by preemptions and migrations [15, 40, 50] and I/O activity [16, 33, 55, 67], unpredictable cache effects such as self-eviction [17, 27], and contention over shared hardware resources [45, 48]. The set of solutions proposed by the real-time community to reason about the timeliness of an application can be placed on a spectrum. On one end are static analysis approaches; on the other are runtime monitoring solutions.

Timeliness (interpreted as the ability to meet a completion deadline) in static analysis approaches [5, 20, 31, 47] is ensured by computing an absolute worst-case execution time (WCET) which is then used to compute a worst-case response time (WCRT). The promise is that WCET/WCRT computation is done by considering the initial state(s) and sequences of system states that lead to the worst possible temporal application behavior. Given the sheer complexity of interactions between applications, system-level, and hardware-level components, static approaches seldomly scale to modern multicore processors [30, 35, 46].

Recently, approaches based on runtime monitoring have gained momentum. At a high level, these approaches select a *monitoring scheme* and a set of *system metrics*. By monitoring such metrics online – and taking management actions accordingly – the system detects and/or avoids undesired outcomes, e.g., uncontrolled contention over a shared resource or a deadline miss for a critical task. To properly contextualize our work with respect to related approaches, we categorize runtime monitoring solutions into *software-* and *hardware-based* approaches.

2.1 Software-based Monitoring and Progress Assessment

The vast majority of solutions for runtime monitoring and progress assessment introduce software mechanisms to enact monitoring and/or enact management decisions. We distinguish four main sub-categories discussed below.

(A) Memory Bandwidth Regulation. Memory bandwidth controllers [59, 62, 66] monitor the number of last-level data cache refills and/or writebacks against an allocation budget. Periodically, they stall the processor if the consumed budget is exceeded. Although bandwidth regulation aims to prevent the unbalanced progress of co-running applications sharing the same memory subsystem, no exact knowledge of application progress is constructed.

(B) Feedback Control Scheduling. Feedback control scheduling represents another form of runtime monitoring. In the context of real-time systems, this approach was pioneered in [60]. The key insight is that the knowledge of task parameters computed offline is refined via online observations performed at task completion. Task admission is geared accordingly to meet a target deadline miss ratio. Since the aforementioned original work, a broad literature on feedback control scheduling has surfaced [19, 44, 53].

(C) Early Deadline Detection. Early deadline detection is the runtime monitoring technique at the center of adaptive mixed-criticality scheduling (AMC) [14, 18]. The key insight is that multiple (at least two) runtime estimates are expressed for high-criticality tasks with

varying degrees of pessimism. Initially, an optimistic execution time is assumed, and an early deadline (virtual deadline) is set accordingly. At runtime, the system detects if any early deadline is missed and takes corrective measures accordingly by dropping [13, 24, 29, 41, 54] or degrading low-criticality tasks [28, 42]. Like feedback control scheduling, runtime monitoring in AMC systems is limited to detecting an application's completion (or lack thereof) by a set (early) deadline. This is equivalent to detecting a single milestone at the application's end.

(D) Progress Detection. A handful of works attempt to provide a finer-grained understanding of the progress of target applications. For instance, the work in [26] periodically monitors the number of retired instructions to detect a sequence of phases in which the application's usage of hardware resources changes. This approach is inherently limited to applications with a single execution path. In a way that is more closely related to our work, the works in [36–38, 58] consider the full CFG of a target application. These works propose to instrument a target application's code via source-to-source translation and/or a modified compiler. The goal is to insert watchpoints at which progress is assessed in software. At runtime, when the execution reaches a watchpoint, an interrupt/syscall is issued to decide whether the system should raise the critical level and drop/suspend low-criticality jobs. In previous works, the overhead is a limiting factor. Kritikakou et al., in an extension [36] to [37, 38], propose an algorithm to ignore some checkpoints in order to reduce the overhead. The authors of PAStime [58] place watchpoints outside of loops to limit the overhead.

Compared to the works in the four categories surveyed above, this paper sets itself apart because we aim at precise progress assessment without the need to modify/recompile the application under analysis. Importantly, we are able to express a notion of timely progress even if the control flow is input dependent. Finally, for the first time, we demonstrate that leveraging widely available tracing hardware for progress assessment is possible and minimizes runtime overhead. Indeed, our system never interrupts the application under analysis while its progress is assessed asynchronously and, therefore, off the critical path.

2.2 Run-time Monitoring via Hardware

Comparatively, less work has explored progress monitoring via specialized hardware support. Most notably, Lo et al. proposed a customized hardware architecture for runtime monitoring of hard real-time tasks [43]. Apart from timely progress, the work aims to monitor other safety properties, such as the presence of uninitialized memory and the correctness of return addresses. Differently from [43], we focus on commercially available hardware.

Few works have also proposed to leverage trace unit at runtime to perform control flow integrity [25, 34], while FPGA-based trace decoders were proposed in [6, 32]. We are the first to utilize a trace unit online to perform timely progress assessment in real-time systems.

3 Background

All the aforementioned approaches for progress assessment [36–38, 43, 58] consider the CFG of critical tasks. Kritikakou et al. have constructed a formal grammar to extract the CFG from a wide range of binaries [37]. There are also a plethora of tools capable of such transformations [57]. The following section provides a brief overview of CFGs.

(A) Basic Block and Branch Instructions. A *basic-block* (BB) is a contiguous sequence of non-branching (assembly) instructions ending with a branching instruction. In other words, except for the last instruction, a basic block only contains instructions for which the *program*

counter (PC) of the CPU – or more generally, processing element (PE) – is monotonously incremented. A branch instruction has one or more target BBs. For example, in ARM® AARCH32/64 [11], an unconditional branch instruction b would take PC to the operand address, the beginning of a BB. Conditional branch instructions b.cond have two target BBs. When b.cond is executed, if the condition is met, the PC is set to the operand address, otherwise to the instruction following the b.cond instruction. The return instruction ret can have more than two target BBs. It is possible to statically know its target(s) if the call sites can be fully enumerated.

(B) Control Flow Graph. A program's control flow transfer information can be expressed as a directed graph $\mathcal{G} = (V, E)$. A node $n \in \mathcal{V}$ represents a BB, and an edge $(n_p, n_s) \in \mathcal{E}$ indicates that the branch instruction in n_p has n_s as a target. We term this type of edge a *normal edge*. In practice, it is unnecessary to expand the complete CFG for runtime monitoring purposes. Instead, one can view the program as a collection of functions with the entry point at main [37]. Thus, if no watchpoints are to be placed inside a function f, all nodes and edges related to f can be removed, and an edge from the caller BB to the returning BB is added. We refer to this operation as the *folding* of function f, and to the newly added edge as the *folding edge*.

(C) Processor Trace. The *processor trace*, often called the *embedded trace*, is a highly compressed data stream generated by a PE when executing binary code. The trace contains the necessary information to reconstruct the history of the executed program. Trace generation is often used for debugging and performance evaluation purposes. As such, the on-chip hardware circuitry dedicated to processor trace generation, i.e., the *trace unit* (TU), is designed to introduce negligible overhead, if at all. The typical use of processor tracing capabilities is in conjunction with external trace probes. In this case, the system runs without modification while external hardware (probe) is connected to a physical trace port. The probe collects (portions of) the produced processor trace data for offline analysis. Two broadly used hardware probes are the Lauterbach® PowerTrace [1] and the Green Hills® Probe V4 [2].

Trace generation units are almost ubiquitous in embedded and general-purpose high-performance CPUs. Many embedded modern processors include more or less capable on-chip TU's. For example, ARM's lineup of hardware modules for tracing and debugging that fall under the CoreSight [7] umbrella includes TU modules such as the Embedded Trace Macrocell (ETM) and Program Trace Macrocell (PTM). The TU solution from Intel® is called Processor Trace (PT). The PT infrastructure has been introduced in 5th generation Intel processors, promising overheads below 5% [21, Chapter 32]. RISC-V also has its own embedded trace specification [4].

Since trace data is produced at the same (or comparable) timescale as instruction execution, the data bandwidth is usually considerably high, even after many lossless compression techniques are applied. A common compression technique only reports the progression of BBs instead of individual instructions. If the current BB is known, then a single bit of information is enough to encode whether the (conditional) branch at the end of the BB is taken or not. When this information is combined with static knowledge of the binary under analysis, the entire control flow can be recovered. If the current BB ends with an indirect branch such as a function return, the trace provides an explicit branching address.

Trace data include additional metadata about the processor state. For instance, in systems that support multiple tasks, the context ID of the process in execution (as determined by the OS) is also generated. The virtual machine ID is also included for systems with hardware

virtualization extensions. Similarly, information that can identify an interrupt context (interrupt taken, interrupt type, interrupt return) is also provided. Other valuable meta-information for performance analysis can also be included, such as the cycle counter and the occurrence of other microarchitectural events.

A TU includes hardware resources that go beyond embedded trace generation to perform some degree of pre-processing. For instance, trace packet filters, counters, sequencers/formatters, external input selectors, or aggregators to combine trace data from multiple sources (e.g., multiple CPUs) can be included in the TU subsystem.

4 System Model and Assumptions

In this section, we describe the assumed system model upon which our MB-TPA is formulated. These assumptions also dictate the system requirements to implement the proposed MB-TPA, and ultimately introduce timely progress assessment as a commodity.

4.1 System-level Assumptions

(A) Tracee PE and Tracer PE. We assume that at least two PEs are present: (1) a main PE (or *tracee*) running the application under analysis and (2) the other PE serving as a *tracer*. Note that no assumption on the components' nature nor performance is made, meaning that the tracer and tracee can be implemented using various technologies. For instance, a system could have high-performance PEs as tracee and be monitored by a low-performance real-time core or specialized hardware implemented as an ASIC or on an FPGA.

(B) Address Range Filters. We assume that the tracee features a TU providing at least one range-programmable instruction address filter. That way, the TU can be programmed to trace specific address ranges corresponding to the immediate next milestones.

(C) On-chip Trace Data Path. We assume that an on-chip data path exists through which the TU-generated trace data stream can be forwarded to the tracer, as it is commonly the case for high-performance embedded systems. For instance, many ARM-based COTS platforms offer dedicated on-chip trace routing and storage within the CoreSight [7] infrastructure[1].

4.2 Application-level Assumption

(A) Single Binary. This work targets single-binary applications running on the tracee. No restrictions on the number of software layers used by the tracee are imposed, meaning that the target applications can equally run on top of a full-fledged OS, inside a virtual machine on a hypervisor, or as a bare-metal application. The binary is sufficient to apply the proposed MB-TPA: we place no assumption on the availability of the target's source code, nor that it can be recompiled and/or binary-instrumented. The goal is that MB-TPA can be automatically employed by a system.

[1] Trace data routing components include the Embedded Trace Router (ETR), Embedded Trace FIFO, and Funnel. Storage components include the Embedded Trace Buffer and Trace Memory Controller.

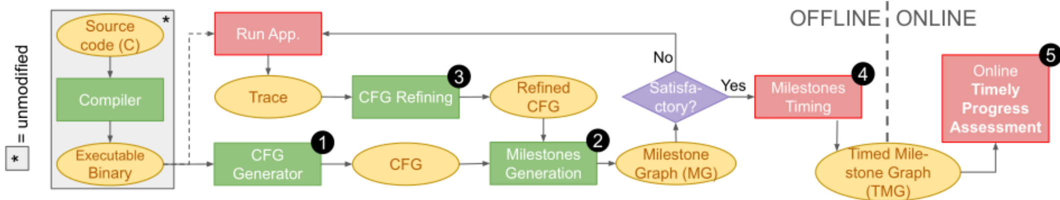


Figure 2 Abstract tool-chain proposed. Ovals represent the inputs and outputs, red rectangles represent timing-sensitive tools, and green rectangles represent timing-insensitive tools.

(B) Single Entry/Exit. Without loss of generality, we assume that the entry BB address and the exit BB address are (1) known, (2) within the target's binary, and (3) they are linked by at least one valid control path. The entry and exit BB of a function generally[2] represent a valid selection. Otherwise, for applications implementing time- or event-triggered logic in an infinite loop, the first and last BBs of the loop iteration can be selected as the entry and exit BB points. If the debug symbols are part of the binary, the entry/exit BB selection can be automated (e.g., given a function name).

(C) Availability of Representative Inputs. Finally, for complex and input-dependent applications, we assume that a set of representative input vectors is available to experimentally produce (offline) a nominal progress reference to check against during the online phase.

5 Methodology for Milestone-Based Timely Progress Assessment

We hereby describe the proposed Milestone-Based Timely Progress Assessment in its different phases. With reference to Figure 1, this section details the design choices and steps involved in going from CFG creation to TMG generation. A bird's eye view of MB-TPA is depicted in Figure 2. The following sections cover the numbered steps (1) through (5) in detail.

5.1 Intuition of Key Challenges and Solutions

(A) Monotonic Progress in Black-Box Binaries. As discussed in Section 3, the execution of a binary implies control flow transfer over a graph. On the other hand, the idea that a target application must execute (and thus complete) on time implies a monotonic notion of progress. Therefore, the first challenge we face is to construct a notion of progress given black-box application binaries.

Our solution consists in identifying BBs that represent *progress milestones* (Section 5.3). Intuitively, a BB is a progress milestone (a.k.a. MBB) if, once reached, it is possible to conclude that a sizable amount of progress has been made by the application logic. Milestone identification is done through a combination of (1) *CFG extraction*, (2) *CFG refinement* by observing concrete runs of the target, and (3) applying the milestone placement algorithm. The output of the algorithm is a milestone graph (MG). The procedure is detailed in Section 5.3.

(B) Keeping up with Trace Data. Timely progress assessment has to be performed in a timely manner. Assuming that a valid set of MBBs has been identified, the goal is to detect the completion of milestones at the tracer as soon as they are reached on the tracee, or with

[2] If no infinite loops are present in the function nor in any other routine that can be called by it.

negligible delay. This way, the tracer can promptly assess TPI violations and trigger any correction countermeasure if necessary. Conversely, if the tracer lags significantly behind the tracee, then it might be too late to act upon detected TPI violations – and one might as well detect TPI violations at target completion instead.

What makes this challenging? The first issue might reside in the **latency** for the propagation of TU-generated data to the tracer PE. As we evaluate in Section 8.1, it is not an issue if the tracer and tracee are different PEs on the same SoC. A second (and more problematic) issue is the limited **bandwidth** of the on-chip channels via which trace data is streamed. Despite aggressive trace compression, allowing the TU to stream trace data unrestrictedly leads to buffer overflows due to the performance gap between tracer and tracee PEs. These overflows can occur both within the TU or at the interface between the TU and the tracer, preventing any packet from reaching the tracer. Thus the naïve solution of constantly streaming data from the TU and matching against `MBB`s does not work.

(C) Dynamic TU Reconfiguration. To reliably ensure milestone detection, we propose to dynamically reconfigure the TU so that it is silent for most of the time and only emits bare minimum packets when the event of interest happens – i.e., one of the next `MBB`s is reached. At this point, a new set of `MBB`s to monitor is configured. The TU then becomes silent again, waiting for the next milestone. In this paradigm, the TU only emits sporadic and short-lived signals, thus consuming a fraction of the sustainable trace bandwidth. The information of which `MBB`s to monitor after a given `MBB` is reached is expressed in the TMG.

5.2 Trace Blackout Window

Two milestones cannot be placed arbitrarily close to one another. This is a consequence of the dynamic TU reconfiguration. Suppose $\texttt{MBB}_1$ and $\texttt{MBB}_2$ are adjacent, i.e., when the TU has detected that tracee's execution has reached $\texttt{MBB}_1$, then the TU should be reconfigured to detect tracee's execution on $\texttt{MBB}_2$. The reconfiguration typically consists of (1) disabling the TU to reprogram the relevant registers, (2) identifying the `MBB` that has been reached, (3) looking up in the TMG the next set of milestones to detect, and (4) resuming the TU.

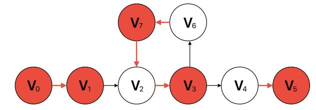
Let t_1 and t_2 denote the time for tracee's execution reaching $\texttt{MBB}_1$ and $\texttt{MBB}_2$ respectively. From the time t_1 at which $\texttt{MBB}_1$ is reached and until the TU is brought back online to monitor $\texttt{MBB}_2$, there is a window of time during which milestones cannot be monitored. We call this the *trace blackout window* and indicate it with the symbol T_r. If the best-case path between $\texttt{MBB}_1$ and $\texttt{MBB}_2$ is such that $(t_2 - t_1) < T_r$, then detection of $\texttt{MBB}_2$ cannot be guaranteed. Our methodology avoids this issue by design.

Formally, call $D(\texttt{MBB}_i, \texttt{MBB}_j) \in \mathbb{R}^+$ the time-cost to reach $\texttt{MBB}_j$ starting from $\texttt{MBB}_i$. Clearly, this cost is a random variable that depends on the specific path taken and the progress at which the target executes. Moreover, $D(\texttt{MBB}_i, \texttt{MBB}_j) = \infty$ if $\texttt{MBB}_j$ cannot be reached from $\texttt{MBB}_i$. We show that a lower-bound of this cost can be computed and impose that, for any two valid $\texttt{MBB}_i, \texttt{MBB}_j$, it must hold that

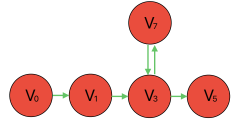
$$\min_{i,j}\{D(\texttt{MBB}_i, \texttt{MBB}_j)\} > T_r. \tag{1}$$

It is worth noting that the blackout window and the sizable progress requirement discussed in the first challenge in Section 5.1 both require the distance between two milestones to be sufficiently large. In practice, the blackout window is generally smaller – we derive this parameter for our implementation in Section 8.1. Thus ensuring that enough progress occurs between milestones implies that the constraint imposed by the blackout window is also met.

(a) The extracted CFG. Red edges are folding.



(b) Nodes satisfying the constraint are colored red.



(c) Remove white nodes, add corresponding edges.

Figure 3 Illustrative MG generation for the `main` of the `disparity` benchmark.

5.3 Milestone Graph Construction (Step 1 and 2)

Figure 3 depicts the intuition behind the Milestone Graph (MG) construction procedure. First, the CFG of the target application is extracted (Figure 3a). The CFG is annotated by adding a weight on each edge that is indicative of the temporal distance between two nodes. Then a subset of nodes satisfying the constraint expressed in Eq. 1 is selected – the red nodes in Figure 3b. Finally, new edges are added to the red nodes to maintain reachability relationships, as per Figure 3c. The resultant digraph is a valid MG.

(A) CFG Notation. Given a target black-box binary, the CFG is extracted (Step 1 in Figure 2). This is a digraph $\mathcal{G}^{CFG} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ and $\mathcal{E}$ are the set of all the vertices and edges, respectively. Here a vertex $v_i \in \mathcal{V}$ is a `BB`. An edge $(v_i, v_j) \in \mathcal{E}$ is either normal or folding (Section 3)[3]. For any edge $(v_i, v_j) \in \mathcal{E}$, we assign a per-edge weight w equal to the lower bound on the time to execute the instructions in v_i, including the folded function if its out-edge is folding. A safe albeit inaccurate lower bound can be obtained by dividing the number of instructions in v_i by the maximum clock frequency of the tracee[4]. We define $D(v_i, v_j)$ for any two vertices in $\mathcal{V}$ as the cost of the path (if any) from v_i to v_j with the minimum cost. This is used to lower-bound the minimum time needed to reach v_j from v_i.

(B) MG Notation. An MG $\mathcal{G}^{MG} = (\mathcal{M}, \mathcal{Q})$, is a digraph where $\mathcal{M} \subseteq \mathcal{V}$ is the set of `MBB`s. For each $\texttt{MBB}_i \in \mathcal{M}$, an edge $(\texttt{MBB}_i, \texttt{MBB}_j) \in \mathcal{Q}$ signifies that (1) $\texttt{MBB}_j$ is one of the next milestones to detect after $\texttt{MBB}_i$ has been reached, and (2) Eq. 1 holds. Note: the edge $(\texttt{MBB}_i, \texttt{MBB}_j)$ might not exist in $\mathcal{E}$ because the corresponding `BB`s might not be in an immediate predecessor/successor relationship in $\mathcal{G}^{CFG}$.

(C) Milestone Selection. The milestone selection problem is the following: (1) given a blackout window T_r, color the vertices in $\mathcal{G}^{CFG}$ either red or white; (2) ensure that for any two red nodes, $r_i, r_j \in \mathcal{V}$, $D(r_i, r_j) > T_r$; and (3) find the maximal set of red nodes. Other optimization objectives and heuristics could also be used – e.g, minimizing the sum of distances among red nodes. Finding the optimal solution is not the focus of this work and left as future work; an algorithm that is guaranteed to find a solution (if one exists) is presented here.

[3] Folding all functions except for `main` can already produce meaningful milestone graphs for applications under test. In practice, if the execution time of a function is long, unfolding it to allow milestones to be placed inside can achieve better granularity.

[4] We assume the CPI is greater or equal to one. Notice this might not be true for multi-issue processors.

(D) Graph Coloring Heuristic. The proposed strategy (Step 2 in Figure 2) is described in Algorithm 1. The algorithm first colors all of the vertices red (Line 6–8), then iterates over any non-visited remaining red vertex in DFS search order – thus, starting from the root `BB` (Line 9). Next, for each red vertex r_i we compute the path with the shortest total cost $D(r_i, r_j)$ to all other red vertices in $\mathcal{V}$ (Line 12). If for some r_j $D(r_i, r_j) > T_r$ does not hold (Line 14), color r_j white (Line 15). The full adjacency map D for r_i can be computed using Dijkstra's algorithm [22]. The only adaptation needed to the standard algorithm is to correctly compute $D(v_i, v_i)$, which is always 0 in the traditional algorithm. Instead, we must compute the cost to come back into v_i if v_i was reached, which can be computed as

$$D(v_i, v_i) = \begin{cases} w_i & \text{if } (v_i, v_i) \in \mathcal{E} \\ \min_{(v_i,v_j)\in\mathcal{E}}\{D(v_j, v_i) + w_i\} & \text{otherwise.} \end{cases} \quad (2)$$

Algorithm 1 Constrained Directed Graph Coloring.

```
1  input:
2  |  G^CFG = (V, E), T_r                              ◁ CFG graph and blackout window
3  output:
4  |  Colored G^CFG                     ◁ CFG graph with red-colored marked MBB's
5  init:
6  |  for each v ∈ V do
7  |  |  v.color ← red                                 ◁ Color all nodes red
8  |  end
9  |  R_left ← Topol(V)                ◁ Red vertices to visit, in DFS search order
10 algorithm:
11 |  for each r_i ∈ R_left do
12 |  |  D ← Dijkstra(r_i, G^CFG)              ◁ Get all shortest-paths from r_i
13 |  |  for each r_j ∈ V s.t. r_j.color == red do
14 |  |  |  if D(r_i, r_j) ≤ T_r then
15 |  |  |  |  r_j.color ← white                 ◁ r_j unsafe milestone from r_i
16 |  |  |  |  R_left ← R_left \ {r_j}               ◁ Remove r_j from R_left
17 |  |  |  end
18 |  |  end
19 |  |  R_left ← R_left \ {r_i}                             ◁ Mark r_i as visited
20 |  end
```

To finalize the MG $\mathcal{G}^{MG}$, we proceed as follows. $\mathcal{M}$ is created from the colored $\mathcal{G}^{CFG}$ by removing all the white vertices v_i. To compute $\mathcal{Q}$ from $\mathcal{E}$, we proceed as follows. For each white vertex v_i, remove any self-loop and say that incoming (resp., outgoing) edges are of the form (v_p, v_i) (resp., (v_i, v_s)). Then, for each direct predecessor v_p of an incoming edge, we add all the edges of the form (v_p, v_s) for any direct successor v_s of v_i in $\mathcal{Q}$.

(E) Degree Reduction. Recall that the number of address range registers available (noted M^*) at the TU is limited (Section 3). Intuitively, M^* constraint how many milestones can be monitored by the TU after (one of) the current milestone is hit. After the MG has been produced following the procedure described so far, there is no guarantee that the outdegree (number of outgoing edges) of all the $r_i \in \mathcal{M}$ is below M^*. Thus, a simple pruning strategy is adopted. That is, for each r_i with outdegree greater than M^*, randomly pick one of the outgoing edges and color the vertex pointed by that edge white; then repeat the procedure to remove white nodes. This is done until no vertex with outdegree greater than M^* is found.

We call FINALIZEMG(Colored $\mathcal{G}^{CFG}$, M^*) the routine that takes in input a colored MG and performs edge construction plus MG pruning. Note that the selection of T_r and computation of the weights w can affect the pessimism of Algorithm 1. Moreover, in the presence of loops, the lack of static knowledge about the number of iterations that will be executed at runtime forces the algorithm to assume that only the iteration lower bound is

taken. Finally, error-handling branches that are never taken during nominal execution create short-cut paths (e.g., from entry to exit in a routine) that prevent many intermediate BBs from being colored in red. Nonetheless, the important advantage of this first step is that an initial MG can be produced *without the need to execute the application*.

5.4 Milestone Graph Refinement with Concrete Runs (Step 3)

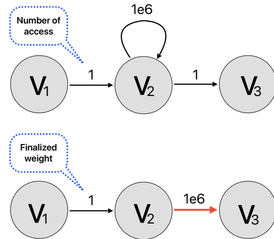
Refinement of the MG with concrete runs (Step 3 in Figure 2) mitigates the problems with static MG construction described in Section 5.3. During refinement, the target is executed on a set of representative inputs, potentially multiple times for each input. Techniques such as *symcretic execution* that combine symbolic execution and concrete runs can be used to automate the generation of representative inputs [23]. For the purpose of this work, we assume that a set of representative inputs has been identified for the target application.

By executing the target application using representative inputs, we are able to measure the temporal distance between two BBs in the CFG and gather additional information about the path(s) taken by the target for each input. Importantly, we can now compute the max/min number of times that each edge $(v_i, v_j) \in \mathcal{E}$ was taken, and thus the min/max number of iterations of each loop is discovered. We record both observed minimum $a_{i,j}$ and maximum $b_{i,j}$ number of times each edge is visited. We only preserve the number of visits, but not their order, despite the trace data does provide the full history of the visited BBs.

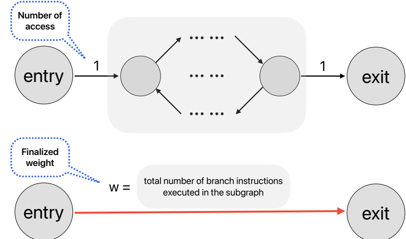
These runs are a way to collect extra information about the target and belong to the offline analysis phase of MB-TPA. In this phase, the TU is configured in a special mode where the TU can slow down the tracee. This is because the high-bandwidth nature of the trace data stream can overflow the internal buffer of the TU and cause information loss. Thus the slowdown ensures that a complete trace from entry to exit of the target is acquired. This is the *only* case in MB-TPA when the target is executed with a (possibly) heavy impact on its runtime due to the activity of the TU.

(A) Branches as a Proxy of Distance. Since the exact temporal progress has been impacted, we need a different metric that correlates (and lower-bounds) the temporal distance between MBBs. The metric must be available from the traces and preserved when the runtime of the application is impacted. Thus, we use the reported number of visited BBs – i.e., the number of executed branch instructions. The advantage is threefold: (1) can be computed directly from the acquired trace without interfacing with any other architectural unit – e.g., a performance measurement unit; (2) when execution flows within the known CFG of the target, one can always retrieve the number of instructions executed; (3) we can put a (conservative) weight on branches to the outside of the CFG under analysis, such as calls to dynamically linked libraries and system calls. From our experience, (2) is unnecessary since the newly acquired information about the min/max number of loop iterations and the presence of never-observed paths already enables much lower pessimism in the MG construction.

Under the new metric, the weight of every normal edge equals to one. The weight of a folding edge depends on the number of branch instructions executed in the folded function which can vary across different sample inputs. To ensure the blackout window condition holds (Eq.1), the weight of a folding edge is assigned to be the minimum across all inputs. Now the effective temporal distance $D(r_i, r_j)$ is the shortest path from r_i to r_j. The following two heuristics can further fold subgraphs with certain properties, so that extra milestones can be placed.



(a) The number of access are $a_{1,2} = 1$, $a_{2,3} = 1$, and $a_{2,2} = 10^6$. After applying the heuristic, the number of access for the self-loop becomes the weight for (v_2, v_3), i.e. $w_{2,3} = 10^6$.



(b) No pair of nodes in the gray region satisfies the constraint. Thus the total number of branch instructions taken inside the region becomes the weight for $w_{\text{en,ex}}$.

Figure 4 Refinement by heuristics. The subgraphs before the heuristics applied are shown on top, in which the number on an edge indicates the number of access $a_{i,j}$. The subgraphs after the heuristics applied are below, in which the number indicates the assigned weight $w_{i,j}$.

(B) Simplify Self-Loops. We identify any BB v_i having only (1) one incoming edge (v_{i-1}, v_i), (2) one outgoing edge (v_i, v_{i+1}), and (3) one self-loop (v_i, v_i). All edges are normal. If the incoming and outgoing edges are both accessed only once, then replace the temporal cost $w_{i,i+1}$ with the minimum number $a_{i,i}$ of self-edge accesses, and remove the self-loop, as shown in Figure 4a. Without this simplification, a suitable milestone candidate v_3 would not be considered due to $D(v_1, v_3) = 2$.

(C) Simplify Sub-graphs. Consider any sub-graph $\mathcal{G}_{sub}^{CFG}$ with a single entry vertex v_{en} and single-exit v_{ex}, in which all edges are normal. If it was unsafe to place any milestones within $\mathcal{G}_{sub}^{CFG}$, then (1) remove all the vertices that belong to $\mathcal{G}_{sub}^{CFG}$ except v_{en} and v_{ex}; (2) add the folding edge $(v_{\text{en}}, v_{\text{ex}})$; and (3) set the temporal cost $w_{\text{en,ex}} = W_{sub}$ to the minimum number of branches W_{sub} observed across all runs inside $\mathcal{G}_{sub}^{CFG}$, as shown in Figure 4b.

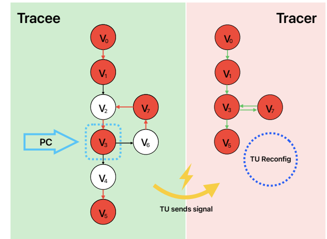
Besides the two heuristics above, the nodes/edges never accessed across all reference inputs are also removed. For this work, we only apply the above refinements, but a large space exists for more advanced heuristics.

5.5 Timed Milestone Graph Generation (Step 4)

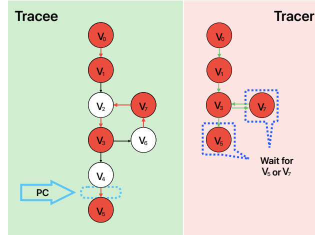
By the end of Step 3 (Section 5.4), an MG refined using concrete runs is obtained. Recall that the goal is to monitor the target's progress online with negligible overhead. At this stage (Step 5 in Figure 2), the (refined) MG is decorated with timeliness information. The output of this step produces a Timed Milestone Graph (TMG) where each milestone is associated with a notion of *when the milestone should be completed* for satisfactory progress.

(A) Milestone Timing. To associate timing information to milestones, the TU is configured never to slow down the traced application. In this mode, allowing full trace generation might result in unpredictable trace overflows, as discussed in Section 5.1. Instead, the refined MG is used to wake up the TU and tracer only when a milestone is reached, as depicted in Figure 5. In the considered example, the tracee is initially (Figure 5a) executing code within v_2. The TU is configured to remain silent; its address range filter registers (see Section 3) are set to detect the arrival of execution into the next milestone (v_3). When v_3 is reached, the TU emits trace activity towards the tracer (Figure 5b). The TU uses the MG to dynamically

(a) Initially, assume that tracee's program counter (PC) is inside v_2. The TU is programmed to monitor arrival at v_3. The TU is silent until then, and the tracer awaits a signal from the TU.



(b) As soon as the tracee starts executing instructions in v_3, the TU signals the tracer. The tracer reconfigures the TU to monitor the next milestones v_5 and v_7 during the blackout window.



(c) The TU reconfiguration is complete and the tracer is ready to wait for tracee's execution to enter either v_5 or v_7. By design, tracee's execution has not yet reached them.

Figure 5 Tracer-Tracee interaction for milestone detection and dynamic TU reconfiguration.

reconfigure the TU to detect the next milestones, in this case, v_5 and v_7. Upon completion of the latter operation, the tracer goes back to waiting for an event from the TU (Figure 5c). Whenever a control transfer between two milestones is observed, the tracer measures the time – in terms of elapsed clock cycles – for the transfer.

(B) Milestone Timeliness Information. Using the measured milestone-to-milestone time, timeliness information is added to the MG in two parts. (1) Each node in the MG is given a *tail* time; (2) each edge in the MG is given a *nominal* time.

Tail time: The tail time $T_t(\mathtt{MBB}_i)$ is the absolute time by which the target must hit $\mathtt{MBB}_i$ for the last time. This value is the maximum taken across all the timed runs on the given set of representative inputs – worst-case in isolation. The tail time can be understood as the WCET till a specific milestone. However, loops and alternative paths make the tail time insufficient to assess a broader set of timeliness properties beyond WCET enforcement. Consider the case where we want to detect timely progress via loop iterations. Even if each iteration of the loop takes longer than usual, the tracer cannot detect per-iteration slowdowns by only using the tail time. The nominal time is designed to overcome such a limitation.

Nominal time: Given an edge $(\mathtt{MBB}_i, \mathtt{MBB}_j) \in \mathcal{Q}$, the nominal time $T_n(\mathtt{MBB}_i, \mathtt{MBB}_j)$ is a reference time the application is expected to spend to transfer from $\mathtt{MBB}_i$ to $\mathtt{MBB}_j$. Once again, the maximum is taken across all the timed runs. Even if the target runs in isolation (all other PEs idle), fluctuations in the value of T_n can occur due to microarchitectural noise. If $(\mathtt{MBB}_i, \mathtt{MBB}j)$ is part of a loop, nominal time is effective in detecting slower-than-expected transfer between $\mathtt{MBB}_i$ and $\mathtt{MBB}_j$. Thus the nominal time offers finer timeliness checking per iteration.

5.6 Online Timely Progress Assessment (Step 5)

Once a TMG has been obtained, online TPA is possible, which is the focus of Step 5 in Figure 2 and described below. The TMG is passed to the tracer when the target is launched. The $\mathtt{MBB}_0$ that corresponds to the selected entry point for the target is programmed by the tracer on the TU. Live tracking of the application under analysis is performed by employing the same strategy described in Section 5.5 and illustrated in Figure 5.

At runtime, we track two times: (1) the *actual time* $\Theta(i)$ and (2) the *running nominal time* $N(i)$. Let $\texttt{MBB}_i$ be the i-th milestone for which a hit has been detected. $\Theta(i)$ is updated with the current time. Therefore, it tracks the time measured since $\texttt{MBB}_0$ was hit and until $\texttt{MBB}_i$ is reached. Conversely, $N(i)$ is updated as $N(i) = N(i-1) + T_n(\texttt{MBB}_{i-1}, \texttt{MBB}_i)$.

At this point, everything is set to assess the timely progress of the target. Whenever a milestone $\texttt{MBB}_i$ is hit, the tracer can check that $\Theta(i) \leq \min(T_t(\texttt{MBB}_i), N(i))$. If a controllable amount of degradation – compared to the reference timing acquired in isolation – is accepted, one can express the allowed slowdown as $\alpha > 1$ and check the following condition instead:

$$\Theta(i) \leq \alpha \min(T_t(\texttt{MBB}_i), N(i)). \tag{3}$$

Importantly, all the elements are in place not only for the detection of TPI violations but also to routinely report positive/negative current slack to the tracee PE. The slack at $\texttt{MBB}_i$ can be calculated as $slack(i) = \min(\alpha T_t(\texttt{MBB}_i), \alpha N(i)) - \Theta(i)$.

6 Use Cases for MB-TPA

We hereby provide three use-cases enabled by the ability of MB-TPA to provide runtime timely progress assessment as a system commodity.

(A) Strict WCET Enforcement. Previous work has provided a methodology based on code-level instrumentation to insert progress checkpoints (milestones in our notations) with the goal of enforcing a target WCET for a high-criticality task under analysis [36–38, 58]. The capabilities of MB-TPA seamlessly support one such use case. Consider a mixed-criticality system in which a critical task is scheduled exclusively on the main core, and low critical tasks are scheduled on other cores. Kritikakou et al. [37] have proved that the following regulation policy can guarantee the timeliness of the critical task[5]. Following their strategy, low-criticality tasks are suspended if a checkpoint is reached and the slack is not sufficient as indicated by the following condition:

$$\mathrm{RWCET}_{\mathrm{iso}}(x) + \mathrm{RWCET}_{\max} + t_{\mathrm{RT}} > D_c - ET(x),$$

where $\mathrm{RWCET}_{\mathrm{iso}}(x)$ is the remaining WCET (measured in isolation) from the arrival at watchpoint x until completion. In our MB-TPA, this is equivalent to $T_t(\texttt{MBB}_{exit}) - T_t(\texttt{MBB}_x)$. $\mathrm{RWCET}_{\max}$ is the WCET from watch-point x to the next watchpoint when other low critical tasks are present, which can be measured as $T_n(\texttt{MBB}_x, \texttt{MBB}_{x+1})$ according to Section 5.5 by adding interference. t_{RT} is the software interrupt overhead. Our MB-TPA does not use interrupts, but to remain safe, the delay in the milestone detection at the tracer must be considered. This term is evaluated in Section 8.1. D_c and $ET(x)$ are deadline and actual time at x. We refer to the latter as $\Theta(x)$. The required metrics for the regulation policy are offered by MB-TPA, thus our method can also achieve strict WCET enforcement.

(B) Progress-aware Profiling. In this use case, we demonstrate that it is possible to acquire application profiles about their interaction with the underlying hardware in a way that is progress aware. This can be done by performing online tracking according to what described in Section 5.6. In addition, the tracer is modified to interface with the performance monitoring

[5] Due to space constraint, the proof is omitted here. The work also includes a treatment to regulate loop components.

unit of the tracee. By doing so, it is possible to measure the progression of architectural events (e.g. cache misses, branch mispredictions, bus stalls) at the reached milestones. This allows precise attribution of exhibited behaviors to specific code paths inside the target. More importantly, it enables correlating slowdowns on specific milestones to root causes in terms of platform behavior. And therefore, to identify hardware bottlenecks on a per-code-path basis. We evaluate this use case in Section 8.2.

(C) Progress-aware Controlled Degradation. Lastly, we consider TPA-driven detection of TPI violations due to contention over shared memory resources and perform regulation of interfering PEs with the goal of tracking a degraded performance setpoint for the target.

In a nutshell, TPI violation is triggered if the target suffers a slowdown greater than a selected α factor. At runtime, if Equation 3 does not hold, the tracer sends a signal to the tracee to pause the activity of all the other PEs. After the interfering cores have been stopped, the target might recover some slack. Thus, it might be possible to resume the paused PEs. To decide when the interfering PEs should be resumed, we use an *aggressiveness* parameter $\beta \in [0, 1]$. Whenever $slack(i) > \beta\alpha N(i)$, the interfering PEs are resumed. As β decreases, the tracer resumes the co-runners as early as possible. When β increases, the tracer becomes more conservative. We evaluate this use case in Section 8.2.

7 System Instantiation and Implementation Details

We performed a full-stack implementation of the proposed MB-TPA. We name our proof-of-concept system instantiation Timely Progress Assessment with 0 Overhead (TPAw0v). TPAw0v was implemented on the ZCU102 development board featuring a Xilinx UltraScale+ MPSoC. The target platform comprises two CPU clusters: (1) the APU cluster with four ARM Cortex-A53 CPUs operating at 1.3GHz, used as the tracee; (2) the RPU cluster with two ARM Cortex-R5 CPUs operating at 600MHz, used to implement the tracer. Following the platform assumptions described in Section 4, the target platform features an ARM Coresight infrastructure commonly with tracing capability.

Figure 6 illustrates the trace data path. Each tracee CPU has a TU, namely an ARM Embedded Trace Macrocell (ETM) [10]. The ETMs produce trace data for the respective core. The ETMs are capable of filtering the trace data by comparing the `PC` against a set of 4 range-address filters. Each filter uses two registers (`TRCACVRn`) for the address range's upper and lower ends. Trace data packets are generated whenever the `PC` falls within any of the defined ranges.

The trace packets traverse multiple on-chip CoreSight components. The bare-metal drivers used by the tracer to manage all these components were written from scratch. In TPAw0v, the ETR is configured to asynchronously store trace packets to the RPU cluster's scratchpad (TCM), where a 2KB circular buffer is reserved. The TMG in binary format is also stored on the TCM. The tracer implements all the modes to carry out the full MB-TPA pipeline described in Section 5, including online tracking.

7.1 Constructing MB-TPA with ETM

To implement MB-TPA, the ETM is driven using a Finite State Machine (FSM) by the tracer and composed of three states (solid circles), two transition states (dashed circles), and several transitions as depicted in Figure 7. The controller starts in the *Inactive* state. This state is the only one in which reconfiguring the ETM (modifying the address filtering registers) is allowed, as the ETM is *idle*. Once reconfiguration is completed, the controller

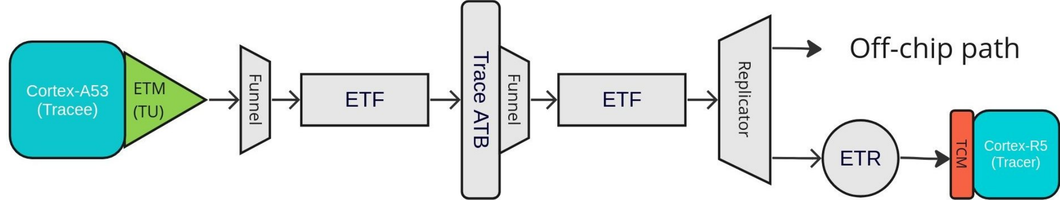


Figure 6 The Embedded Trace Macrocell (ETM) is the CPU-local device responsible for trace generation. The Trace Memory Controller [8] can be configured into an Embedded Trace FIFO (ETF) or Embedded Trace Router (ETR). The former serves as a buffer for the trace stream; the latter routes trace data to memory. ARM AMBA Advanced Trace Bus (ATB) [9] is adopted for trace data transmission. Funnels merge trace streams from potentially multiple ETMs and ATBs into a single ATB. The Replicator duplicates trace data from a single master to two slaves [12].

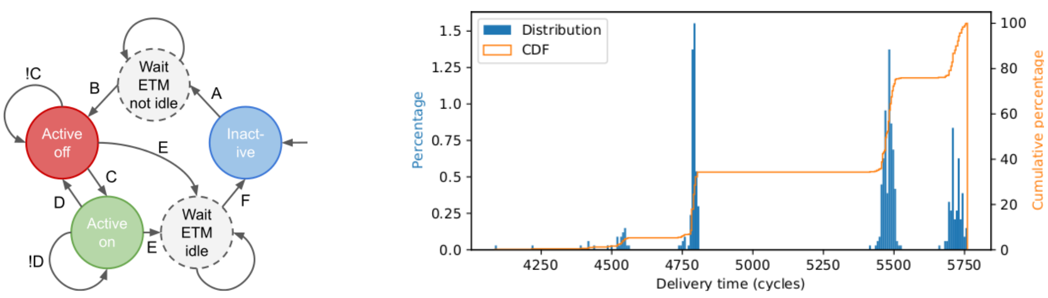


Figure 7 Tracer's controller as a finite state machine.

Figure 8 Delivery time (cumulative) distribution.

activates the ETM by asserting the `TRCPRGCTLR.EN` register (**A**), leading to a transition state to guarantee that the ETM is not idle. Here, the tracer waits for the `TRCSTATR.IDLE` register to be cleared before moving to the *Active-off* state (**B**). In *Active-off*, the ETM is monitoring the `PC`, but not generating informative packets[6], because the `PC` has not reached any addresses specified by the address filtering registers. I.e. the `PC` has not reached any milestones yet. When the `PC` reaches any of the specified addresses, Three packets are emitted in order by the ETM: a `synchronization`, a `trace-on`, and an `address` packet. This sequence signifies that a milestone was hit and the `address` packet includes the current value of the `PC`. Then, the controller moves to the *Active-on* state (**C**). Otherwise, the controller stays in Active-off (**!C**). Similar to its *"off"* counterpart, the Active-on state keeps the ETR actively waiting for the next packet (**!D**). Once the packet is finally captured, the controller (1) identifies the milestone hit via the `PC`, (2) computes the negative slack, and (3) propagates the latter to the tracee. The controller then moves back to the *Active-off* state (**D**). In both active states, the controller is allowed to request a change of address range to monitor. In such case, the ETM must be set to *idle* by clearing the `TRCPRGCTLR.EN` register (**D**). Then, the controller enters a transition state where it awaits for the `TRCSTATR.IDLE` register to be asserted, ensuring the ETM is *idle* (**E**).

[6] In *Active-off* state the ETM still generates `synchronization` and `address` packet pairs at a very low rate. These packet pairs can be ignored for our purpose.

8 Evaluation

First, we evaluate TPAw0v to understand its performance in terms of milestone detection delay, size of the trace blackout window, and overhead on the tracee. Next, we evaluate the ability to enact progress-aware profiling and controlled performance degradation.

8.1 Progress Assessment Performance

(A) Delivery Time. Let t denote the time at which tracee executes the first instruction in the monitored MBB. The TU generates a trace packet toward the tracer via on-chip buses. Let t' denote the time at which the tracer receives it. The delivery time $\Delta t = t' - t$ has to be comparably small so that the TPAw0v can operate effectively. To measure Δt, we use a synthetic benchmark on the tracee in which the cycle counter is periodically read. MBBs are chosen as the BBs where the cycles are sampled. The tracer reads the same cycle counter upon receiving the signal. For a given MBB, the application's timestamp is t; the tracer's is t'. We sample 1500 data points, 50% in isolation and the rest with interference from memory-intensive applications. Figure 8 shows the overall (cumulative) distribution. The delivery time is upper-bounded by 5750 cycles, or $4.4\mu s$ on our 1.3GHz tracee.

Recall that software-based detection methods [38, 58] inevitably introduce overhead due to synchronous interrupt handling. In contrast, our method never interrupts the tracee. Due to our monitoring scheme's asynchronous nature, the delivery time is not an overhead term. Nonetheless, it is informative to contrast the overhead for software-based detection to the magnitude of our delivery time. A convenient way to obtain such measurement is to use a widely-adopted Linux support for dynamic binary instrumentation, namely UPROBEs [3]. They allow hooks to be registered at different locations of a user application. A software interrupt is issued when a hook is reached and time can be sampled. We measured the overhead of UPROBEs at about $4\mu s$.

(B) Blackout Window Size. The reconfiguration of the TU is solely handled by the function `reconfigure` residing in the control logic of the tracer. Thus by reading the cycle counter before/after the function call of `reconfigure`, the size of T_r can be measured. We conduct such measurements while running TPAw0v normally with target applications from the SD-VBS suite [63] which is a diverse collection of computer vision applications. The characteristics of these benchmarks have been extensively studied by the community [51,52,61]. Our measurements show that T_r is around $3\mu s$. Recall that we choose T_r in terms of number of executed branch instructions. In the (very unlikely) worst case, all the instructions executed during the blackout window are branch instructions. Thus, we conservatively set $T_r = 10000$ given the 1.3GHz tracee.

(C) Overhead on Tracee. When the tracer only performs TPA but takes no regulation actions, the target should only experience a negligible slowdown. Five SD-VBS benchmarks were evaluated: `disparity`, `texture_synthesis`, `mser`, `tracking`, and `sift`.

We run benchmarks with their respective default inputs in two configurations: (1) without TPAw0v, and (2) with TPAw0v but taking no regulation actions. Ten runs are conducted per benchmark and in each configuration. The top section of Table 1 reports the slowdown caused by TPAw0v on the benchmarks as a percentage of their runtime. Expectedly, the overhead is low ($< 0.6\%$). The low yet visible overhead in some applications might arise from interference on the main interconnect between the tracer and the tracee CPUs. Implementing the tracer

Table 1 Overhead (%) of tracer activity and TMG/trace size information.

Benchmark	disparity	text.	mser	tracking	sift
Mean(%)	0.512	-0.009	0.250	-0.072	0.168
Max(%)	0.585	0.033	0.263	-0.110	0.194
Min(%)	0.483	0.085	0.225	-0.059	0.173
# of MBBs in TMG	17	5	18	16	13
# of MBB hit in execution	143	1169	20	18	19
# of unfolding functions	1	1	1	1	2
TMG size (bytes)	340	108	408	320	324
Raw trace size (MB)	10	44.4	14	175.2	236.4
Filtered trace size (bytes)	1500	9400	210	350	300

on the on-chip FPGA might mitigate the issue [64] and further reduce the overhead. Negative entries indicate that the applications run faster when traced. H. Shah et al. [56] observed and theorized such counterintuitive timing anomalies.

(D) Application Considerations. The sum of delivery time and blackout window size ($\sim 7.4\mu s$) indicates the responsiveness of the tracer in detecting and reacting to milestone hits. Thus, TPAw0v is better suited for applications with execution times on the order of $10^3\mu s$ and above, e.g., data processing workloads. Approaches using software interrupts would incur overheads of at least $4\mu s$, as measured on our platform. Thus, for short-lived applications, the overhead introduced by software instrumentation would significantly degrade performance.

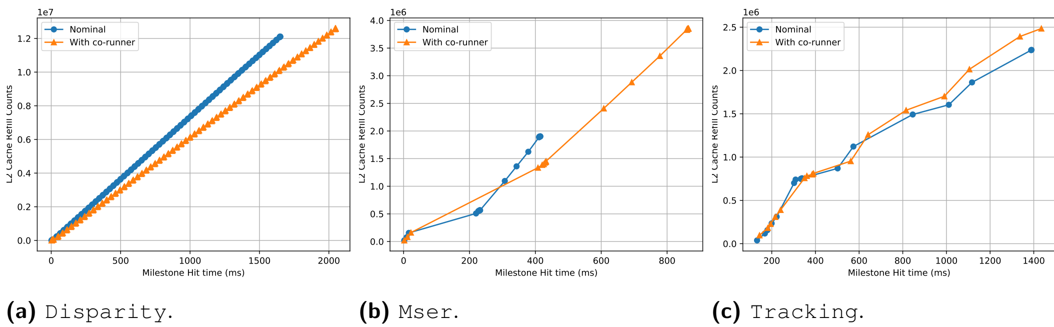
8.2 Evaluation of MB-TPA Use Cases

We hereby evaluate the last two use cases described in Section 6. For our evaluation, we consider the same five aforementioned SD-VBS benchmarks. The memory-intensive application `bandwidth` from IsolBench [62] is deployed on all the other cores to create interference in both main memory and shared cache.

(A) TMG Construction. First, we provide information regarding TMGs and trace data in the second section of Table 1. When a milestone is placed inside a loop, high granularity regulation can be achieved. `disparity` and `texture synthesis` demonstrate such granularity as the number of milestones hit is high. TMG size refers to the memory usage for the tracer to store the binary TMG; raw traces are only used during the offline MG refinement phase; the TU generates the filtered trace during online tracking.

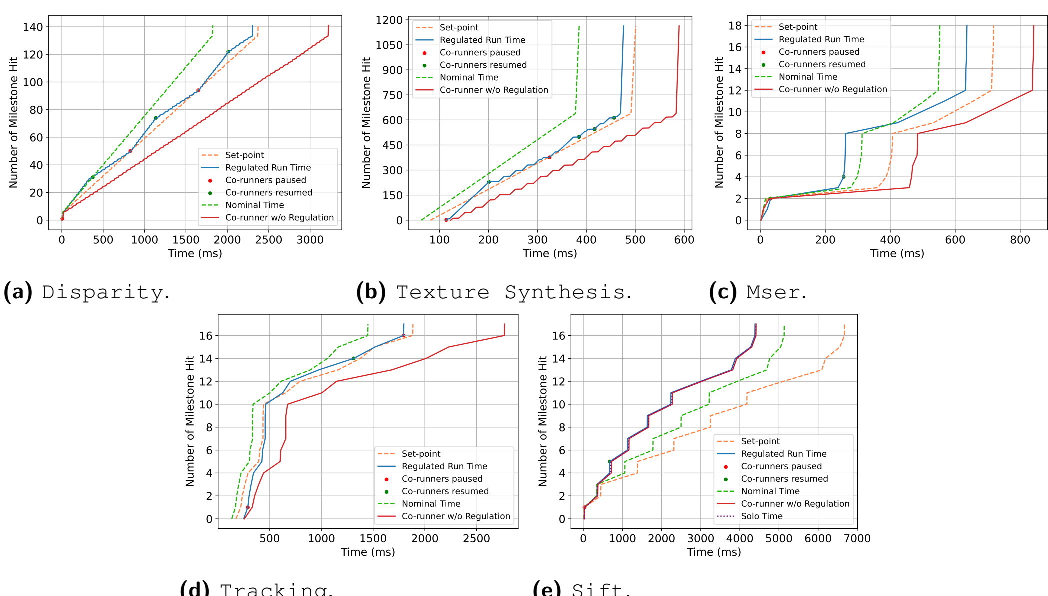
(B) Progress-aware Profiling. When the execution reaches a milestone, we collect architectural event statistics by directly reading the PMU event counters[7]. In this evaluation, the architectural event monitored is the L2 data cache refill, i.e. we track last-level cache misses. The benchmarks under evaluation run (1) in isolation and (2) with interference tasks. In each

[7] ETM can also report architectural events in the trace stream. ETM can optionally implement external inputs which connect to PMU event bus lines. Event packets can be inserted into the trace stream whenever the monitored events occur.



(a) `Disparity`. **(b)** `Mser`. **(c)** `Tracking`.

Figure 9 Relationship between timeliness (x), L2 cache misses (y), and milestones (markers).



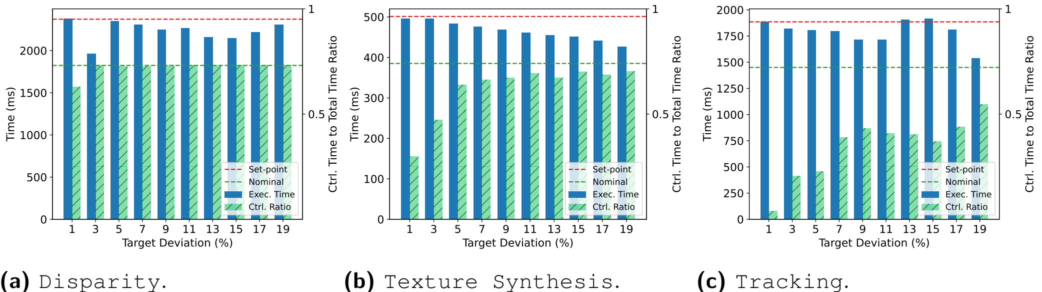
(a) `Disparity`. **(b)** `Texture Synthesis`. **(c)** `Mser`. **(d)** `Tracking`. **(e)** `Sift`.

Figure 10 The TMG for `disparity` and `texture synthesis` captures appropriate loops, achieving fine granularity. Despite a coarser control for `mser`, TPI is maintained.

case, the benchmark runs 20 times. The tracer reports the time and cache refill statistics at each milestone hit. The relationship between elapsed time (x-axis), cumulative number of L2 misses (y-axis), and milestones hit (markers) – and therefore segments of executed code – as captured for three SD-VBS applications is reported in Figure 9. The figure highlights that `disparity` and `tracking` suffer only marginally from cache contention, while five milestones in `mser` are significantly impacted by contention in L2.

The significance of relating the consumption of hardware resources to progress is twofold. First, resource management decisions can be enacted proactively as opposed to reactively. Second, by comparing the expected profile at a milestone to what is observed online, a system can identify the root causes of performance degradation and enact appropriate corrective actions. The combination of progress tracking and progress-aware resource management requires extensive research.

(C) Controlled Performance Degradation. In this scenario, we evaluate the ability to set a degraded performance setpoint for the application under analysis and stop/resume interfering cores based on the online slack calculation reported by the tracer. The behavior of



(a) `Disparity`. **(b)** `Texture Synthesis`. **(c)** `Tracking`.

Figure 11 As target deviation β increases, the tracer becomes more conservative, and only resumes the co-runner when a sufficient positive slack presents. Thus, the application follows the set-point more closely for small β.

the five SD-VBS benchmarks is reported in Figure 10. We compare the runtime under tracer-enforced regulation ("Regulated Run Time") with two other cases: (1) the nominal case, i.e., the worst-case progress in isolation, and (2) the progress under unregulated interference ("Co-runner w/o Regulation"). We use $\alpha = 1.3$ and $\beta = 7\%$; the resulting progress reference is labeled "Set point." The history of accessed milestones in chronological order is reported on the y-axis; the time elapsed between milestones is reported on the x-axis; the binary decisions to suspend (red dot) or resume (green dot) the co-runners are reported.

In all the cases, the tracer was able to enforce a controllably degraded notion of TPI for the target. Corrective measures are taken as soon as the detected progress falls below the reference. The specific value of β we considered works well in most cases but becomes overly conservative in the case of `mser`. In this case, preventing a slowdown in the early stages (at milestones 2–4) is sufficient to ensure that the setpoint is met for the rest of the run. The behavior of `sift` (Figure 10e) is interestingly different. The solo, uncontrolled, and controlled progress nearly coincide. This indicates that `sift` is unaffected by the interference tasks. The nominal progress, however, is slower than the above three. Recall that the nominal time for each edge is taken as the maximum transfer time across all runs. But in a single run, not all transfers take the worst-case time.

To better understand the impact of β on the behavior of the applications, we sweep through values of $\beta \in [1\%, \ldots, 19\%]$ and present the results in Figure 11. The "Exec Time" bar captures the runtime under contention and regulation. The "Ctrl. Ratio" bar reports the fraction of time during which the real-time is below the set-point. As β increases, TPAw0v becomes more conservative, and the aggressiveness of the regulation increases. `sift` is not included since it does not suffer from performance degradation.

9 Conclusion

Prompted by the demand for high-performance embedded platforms, the design of modern system-on-chip has gained in complexity at the expense of software predictability and timeliness. We argue that reasoning on the progress of live applications must be a key requirement to achieve *Timely Progress Integrity.* In this paper, we propose a method called MB-TPA and present a prototype, TPAw0v, feasible on widely available commercial platforms featuring tracing capabilities. Our experiments show that our prototype is successful in tracking the progress of applications under test with near-zero overhead while operating on a lower-performance core! Moreover, through its prototype implementation, we demonstrate

the capability of our model to detect execution anomalies and enforce corrective measures to preserve TPI. We envision that the contributions made by this work represent the first building blocks towards elaborated real-time policies with TPI at their core.

References

1 Powertrace iii. https://www.lauterbach.com/powertrace3.html. Accessed: 01-03-2023.
2 Technology overview. https://www.ghs.com/products/probe.html. Accessed: 01-03-2023.
3 Uprobe-tracer: Uprobe-based event tracing. https://docs.kernel.org/trace/uprobetracer.html.
4 Working draft of the risc-v processor trace specification. https://github.com/riscv-non-isa/riscv-trace-spec. Accessed: 01-03-2023.
5 Jaume Abella, Carles Hernandez, 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)*, pages 1–10, 2015. doi:10.1109/SIES.2015.7185039.
6 Seyed Mohammad Ali Zeinolabedin, Johannes Partzsch, and Christian Mayr. Analyzing arm coresight etmv4.x data trace stream with a real-time hardware accelerator. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1606–1609, 2021. doi:10.23919/DATE51398.2021.9474035.
7 ARM. Coresight components technical reference manual, 2004. URL: https://developer.arm.com/documentation/ddi0314/h/.
8 ARM. CoreSight trace memory controller technical reference manual, 2010. URL: https://developer.arm.com/documentation/ddi0461/b/.
9 ARM. AMBA ATB Protocol Specification, 2012. URL: https://developer.arm.com/documentation/ihi0032.
10 ARM. Embedded trace macrocell architecture specification etmv4.0 to etm4.6, 2012. URL: https://developer.arm.com/documentation/ihi0064/h/?lang=en.
11 ARM. Arm architecture reference manual for a-profile architecture, 2013. URL: https://developer.arm.com/documentation/ddi0487/latest.
12 ARM. ARM CoreSight SoC-400 Technical Reference Manual, 2015. URL: https://developer.arm.com/Processors/CoreSight%20SoC-400.
13 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 2012)*, pages 145–154, Los Alamitos, CA, USA, July 2012. IEEE Computer Society. doi:10.1109/ECRTS.2012.42.
14 S.K. Baruah, A. Burns, and R.I. Davis. Response-time analysis for mixed criticality systems. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 34–43, 2011. doi:10.1109/RTSS.2011.12.
15 Andrea Bastoni, Björn B. Brandenburg, and James H. Anderson. Cache-related preemption and migration delays : Empirical approximation and impact on schedulability. In *Proceedings of the 6th annual workshop on. Operating Systems Platforms for Embedded Real-Time Applications*, volume 10 of *OSPERT'10*, pages 33–44, 2010.
16 Emiliano Betti, Stanley Bak, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Real-time i/o management system with cots peripherals. *IEEE Transactions on Computers*, 62(1):45–58, 2013. doi:10.1109/TC.2011.202.
17 Reinder J. Bril, Sebastian Altmeyer, Martijn M. H. P. van den Heuvel, Robert I. Davis, and Moris Behnam. Fixed priority scheduling with pre-emption thresholds and cache-related pre-emption delays: integrated analysis and evaluation. *Real-Time Systems*, 53(4):403–466, July 2017. doi:10.1007/s11241-016-9266-z.

18 Alan Burns and Robert Ian Davis. *Mixed Criticality Systems – A Review (13th Edition, February 2022)*. Universities of Leeds, Sheffield and York, February 2022. URL: `https://eprints.whiterose.ac.uk/183619/`.

19 M. Caccamo, G. Buttazzo, and Lui Sha. Elastic feedback control. In *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000*, pages 121–128, 2000. `doi:10.1109/EMRTS.2000.853999`.

20 Hugues Cassé and Pascal Sainrat. OTAWA, a Framework for Experimenting WCET Computations. In *Conference ERTS'06*, Toulouse, France, January 2006. URL: `https://hal.science/hal-02270434`.

21 Intel Corp. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide, 2022. URL: `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`.

22 Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

23 Peter Dinges and Gul Agha. Targeted test input generation using symbolic-concrete backward execution. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 31–36, New York, NY, USA, 2014. Association for Computing Machinery. `doi:10.1145/2642937.2642951`.

24 Pontus Ekberg and Wang Yi. Outstanding paper award: Bounding and shaping the demand of mixed-criticality sporadic tasks. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 135–144, 2012. `doi:10.1109/ECRTS.2012.24`.

25 Lang Feng, Jeff Huang, Jiang Hu, and Abhijith Reddy. Fastcfi: Real-time control-flow integrity using fpga without code instrumentation. *ACM Trans. Des. Autom. Electron. Syst.*, 26(5), June 2021. `doi:10.1145/3458471`.

26 Robert Gifford, Neeraj Gandhi, Linh Thi Xuan Phan, and Andreas Haeberlen. DNA: Dynamic resource allocation for soft real-time multicore systems. In *27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '21)*, May 2021. `doi:10.1109/RTAS52030.2021.00024`.

27 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), November 2015. `doi:10.1145/2830555`.

28 Xiaozhe Gu and Arvind Easwaran. Dynamic budget management with service guarantees for mixed-criticality systems. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 47–56, 2016. `doi:10.1109/RTSS.2016.014`.

29 Xiaozhe Gu, Arvind Easwaran, Kieu-My Phan, and Insik Shin. Resource efficient isolation mechanisms in mixed-criticality scheduling. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 13–24, 2015. `doi:10.1109/ECRTS.2015.9`.

30 Jan Gustafsson. Usability aspects of WCET analysis. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 346–352, 2008. `doi:10.1109/ISORC.2008.55`.

31 Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. The Heptane Static Worst-Case Execution Time Estimation Tool. In Jan Reineke, editor, *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 57 of *OpenAccess Series in Informatics (OASIcs)*, pages 8:1–8:12, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/OASIcs.WCET.2017.8`.

32 Augusto Hoppe, Jürgen Becker, and Fernanda Lima Kastensmidt. High-speed hardware accelerator for trace decoding in real-time program monitoring. In *2021 IEEE 12th Latin America Symposium on Circuits and System (LASCAS)*, pages 1–4, 2021. `doi:10.1109/LASCAS51355.2021.9459137`.

33 Tai-Yi Huang, J.W.-S. Liu, and D. Hull. A method for bounding the effect of DMA I/O interference on program execution time. In *17th IEEE Real-Time Systems Symposium*, pages 275–285, 1996. `doi:10.1109/REAL.1996.563724`.

34 Marine Kadar, Gerhard Fohler, Don Kuzhiyelil, and Philipp Gorski. Safety-aware integration of hardware-assisted program tracing in mixed-criticality systems for security monitoring. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 292–305, 2021. doi:10.1109/RTAS52030.2021.00031.

35 Raimund Kirner and Peter P. Puschner. Discussion of misconceptions about WCET analysis. In Jan Gustafsson, editor, *Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis, WCET 2003 – A Satellite Event to ECRTS 2003, Polytechnic Institute of Porto, Portugal, July 1, 2003*, volume MDH-MRTC-116/2003-1-SE, pages 61–64. Department of Computer Science and Engineering, Mälardalen University, Box 883, 721 23 Västerås, Sweden, 2003.

36 Angeliki Kritikakou, Thibaut Marty, and Matthieu Roy. Dynascore: Dynamic software controller to increase resource utilization in mixed-critical systems. *ACM Trans. Des. Autom. Electron. Syst.*, 23(2), October 2017. doi:10.1145/3110222.

37 Angeliki Kritikakou, Claire Pagetti, Olivier Baldellon, Matthieu Roy, and Christine Rochange. Run-time control to increase task parallelism in mixed-critical systems. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 119–128, 2014. doi:10.1109/ECRTS.2014.14.

38 Angeliki Kritikakou, Christine Rochange, Madeleine Faugère, Claire Pagetti, Matthieu Roy, Sylvain Girbal, and Daniel Gracia Pérez. Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 139–148, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2659787.2659799.

39 Don Kuzhiyelil, Philipp Zieris, Marine Kadar, Sergey Tverdyshev, and Gerhard Fohler. Towards transparent control-flow integrity in safety-critical systems. In *International Conference on Information Security*, pages 290–311. Springer, 2020.

40 Chang-Gun Lee, Hoosun 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. doi:10.1109/12.689649.

41 Jaewoo Lee, Hoon Sung Chwa, Linh T. X. Phan, Insik Shin, and Insup Lee. Mc-adapt: Adaptive task dropping in mixed-criticality scheduling. *ACM Trans. Embed. Comput. Syst.*, 16(5s), September 2017. doi:10.1145/3126498.

42 Di Liu, Jelena Spasic, Nan Guan, Gang Chen, Songran Liu, Todor Stefanov, and Wang Yi. Edf-vd scheduling of mixed-criticality systems with degraded quality guarantees. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 35–46, 2016. doi:10.1109/RTSS.2016.013.

43 Daniel Lo, Mohamed Ismail, Tao Chen, and G. Edward Suh. Slack-aware opportunistic monitoring for real-time systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 203–214, 2014. doi:10.1109/RTAS.2014.6926003.

44 Chenyang Lu, John A. Stankovic, Sang H. Son, and Gang Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Systems*, 23(1):85–126, July 2002. doi:10.1023/A:1015398403337.

45 Tamara Lugo, Santiago Lozano, Javier Fernández, and Jesus Carretero. A survey of techniques for reducing interference in real-time applications on multicore platforms. *IEEE Access*, 10:21853–21882, 2022. doi:10.1109/ACCESS.2022.3151891.

46 Mingsong Lv, Zonghua Gu, Nan Guan, Qingxu Deng, and Ge Yu. Performance comparison of techniques on static path analysis of wcet. In *2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, volume 1, pages 104–111, 2008. doi:10.1109/EUC.2008.178.

47 Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. A survey on static cache analysis for real-time systems. *Leibniz Transactions on Embedded Systems*, 3(1):05:1–05:48, June 2016. doi:10.4230/LITES-v003-i001-a005.

48 C. Maiza, H. Rihani, J. Rivas, J. Goossens, S. Altmeyer, and R. Davis. A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. *ACM Comput. Surv.*, 52(3), June 2019. doi:10.1145/3323212.

49 R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54, 2013.

50 Hemendra Singh Negi, Tulika Mitra, and Abhik Roychoudhury. Accurate estimation of cache-related preemption delay. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '03, pages 201–206, New York, NY, USA, 2003. Association for Computing Machinery. doi:10.1145/944645.944698.

51 Mattia Nicolella, Denis Hoornaert, Shahin Roozkhosh, Andrea Bastoni, and Renato Mancuso. Know your enemy: Benchmarking and experimenting with insight as a goal. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, RTSS 2022, 2022. URL: https://cs-people.bu.edu/rmancuso/files/papers/RTBench_RTSS22.pdf.

52 Mattia Nicolella, Shahin Roozkhosh, Denis Hoornaert, Andrea Bastoni, and Renato Mancuso. Rt-bench: An extensible benchmark framework for the analysis and management of real-time applications. In *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, RTNS 2022, pages 184–195, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3534879.3534888.

53 Alessandro Papadopoulos, Enrico Bini, Sanjoy Baruah, and Alan Burns. Adaptmc: A control-theoretic approach for achieving resilience in mixed-criticality systems. In Sebastian Altmeyer, editor, *Proceeding ECRTS Conference*, pages 14:1–14:22, Dagstuhl, July 2018. LIPICS. URL: https://eprints.whiterose.ac.uk/133393/.

54 J. Ren and L. Xuan Phan. Mixed-criticality scheduling on multiprocessors using task grouping. In *2015 27th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 25–34, Los Alamitos, CA, USA, July 2015. IEEE Computer Society. doi:10.1109/ECRTS.2015.10.

55 Gero Schwaricke, Rohan Tabish, Rodolfo Pellizzoni, Renato Mancuso, Andrea Bastoni, Alexander Zuepke, and Marco Caccamo. A real-time virtio-based framework for predictable inter-vm communication. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 27–40, 2021. doi:10.1109/RTSS52674.2021.00015.

56 Hardik Shah, Kai Huang, and Alois Knoll. Timing anomalies in multi-core architectures due to the interference on the shared resources. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 708–713, 2014. doi:10.1109/ASPDAC.2014.6742973.

57 Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

58 Soham Sinha, Richard West, and Ahmad Golchin. Pastime: Progress-aware scheduling for time-critical computing. *arXiv preprint*, 2019. arXiv:1908.06211.

59 Parul Sohal, Rohan Tabish, Ulrich Drepper, and Renato Mancuso. E-warp: A system-wide framework for memory bandwidth profiling and management. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 345–357, 2020. doi:10.1109/RTSS49844.2020.00039.

60 J.A. Stankovic, Chenyang Lu, S.H. Son, and Gang Tao. The case for feedback control real-time scheduling. In *Proceedings of 11th Euromicro Conference on Real-Time Systems. Euromicro RTS'99*, pages 11–20, 1999. doi:10.1109/EMRTS.1999.777445.

61 Dharmesh Tarapore, Shahin Roozkhosh, Steven Brzozowski, and Renato Mancuso. Observing the invisible: Live cache inspection for high-performance embedded systems. *IEEE Transactions on Computers*, 71(3):559–572, 2022. doi:10.1109/TC.2021.3060650.

62 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)*, pages 1–12, 2016. doi:10.1109/RTAS.2016.7461361.

63 Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. SD-VBS: The san diego vision benchmark suite. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 55–64, 2009. doi:10.1109/IISWC.2009.5306794.

64 Xilinx. Zynq UltraScale+ Device Technical Reference Manual, 2023. URL: https://docs.xilinx.com/r/en-US/ug1085-zynq-ultrascale-trm/Components.

65 H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, 2014.

66 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 *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013. doi:10.1109/RTAS.2013.6531079.

67 Matteo Zini, Giorgiomaria Cicero, Daniel Casini, and Alessandro Biondi. Profiling and controlling I/O-related memory contention in COTS heterogeneous platforms. *Software: Practice and Experience*, 52(5):1095–1113, 2022. doi:10.1002/spe.3053.

Consensual Resilient Control: Stateless Recovery of Stateful Controllers

Aleksandar Matovic ✉
Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg

Rafal Graczyk ✉
Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg

Federico Lucchetti ✉
Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg

Marcus Völp ✉
Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg

Abstract

Safety-critical systems have to absorb accidental and malicious faults to obtain high mean-times-to-failures (MTTFs). Traditionally, this is achieved through re-execution or replication. However, both techniques come with significant overheads, in particular when cold-start effects are considered. Such effects occur after replicas resume from checkpoints or from their initial state. This work aims at improving on the performance of control-task replication by leveraging an inherent stability of many plants to tolerate occasional control-task deadline misses and suggests masking faults just with a detection quorum. To make this possible, we have to eliminate cold-start effects to allow replicas to rejuvenate during each control cycle. We do so, by systematically turning stateful controllers into instants that can be recovered in a stateless manner. We highlight the mechanisms behind this transformation, how it achieves consensual resilient control, and demonstrate on the example of an inverted pendulum how accidental and maliciously-induced faults can be absorbed, even if control tasks run in less predictable environments.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Embedded and cyber-physical systems; Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases resilience, control, replication

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.14

Funding This work is supported by the European Commission through H2020 grant 871259 – ADMORPH.

Acknowledgements Thanks to the anonymous reviewers and shepherd for their fruitful comments and suggestions how to improve this paper. A special thanks goes to Martina Maggio and to Filip Markovic for their helpful feedback and advice.

1 Introduction

Safety-critical systems used to be closed systems built from highly predictable components and with accidental fault tolerance obtained through triple-modular redundancy (TMR) [37] (e.g., in the time-triggered architecture (TTA) [31]). Although some systems continue to be built along these lines, real-time systems, in general, became more open, networked, functionally richer and dynamic and, as such, also more susceptible to accidental faults and targeted attacks. Cyberattacks are a reality for real-time as well as safety-critical systems [34, 11, 58, 69, 36, 70, 28, 56, 49, 53, 41].

© Aleksandar Matovic, Rafal Graczyk, Federico Lucchetti, and Marcus Völp;
licensed under Creative Commons License CC-BY 4.0
35th Euromicro Conference on Real-Time Systems (ECRTS 2023).
Editor: Alessandro V. Papadopoulos; Article No. 14; pp. 14:1–14:27
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Fault and intrusion detection, paired with a mechanism to recover and re-execute faulty tasks (see, e.g., Zou et al. [73]), as well as fault-masking through voting has been proposed as application-agnostic techniques to mitigate accidental and intentionally-induced malicious faults. However, these techniques come at high costs, in particular, due to cold-start effects when running recovered tasks from their initial state or from a checkpoint. As control systems become more complex, we observe recovery effects become more prominent. For example, stopping and restarting (from its initial state) the perception module of an autonomous driving stack may well lead to cold-start effects that require the vehicle to stop for several seconds before environmental perception gets restored[1].

In this paper, we address the performance problems of recovering tasks from a cold state to allow them to rejuvenate each time the control task is invoked. We utilize this possibility to rejuvenate to operate control with a quorum that is just large enough to detect faults. We then leverage recent results from Maggio et al. [39] and Vreman et al. [65], which state conditions under which a controlled system can tolerate missing up to m subsequent actuations, to reach consensus over time. More precisely, in case the detection quorum is not able to reach consensus immediately (which is the case if a fault manifests in a disagreement of votes), we rejuvenate and re-execute control task replicas in the subsequent control periods – which we call *epochs*. Rejuvenated tasks re-execute the original problem (i.e., sensor inputs and state) to collect over up to k epochs the matching proposals we need to reach consensus. We do so while ensuring k is bounded from above by the missable deadlines (i.e., $k \leq m$).

More precisely, Maggio et al. [39] identified an inherent stability of many plants that allows them to tolerate several deadline misses in a row without losing said stability, provided no wrong actuator signal reaches the plant. Vreman et al. [65], further found that an even larger number of deadline misses can be tolerated, provided the controller enters a subsequent no-miss phase in which deadlines can be guaranteed to be met. Whereas the first result allows operating the controller just with a detection quorum, reaching an agreement over time, the latter gives rise to adjust the system's resilience by switching from a detection to a masking quorum, by adjusting its resilience to adapt to more critical failures [57] or by engaging in more elaborate recovery actions.

The prerequisite for applying any of these techniques is the system's ability to recover faulty replicas extremely fast to allow rejuvenating them after each invocation. Naive recovery would require creating a new instance of the control task, bringing it up to speed with the state of its peers (e.g., by resuming it from a checkpoint and by replaying previous requests), and configuring its privileges to participate in the consensus decisions instead of the faulty task it replaces. The costs of these operations are high and challenging to bind from above. In other words, such a recovery method is not suitable to be applied in between any two invocations of the control task. Imagine instead the task would be stateless in the sense of observing all required information by reading out the plant's sensors. It would need to maintain no other state from one invocation to another. Then, rejuvenation would amount to a trivial reset of the task, its control flow and stack to the beginning of its control loop. Unfortunately, most control tasks are not stateless and even seemingly stateless control algorithms, such as the Linear Quadratic Regulator (LQR), may become stateful in case, not all values can be directly observed from the plant.

This paper demonstrates how stateful tasks can be systematically transformed into instances that can be recovered like stateless ones. This way, recovery becomes fast enough to be executed before every invocation. Moreover, we show how consensual memory [19] helps protect any state that needs to be maintained across control-task invocations.

[1] Observation from injecting crash faults into Apollo's perception system [14].

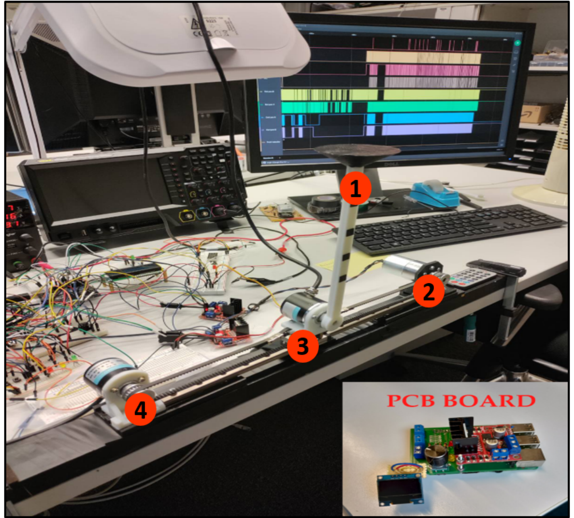
We make the following contributions:

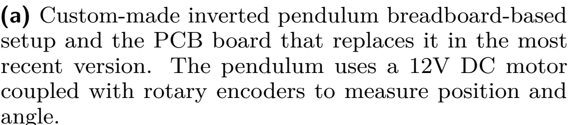
- We introduce in Section 5 our Consensual Resilient Control (CRC) approach, capable of masking up to f accidental faults every control period T (which we call *epoch*), including some maliciously-induced faults as detailed in our system and fault model in Section 4.
- To achieve this, we systematically convert stateful control tasks into statelessly recoverable instances (see Section 5.1), which we execute in a replicated manner, but just with a detection quorum of $n = f + 1$ replicas.
- We equip replicas with a means to revert to the current and previous plant state (see Section 5.2) and utilize the latter to reach consensus, in case this was not possible in a single epoch.
- We provide them with a trusted voter, which is simple enough to allow hardware or FPGA implementations, but more importantly, which can be brought to a zero defect target. The voter is used to actuate the plant after agreement has been reached, but also to store data in consensual memory if this information needs to be carried across control-task invocations (see Section 5.3). An FPGA implementation is not part of this work.
- And, we evaluate the performance of our approach against a non-resilient (singleton) version of the control task, as well as against a classical Triple Modular Redundancy (TMR) setup. To prevent adversaries from exhausting the healthy majority that TMR needs to mask faults, we consider also the proactive rejuvenation of replicas that participate in TMR (e.g., by operating with $n = 4$ replicas to tolerate one fault, even when one of the four replicas is rejuvenated). We restart this replica from its initial state.

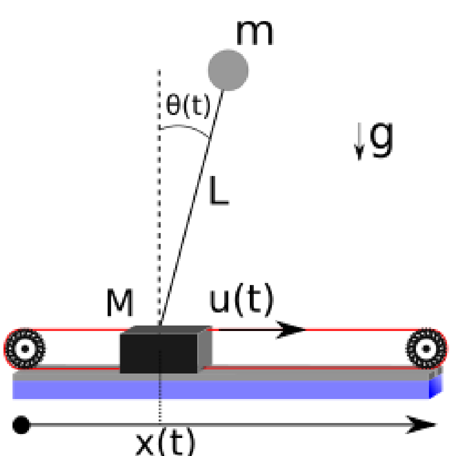
This paper is organized as follows: in Section 2, we present our evaluation vehicle and running example (an inverted pendulum). We continue by relating our work to the works of others (in Section 3) and by formally introducing our system and fault model (in Section 4). In this work, we focus exclusively on faults pertaining to the execution of control tasks. We assume (possibly fused) sensor data to be correct. In the future, we will then investigate the interplay of resilience mechanisms for internal faults with measures, such as Choi et al.[12], to cope with sensor failures.

2 Running Example: Inverted Pendulum

The inverted pendulum serves today as the text-book example in control-theory [3]. It lies at the heart of many control theory problems ranging from self-balancing hover-boards to stabilizing rocket propulsion systems. Indeed the inverted pendulum is investigated throughout the scientific literature as a minimum-viable benchmark to study a myriad of control problems present in neural-network based controllers [67], complex-simplex control systems [42], and works in the real-time systems domain [54]. By virtue of its simplicity, we chose the one-dimensional inverted pendulum model as the running example throughout this paper. We evaluated our Consensual Resilient Control approach by using a custom-made design (shown in Figure 1(a)). The pendulum consists of a moving mass (M) articulating another mass (m) through a free falling rigid rod (of length L) along a one-dimensional axis. In this work, we shall use the pendulum as an example to illustrate how our novel CRC approach can mask faults in stateful controllers with just $n = f + 1$ replicas, which normally allow only detecting the presence of a wrong actuation. For that, we restrict ourselves to the linear regime where the inverted pendulum (in the literature referred to as *the plant*) state $h(t)$ is proximal at any moment t to its stable point i.e. $h(t) = [x(t), \dot{x}(t), \theta(t), \dot{\theta}(t)] \rightarrow h_s = [x_0, 0, 0, 0]$. Here x is the position of the pendulum along the one-dimensional axis, $\dot{x}$ its linear velocity, θ




(a) Custom-made inverted pendulum breadboard-based setup and the PCB board that replaces it in the most recent version. The pendulum uses a 12V DC motor coupled with rotary encoders to measure position and angle.



(b) Pendulum schematics and parameters that govern the pendulum's equations of motion.

Figure 1 Experimental setup of our inverted pendulum. A motor (2) provides a force u via a belt on a rail-sliding mass M whose position along the x axis is determined by a rotatory encoder (4). A secondary rotatory encoder (3) measures the angle θ of the rod (1) which has a length L.

its time-varying angle with-respect to the vertical axis and $\dot{\theta}$ the associated angular speed. In this stability region and in the presence of a feedback control $u(t)$ the fully non-linear equation of motion approximately linearize and are given by $\ddot{\theta} = \frac{1}{L}\big(g\cos\theta(t) - u(t)\sin\theta(t)\big)$. The task of every feedback control system is to keep the state of the plant close to the stable point by first sensing its current state and subsequently imparting a counter balancing force u, in our case to the sliding mass M. Various control algorithms exist, the most common ones used in order to stabilize the inverted pendulum are the Linear-Quadratic-Regulator (LQR) and the Proportional–Integral–Derivative controller (PID).

LQR implements the control task by feeding back a force which is proportional to the error of the current state with respect to the stable point such that $u_k = -\mathbf{K} \cdot \delta h_k$, where $\delta h = h_k - h_s$ and $\mathbf{K}$ is a matrix of constant weights that are fine-tuned as a function of the plant's dynamical properties.

PID takes a similar approach by adding two additional terms to the proportional term of LQR $u_k = \mathbf{K}_p \cdot \delta h_k + \mathbf{K}_i \cdot \sum_{m=k-l}^{k} \delta h_m + \mathbf{K}_d \cdot \frac{d\delta h_k}{dk}$ where the integral term accumulates the $k - m$ historic errors and the derivative term determines how fast the stable point is reached[2]. One striking difference between PID and LQR is that the former needs to keep track of the historic states in order to compute the integral term and is therefore referred to as a stateful controller as opposed to LQR being a stateless controller. However, given the technical specifications of our measurement devices[3], instantaneous measures such as

[2] We refer the reader to the corresponding control literature for further information on how to tune the PID gains and K matrix for LQR.

[3] Position and angle are sensed through two rotary encoders, which detect changes of the encoders' rotation angle as quadratically shifted square waves in two channels. That is, angular changes are reported as raising and falling edges of the square waves, whereby the shift between channels indicates the direction of rotation.

linear and angular speed are not immediately available for a given epoch but have to be indirectly inferred through first recording past positions and angles and then computing the temporal variation of the latter. Consequently, LQR becomes effectively stateful. Formally for a given epoch k the forward function that models the control feedback loop can be written as $f(h_{k-l}, ..., h_k) = u_k$ where for LQR $l = 1$ and PID $l \geq 1$. The necessary state that the controller needs to keep track of (h) would allow us to turn an effectively stateful controller into a stateless recoverable instance (see Section 5.1).

For PID and LQR of an inverted pendulum, this state is trivially small for modern computational devices as just a couple of variables are saved across invocations. However, for this work, the actual algorithms are not relevant. We must observe that over the years, several increasingly sophisticated control algorithms have been proposed to cope with increasing plant complexities, including Model Predictive Controllers (MPC). One glaring example is the electric microgrid, as exemplified by Huo et al. [27] where MPC optimizes the energy generation and storage decisions based on a state as large as 420KB (at each step).

On the other hand, there is a theoretical possibility to optimize an MPC algorithm implementation on a 43KB-limited microcontroller [40], using techniques that enable satisfactory control performance while respecting memory constraints.

3 Related Work

To the best knowledge of the authors, this is the first work to leverage application-specific knowledge to optimize the more general resilience problem of tolerating up to f simultaneous faults over extended periods of time (from at least $n = 2f + 1$ replicas – plus potentially additional replicas to compensate for unavailable ones during rejuvenation – to $n = f + 1$ replicas). Several works contribute as individual building blocks for this work, which we review in the following.

Hard and weakly hard real-time systems. Traditionally hard-real time systems consider deadline misses fatal as they put safety at risk. However, this is not generally true. Weakly-hard real-time systems [7] characterize systems by the number of deadlines that can be missed during any given time window. The $m - K$ model[4] [2, 18, 23] allows up to m deadlines to be missed among any K consecutive jobs of the task. In control, the parameters m and K can be derived when analyzing the inherent stability of plants [39, 65]. In this work, we leverage these results to operate under a detection quorum, respectively, in reduced tolerance settings until the system can be adapted to mask faults immediately.

Fault tolerance and recovery. The Time-Triggered Architecture (TTA) [31] is among the most advanced and elaborate bodies of work developed to tolerate faults in safety-critical systems. TTA ensures message exchange in non-overlapping message slots and provides membership, fault tolerance, and actuation voting by leveraging apriori knowledge about the messages that replicas should send in the individual slots. TTA and its time partitioning are required in many standards, including ISO 26262 (automotive), IEC 61508 (industrial control), and DO-178C (avionics), and adopted by prominent industrial players, such as Audi, Volkswagen, and Honeywell [51]. The fault tolerance layer [5] is based on cold restart from the ground state (g-state) or history states (h-state). TTA is often used in conjunction with other methods for fault tolerance, such as redundancy, re-execution, and self-healing, to

[4] Not to confuse our K gain matrix of the LQR controller with a $m - K$ model from the control literature.

build robust and reliable real-time systems. Our CRC differs in that it ensures that both recovery states, critical for preserving data integrity and consistency, are always accessible to applications, even in case of a system failure or unexpected interruption.

One of the most popular methods used in fault tolerance is TMR [37], an instance of active replication. Its purpose is to improve the reliability of a system by using three (or, in general, n) functionally equivalent components to perform the same function, with the system's output being the majority vote of the three (n). TMR is commonly used in safety-critical systems, such as aerospace [68], nuclear power plants [66], and medical devices [17], where a single failure could result in severe harm or damage. The idea behind TMR is that the probability of all three components failing simultaneously is extremely low, so the system is highly reliable. While effective in improving system reliability, there are some downsides such as increased cost, power consumption, and complexity, which may make it less practical for some applications. Our CRC is an instance of active replication. It reduces the costs of the system by reducing the number of nodes from $n = 2f + 1$ to $n = f + 1$. One of the prerequisites for this reduction is the possibility of utilizing shared memory.

In addition to active replication, spare replicas can be kept hot, warm or cold, which translates to different response times for taking over after a fault is detected and the spare activated. Our approach applies active replication. Both active and passive replication will, over time, exhaust the majority of healthy replicas [59], in particular when cyberattacks persist. Our approach specifically address this concern by providing an extremely fast rejuvenation scheme for control-task replicas. Rejuvenating active replicas normally requires a cold restart of the faulty replica and additional replicas to compensate this downtime. Our approach avoids both.

Re-execution [72, 25] is a fault recovery technique used to improve the reliability of tasks by executing them multiple times and by selecting the correct output from multiple executions. It uses slack time on the processor to detect faults locally at the end of task execution and re-execute the task when a fault is detected. A faulty task can either be re-executed from the start or restored from the most recent checkpoint before the fault occurred [48]. Our CRC approach extends this technique by transforming control tasks to always maintain a known healthy saved state from which we restart tasks. This is achieved by not only voting on the result, but also on the state that must be maintained.

Other recent works in the intersection of fault tolerance and real-time systems include[32, 46, 10, 13, 1, 21].

Shared State. Replicated systems are typically constructed to avoid shared state due to the vulnerabilities entailed with this state failing. However, since recent microcontroller product families for safety-critical systems [61] offer ECC and RAID-protected shared memory [24, 64], we will leverage such memory to allow control replicas to maintain state across epochs. In particular, we will turn this memory into consensual memory, as exemplified by Gouveia et al. [19]. Read-only shared memory is commonly used in hypervisor-based systems to isolate VMs [43] (e.g., when deduplicating pages in their memory image). Our solution works in a hypervisor or RTOS setup, but equally well also on a bare-metal configuration where replicas receive read-only access to their shared memory. Read-only access suffices because, as we shall see, updates are performed consensually through a voter.

ECC embeds the possibility to tolerate faults without exposing them to the application and its state. It encodes values (e.g., into a hamming code) to tolerate a certain number of bit flips by correcting them when reconstructing the original value. The same coding can further be used to detect additional bit flips. As bit flips accumulate over time when

unhandled, ECC should be frequently be overwritten with a technique called scrubbing to restore its tolerance capabilities. Scrubbing overwrites the memory with the same value to restore non-stuck bits to their correct encoding of the value, which allows tolerating the original number of bit flips minus those that got stuck. We shall use ECC memory to protect state in consensual memory.

Checkpoint Recovery. Macroscopically, our approach could be characterized as a checkpoint recovery scheme albeit with an ultra-high checkpoint frequency and unconditional recovery at the end of each epoch. However, when we compare to other checkpointing approaches, such as [47, 71, 4], in more detail, there are substantial differences, which we characterize in the following:

- Checkpoints typically capture the entire state of a task in a consistent cut across all replicas (updating the previous checkpoint with what has been modified since then), whereas in our approach and using consensual memory, replicas propose only very selectively what portion of that state they will need for future epochs. The remaining (writable) state is simply discarded.
- Computation of and agreement on a new checkpoint are typically separate operations. Our approach combines both by only changing the shared state after $f+1$ healthy replicas agree on the update.
- Checkpoints are typically computed asynchronously to the execution of checkpointed tasks (e.g., by marking modified pages as copy on write to create a consistent cut). This is not necessary, since replicas end their activity in an epoch by proposing both what should be updated in the shared state and how to actuate the plant. This further ensures that the agreed-upon state corresponds to the latest plant actuation, since agreement is reached on both simultaneously and the voter follows suite in applying the updates.
- Recovery from a checkpoint is typically performed only after replicas have failed. This is not sufficient as compromised replicas might remain stealthy and go undetected. For this reason, we recover replicas after every epoch, by resetting them to the beginning of their control loop.
- Last but not least, checkpointing diversified replicas requires determining whether the individual checkpoints denote the same progress, whereas agreeing just on the values to be carried across control epochs, avoids such complications, because (i) the agreed upon state needs not to be diversified (it can only be written consensually), and (ii) healthy replicas agree on the same updates, despite computing them in a sufficiently different manner.

CPS Attacks. Various studies have investigated attacks on CPSs, including sensor [60], GPS spoofing [63, 26], and AI-related attacks [20]. In this work, we focus exclusively on tolerating cyberattacks that may be successful in compromising up to f control replicas while assuming adversaries cannot exceed that bound faster than T_a. Rejuvenating all replicas faster than T_a allows us to tolerate such adversarial attacks over extended periods of time [59].

4 System and Fault Model

System Model. This work concerns the fault tolerant control of a plant by means of replicating its control task across n nodes (see Figure 2). We assume nodes fail independently, but are sufficiently closely coupled to access a voter through the IO channels it offers and to access shared ECC and possibly RAID protected memory. These can be cores of a multi- or

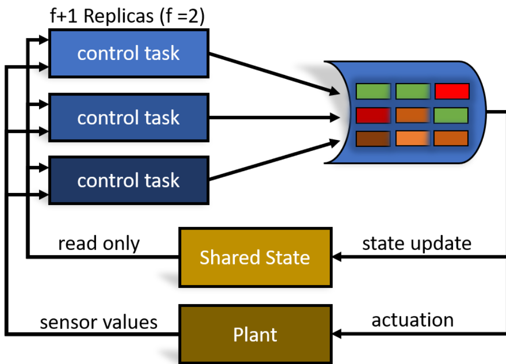


Figure 2 Replicated control architecture. Control task replicas sense the plant and have read-only access to shared state. They propose an actuation signal and state update, which the voter applies after reaching consensus.

many-core system (e.g., controlling a drone), multi-chip modules or more loosely coupled, but close compute nodes. If cached, the minimum requirement for the shared ECC memory is to invalidate cachelines upon writes, which as we shall see are updated exclusively by the voter.

A minimal control task senses the state of the plant, executes a control algorithm and proposes a signal for actuation. However, control tasks may also be more complex (e.g., structured as a directed acyclic graph of runnables) and involve filters, sensor aggregation and models of the plan to compute hidden state. Our goal is to support plants that are unaware of their controller's replicated nature. As such, we introduce a voter, which is trusted to consolidate control-task proposals into a singleton actuation signal. We shall also use the voter to update shared memory after reaching a consensus on how this state should be updated.

For simplicity, we assume a single control task having a single period is responsible for actuating the plant. Replicas of that task are invoked periodically every T time units and receive a consistent view of the plant state as far as this is observable through the plant's sensors. Supporting multi-periodic tasks (see e.g., Pagetti et al. [45]) is trivial as long as actuations are independent one from another. By replicating the multi-periodic control tasks individually and by introducing additional voters for each such group of replicas, one can achieve the desired actuation rates in the absence of errors. However, under faults, plants will have to tolerate missing some actuations for a number of epochs while others keep arriving. A discussion of multi-periodic control tasks with dependent actuations, where related actuations must not be passed to the plant if any of them cannot be performed at a given time, is out of the scope of this paper. We return in Section 5.2 to demonstrate how periodic activation and consistent sensing can be achieved using a trusted real-time operating system (RTOS) but also on bare metal. We call the T-distant invocations *control epochs*. Being invoked synchronously every T with a consistent view implies we operate under the assumptions of a synchronous system model.

Our approach is parametric in the number of faults f it can tolerate in an epoch (see our Fault model below for details) and in the number of epochs k by which we guarantee it to reach consensus. The parameters f and k determine how many replicas are required. Maggio et al. [39] found that many plants tolerate missing deadlines. The parameter k is bounded from above by this number m of deadlines that can be missed. Some plants, such as electric

steering can miss up to $m = 17$ deadlines. Our goal is to leverage this possibility to miss deadlines to optimize the system by reducing the number of replicas n required. Immediate masking (i.e., $k = 1$) within a single epoch (e.g., TMR) requires $n \geq 2f + 1$ replicas. Our goal is to reduce this number to as low as just $n = f + 1$ replicas, which can only detect faulty invocations that manifest in deviating proposals passed to the voter. With $n = f + 1$ replicas, at least one replica is guaranteed to make a healthy proposal. More generally, we are guaranteed to receive $n - f$ such healthy proposals during each epoch. To reach consensus, we have to collect $f + 1$ matching proposals including from at least one healthy replica. Our fault model rules out reaching such a match without healthy replica. But with only $n - f$ healthy proposals in each epoch, we are sure to collect the $f + 1$ matching proposals only after $\left\lceil \frac{f+1}{n-f} \right\rceil$ epochs, which bounds k from below. TMR typically operates under the assumption of $f = 1$, but there are systems deployed (e.g., for energy-grid safety), which have to tolerate $f = 5$ or even more faults simultaneously.

Fault Model. As mentioned above, we aim to protect against both accidental faults and intentionally induced, malicious faults (e.g., from cyberattacks). We shall therefore discuss several classes of faults and how we represent them in our abstract fault model, which is a slightly extended variant of the standard fault model for persistent and repeatedly partially-successful cyberattacks originally introduced by Sousa et al. [59].

Our fault model and hence the system we propose is parametric in the number of simultaneously occuring faults f that it can tolerate as well as in a few parameters ($T^f_{fault-type}$) which depend on the type of fault and which characterize when faults of that type may reoccur. The combination of number of faults and time of re-occurrence is quite standard, even in real-time systems. For example, the fault-tolerant time-triggered protocol by Kopetz and Grundsteidl [30] already assumed such a model. TTP can tolerate one fault among four synchronization replicas, provided the fault will not re-appear in the immediately following synchronization interval.

These parameters f, k and $T^f_{fault-type}$ are related to the configuration of the system in terms of the number of control-task replicas n that need to be deployed to tolerate this number of faults, the time $T_{rejuvenate}$ to rejuvenate replicas and in the time T_{agree} until agreement must be reached. The latter is further constrained by the number of subsequent deadline misses m that the plant can tolerate.

Deploying a system in an environment where these constraints cannot be guaranteed (e.g., because more than f faults of a class occur faster than $T^f_{fault-type}$) constitutes a failure in using the system. Fault tolerance and in consequence safety are no longer guaranteed once the thresholds are exceeded.

In terms of accidental faults, we consider transient and correctable faults that manifest in all parts of the system state, including in memory and in the internal and architecturally visible registers of the CPU. Such faults include bit flips due to radiation, charge deposited in flash memory cells, etc. We assume the RTOS frequently corrects such faults (e.g., by overwriting registers with the correct value or by scrubbing ECC memory). In particular we shall establish consensually-updated memory in ECC and possibly even RAID-protected memory. This memory will be shared between replicas and must return the latest written values, even in the presence of faults. If, on the other hand, bit flips occur only in a replica's memory and in not more than f over a period $T^f_{accidental}$, leaving our consensual memory unaffected, our system can handle these faults by collecting proposals from other replicas and by rejuvenating the faulty replica.

Accidental faults typically follow well understood characteristics from which a mean-time-to-failure (MTTF) can be derived and hence a high probability that faults will not reoccur within a certain time period, which for accidental faults we call $T^f_{accidental}$.

Faults of this nature often arise from external factors like radiation or fluctuations in temperature and are generally considered to be random events. To model these faults, stochastic processes, such as Poisson processes, are commonly employed. These models can help estimate the likelihood of faults happening within a designated time frame [38].

In the case of alpha particles hitting memory, this phenomenon is known as soft errors or single-event upsets (SEUs). Soft errors are transient faults that do not cause permanent damage to the system. They occur when high-energy particles, such as alpha particles or cosmic rays, strike the sensitive regions of semiconductor devices, causing bit flips in memory cells [6].

Malicious faults result from an attacker redirecting the control flow and altering the task's state to serve its purposes. Notice that techniques, such as return-oriented programming [9, 50], allow deviating from the task's intended control flow without modifying its code. This can happen, for example, by exploiting vulnerabilities such as buffer overflows to push return addresses that redirect the control flow to snippets of the task's code that, when combined, implement the adversary's desire. We assume a strong adversary capable of identifying, reaching, and exploiting such a vulnerability in control replicas.

Obviously, with identical replicas, no bound on the simultaneously affected replicas can be guaranteed. Instead, replicas need to be sufficiently diverse such that an attack applied to one replica cannot just be applied to another replica. Over time, adversaries may find vulnerabilities in more than f replicas by analyzing their current state (e.g., how its address space is randomized) and adjusting their exploit to the replicas state. This leads to two durations, which characterize the adversary. The time T^f_{deploy} required to deploy an attack and compromise a replica in the desired way, and the time T^f_{exceed} by which the adversary has analyzed more than f replicas. In this work, we allow T^f_{deploy} to become small (see below). However, we shall assume, as recommended by Sousa [59], that the RTOS diversifies all n replicas as part of the rejuvenation process faster than T^f_{exceed} (e.g., by re-randomizing their address space layout [8, 15] or by applying other diversification techniques [16, 35, 52]). Notice also that fault statistics do not apply to malicious faults.

We shall not further discuss diversification in this paper, as this needs to be applied at a different time scale[5], but they can easily be merged with the rejuvenation process we will introduce in Section 5.1 by not returning to the original binary's control loop and instead first activating a transition control loop after adjusting the address space of the task and then to the diversified version's control loop. See Section 5.4 for further details.

Rejuvenating replicas before each epoch to address faults, our approach can tolerate accidental and malicious faults, provided (i) the overall number of accidental and malicious faults will not exceed f and provided (ii) no more than f faults happen within any sliding window of length kT. The second condition holds if $T^f_{accidental} > kT$ and if $T^f_{deploy} > kT$. Of course, mean-time-to-failure is a value derived from fault statistics, which means that with a certain probability accidental faults can occur more frequently.

We shall not make such assumptions about accidental faults, but support more frequent occurrence of accidental faults, by leveraging another characteristics of such faults, namely that it is highly unlikely that two faults in two replicas will result in identical proposals. We

[5] T^f_{exceed} well exceeds the m epochs by which agreement needs to be reached.

will further reduce the likelihood of this happening by requiring replicas to solve a challenge for their proposal to be considered. Notice that this also addresses persistent faults, since a replica experiencing such a fault is unlikely to solve the challenge. Of course, persistent faults must be properly attributed in the overall number of faults and must be addressed by replacing the affected replica, which is out of the scope of this work. We shall return to this in Section 5.4.

It is also highly unlikely that adversaries will be able to predict accidental-fault intrinsics of a replica that still is able to solve the challenge, such that it can predict what that replica will propose. For application scenarios which can tolerate a low residual likelihood that the system becomes unsafe in case such a combination of events happen, we can therefore also deploy our solution in environments where up to f faults happen in each of the k epochs and where from the fk total faults, at most f are maliciously induced.

While replicas may fail as described above, we shall assume that the voter, the RTOS and the system clock will not fail in a similar manner. We assume they remain correct even in the presence of accidental faults and cyberattacks. For the voter, this assumption is justified by its simplicity, which allows implementing it entirely as custom logic in silicon or on an FPGA. Implementing the RTOS itself in a fault tolerant manner [55, 19] allows lifting the second assumption. Such a fault tolerant RTOS may then consensually update the system clock to remain in synchrony with other nodes in the system. Our solution is not resilient to physical attacks.

5 Consensual Resilient Control

In this section, we present our approach to consensual resilient control to tolerate up to f faults with just a detection quorum of $n \geq f + 1$ replicas. For now, let $n = k = f + 1$ and $f = 1$. That is, n replicas are periodically invoked with a consistent view of the plant state and are expected to produce an actuation signal, which they pass to the voter, which actuates the plant only after $f + 1$ replicas agree to the actuation value. In addition, to allow extremely fast recovery from faults and to make it possible to rejuvenate replicas in between any two subsequent invocations, they also vote on the state they would like to preserve across epochs.

Figure 3 shows for $f = 1$ and $n = k = 2$ how such a majority for the state update and actuation signal can be formed. In the first epoch, no faults happen, and the two replicas propose the same state update and actuation signal, which the voter applies since the $f + 1$ agreement has been reached. Even though replicas were correct, they are proactively rejuvenated to also return compromised but stealthy replicas to a known good state. In epoch 2, replica R_1 becomes faulty (either due to an attack or accidentally) and proposes an actuation value a'_2 instead. Without further knowledge about the plant, the voter cannot discern which of the two proposed actuation signals is correct and will therefore not actuate (while possibly holding the previous actuation value a_1 if the plant requires that). It will also not update the state, even though the replicas agree on this part of the proposal. This is to avoid inconsistencies between the plant and the state maintained by the replicas. As for now, the plant is not actuated, and we experience a deadline miss, which, since we so far missed less than k deadlines, we assume the plant tolerates. After rejuvenating the replicas, the replicas start. However, this time, no agreement could be reached in the previous epoch, with the sensor information captured at the beginning of epoch 2. This time replica R_2 fails in epoch 3. If the previous fault of R_1 was due to a cyberattack, R_2 could fail only due to accidental causes because we assume adversaries cannot compromise more than f

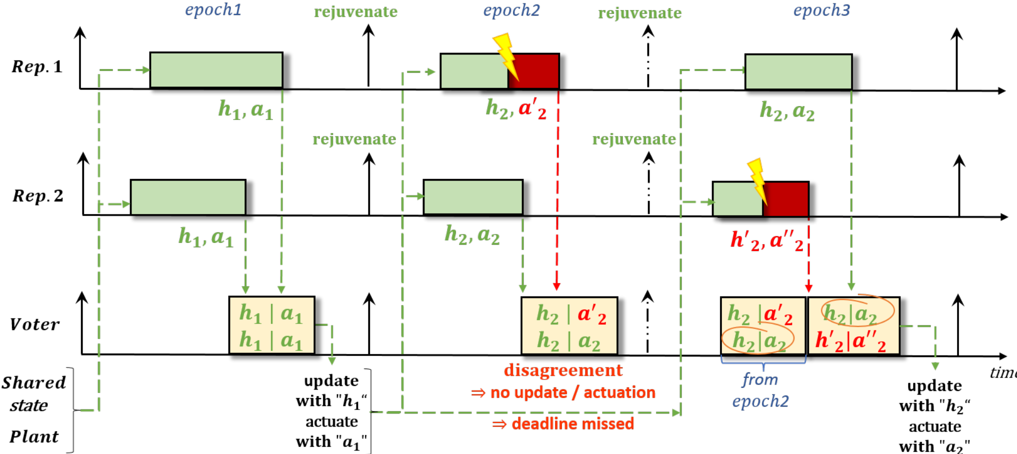


Figure 3 Example illustrating how $f+1$ agreement can be achieved despite replicas failing. Shown is a scenario with $f=1$ and $n=k=f+1=2$ over three epochs. In the first, correct replicas agree. In the second epoch, no agreement can be reached due to replica R_1 failing. In epoch 3, the voter is able to collect $f+1$ matching proposals after R_1 rejuvenates, even if this time R_2 fails.

replicas faster than the duration of k epochs. Also, by our fault model, the proposal of such an accidentally failing replica will not match the proposal R_1 made during epoch 2. If R_1 fails accidentally, adversaries are unlikely to predict how the failure will manifest. In both cases, the proposal from R_1 in epoch 2 and R_2 in epoch 3 will not already form a majority. However, after $k=2$ epochs, the voter collected two votes from correct replicas (from R_2 in epoch 2 and R_1 in epoch 3). Finally, the voter is able to actuate again (with a_2) and update the state (with h_2).

Moreover, operating the system with more than $n=f+1$ replicas is possible. In this case, $n-f$ replicas are correct by our fault model, and the voter can collect $n-f$ correct proposals in each epoch. Therefore, the number of epochs k needed before $f+1$ agreement can be reached is $k=\left\lceil\frac{f+1}{n-f}\right\rceil$. As long as a plant can tolerate at least k deadline misses, n and k will be a correct configuration to tolerate up to f faults for that plant. An important prerequisite for this approach to work is that replicas can be recovered fast enough from faults and rejuvenated between any two invocations.

In the following, we shall therefore discuss how to systematically turn stateful control tasks into statelessly-recoverable instants (Section 5.1), how to invoke replicas with the same plant state (Section 5.2), and how to design a voter that is capable of supporting this construction and that is sufficiently simple to be implemented at the hardware level as a trusted-trustworthy component (Section 5.3). Then in Section 5.4, we bring everything together and discuss in Section 5.5 why it is safe to deploy our solution in an environment that meets the conditions laid out in the fault model in Section 4.

5.1 Converting stateful replicas into statelessly-recoverable instants

Control tasks, like other applications, modify their internal state. For example, both single and multi-threaded control tasks typically implement function invocation and local variables using a stack, they use global variables and, at least during startup, they may allocate objects on the heap. The easiest way of converting this state into easily recoverable information is to make all state read only and to store it in ECC-protected memory. This way, accidental

faults cannot manifest in the state and the error correcting code (ECC) captures accidental faults that occur in the memory block. Moreover, by making state read only, adversaries have no chance of modifying it without bypassing the processor's protection mechanism[6].

Indeed, it will not be possible to make the entire state of a control task read-only. At least the stack must remain writeable to support function calls and local variables. Fortunately, resetting the stack pointer to the location before a function call discards all values a previous function call has pushed. Therefore, to trivially recover control tasks, we turn the control loop of these tasks into a call to a function, which, as we shall see, will never need to return. Instead, it checks the voter to see whether the previous epoch was successful, which determines whether the current plant state should be considered or whether the replica needs to execute the control problem of a previous epoch, by reaching to that epoch's captured state and sensor values, to reach consensus about this epoch's control problem. It then proposes the actuation value and whatever part of this dynamic state should be preserved for the next epoch. Then, because we rejuvenate control tasks irrespective of their fault status, the only remaining part is to return to the control loop function while resetting the stack pointer. In other words, we turn the control loop function into a continuation and invoke it after every rejuvenation of the control task.

As we have seen, the state that control algorithms need to carry across epochs may range from a few values (such as the error and accumulator for PID) to several kilobytes of data (as in the electric microgrid controller from Huo et al. [27]). Our strategy for protecting this state is to store it in consensually-updated memory [19]. Consensually-updated memory is a memory shared among several replicas. To read, replicas can directly access the memory as it can be mapped read-only into the replica's address space. However, writing requires agreeing on which part of the memory should be updated and how. We leverage the voter to perform also these updates. In particular, we propose simultaneously all updates and the actuation signal to avoid inconsistencies due to partial updates. Also, since we collect proposals over k epochs, we cannot go back in time to receive additional parts of a proposal from a replica.

Control tasks not specifically built for our system will include code to read and write parts of this state. The transformation required to turn these control tasks into consensual-resilient-control (CRC) aware tasks is as follows. We analyze the program and allocate space on the stack in the context of the control-loop continuation. Upon the first write to a variable in consensual memory, we create a copy in that space and modify this copy instead of the original location[7]. Subsequent reads and writes then refer to this copy instead of the original location, and finally, the value of this copy is proposed as part of the update that the voter should apply. Transformations like the above are readily available in modern compilers (e.g., when constructing single-assignment form).

Figure 4 illustrates the above on an example address space layout of control task applications. Replicas receive a read-only copy of the plant state (see next section) and share as read-only mapping the code, read-only data, and shared memory. Stack and the MMIO interface to invoke the voter remain mapped in a writable manner. As part of the first write, a copy of the consensually updated state is created in the space allocated in the control-loop continuation's context on the stack, and all subsequent reads and writes are directed to this copy. The last operation of the control-loop continuation (fn_ctl) is to propose the copy as part of the state update and with the actuation signal, which the voter applies once $f + 1$

[6] We hope future safety-critical systems will be constructed from hardware components that are resistant to protection-bypassing attacks, such as Rowhammer [29].

[7] Being at the top of the stack, functions called from the continuation can reach this space.

Figure 4 Address space layout of a control task replica. Shared state, code, and data are mapped read-only into the address space. Dashed lines indicate the data flow for variables in the consensually updated shared memory. Upon the first write, a copy is created on the stack and finally proposed to update the state after reaching a consensus. After reset, the instruction pointer (IP) is reset to the control loop function (fn_ctl) and the stack pointer (SP) to the beginning of the stack.

agreement with other replicas is reached. Irrespective whether or not fn_ctl terminates, the control task will be resumed in the next epoch with that function, after resetting its stack to remove any modifications an attack could have performed. This is important since compromised replicas may fail in an arbitrary manner, including by not proposing or by not terminating.

5.2 Sensing and control-task invocation

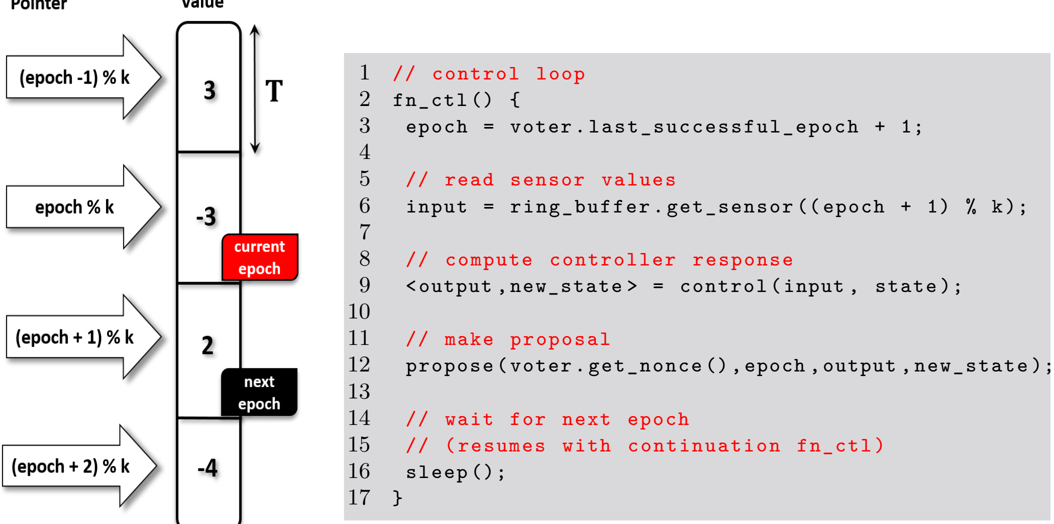
This section explains how we ensure replicas are invoked with the same view of the plant, both in a hosted environment and on bare metal. We start by looking at the implications of not invoking replicas with the same view of the plant. In this case, each replica would need to sense the plant state individually and would produce slightly different actuation signals and values to carry to the next epoch, even if we consider only correct replicas. Consequently, the voter would now need to identify when values are sufficiently close, which adds extra complexity to support this form of approximate agreement.

Instead of adding this complexity to the voter, it can also be added to the replicas by either reaching agreement on the sensed values or by agreeing on the actuation value and state update before presenting this to the voter. In either case, first reaching agreement requires collecting the opinions of $f + 1$ healthy replicas, which can only be guaranteed to happen after k epochs. Therefore, any additional agreement would require the plant to tolerate an extra k epochs of deadline misses, which would severely limit the applicability of our approach. In the following, we therefore present solutions that do not require additional agreement rounds other than for actuating the plant and updating consensual memory.

5.2.1 Hosted environments

Assuming a trusted-trustworthy operating system (OS), OS-level drivers could read sensors on the replicas' behalf and provide them with the values they read. Since replicas may need to revert to the past k elements, a $k + 1$ element ring buffer suggests itself as data structure, which the RTOS can map to the replicas' address spaces in a read-only manner.

Figure 5 shows the ring-buffer data structure used to grant the control task access to past sensor values and the pseudocode for a very simple controller leveraging this data structure. Retrieving from the voter the last epoch where votes were successful, replicas either contribute to the current control problem at hand (if this was the previous epoch)



(a) Ringbuffer's W/R mechanism. **(b)** Simple controller pseudocode.

Figure 5 Controller function fn_ctl and the ring buffer data structure used to refer back to previous plant states in case the previous epoch was not successful.

or they contribute to forming a majority for a past control problem that has failed so far to reach an agreement. As the code shows, with the ringbuffer in place, both cases can be treated in the same way, by obtaining the sensor value from the buffer (Line 6), computing the control output and state to carry over (Line 9) and by proposing both (Line 12) before yielding or sleeping until the next invocation. Replicas will be woken up as part of the rejuvenation process and resume at the beginning of the control loop captured in fn_ctl (at Line 3).

```
1 on rotary_interrupt:
2   epoch = (now() - start_time) / T;
3   angle[(epoch + 1) mod m] += direction()
4   return from interrupt
```

Figure 6 Interrupt handler for decoding rotary controller interrupts from the rotary encoder sensors of our pendulum into angular values (See also the pendulum in Section 2).

Let us illustrate the use of this data structure on an example with less cooperative sensors. Rotary encoders do not reveal the angle directly, but instead signal a change of their rotation angle by triggering interrupts. In our running example, we use the data structure shown in Figure 5 to sample the angles of the rotary controllers from the interrupts they generate at the rising and falling edge. To obtain the desired angle, rotary controllers require the operating system to accumulate angular changes, which they notify through interrupts. The interrupt handler code in Figure 6 shows this decoding of interrupts to angular values, where angle is the ringbuffer shown in Figure 5(a) and direction returns +4 or -4, depending on whether the encoder was turned right or left (i.e., depending on which of the two channels preceded the other (see Section 2)).

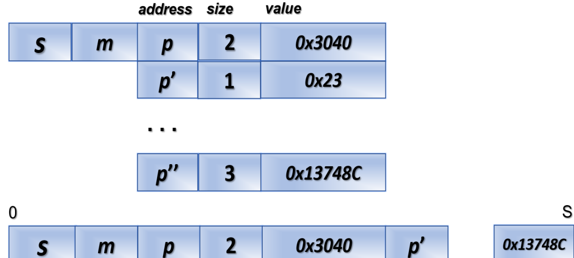


Figure 7 Layout of one of the voter buffers. The size s of the proposal and its inner structure in the form of m address, size, value triples are stored consecutively for easier comparisson.

5.2.2 Bare metal

Not all control tasks run in a hosted environment. In the following, we therefore sketch how replica invocation and sensing can be handled in a simple microcontroller for applications running on bare metal. We assume that in such an architecture, we still have the possibility to statically configure privileges (before the system starts critical operation) and to program a non-maskable timer to enter a read-only interrupt service routine that executes the code to activate the fn_ctl continuation at the beginning of the epoch and to reset the stack accordingly (Line 2 in Figure 5(b)).

Without additional hardware support, replicas, in a bare-metal configuration, have to sense themselves, which requires agreement and plants that tolerate at least $2k$ deadline misses before actuation can be guaranteed. To avoid the overhead entailed with this agreement, we suggest deploying capture hardware units [62] to periodically sample sensors in a reliable way and store the sampled results in memory that gets mapped to the replicas' address spaces in a read-only manner. Deploying $2f_{ccu}+1$ such units, where f_{ccu} is the tolerated fault threshold for these units allows replicas to immediately mask wrong sensor values and proceed with their control tasks. In particular for interrupt-driven sensors, such as the rotary controllers of our pendulum, the capture units should perform the accumulation task to avoid replicas having to accumulate captured interrupts themselves.

5.3 Voting on state updates and actuation

Consolidating the replicas' proposals into a single actuation output turns the voter into a necessarily trusted component, which, to be trustworthy, should remain as simple as possible. However, unlike voting in traditional TMR systems, not all proposals are available simultaneously, which requires the voter to buffer requests before $f+1$ matching votes can be extracted. In particular, we need nk buffers for $n \geq f+1$ replicas and for $k = \left\lceil \frac{f+1}{n-f} \right\rceil$ epochs.

For our pendulum and most systems we have investigated, actuation amounts to writing several memory-mapped registers, where the final write typically triggers the actuation. Likewise, consensual memory updates of state that should be carried to the next epoch also amount to writes to ECC-protected memory, respectively, to multiple locations in case of RAID. These write locations are typically not consecutive and, as we have seen before, proposals must be submitted in their entirety. Therefore an interface suggests itself where replicas specify m writes as address-size-value triples, stored as s consecutive bytes, as shown in Figure 7. This way, the $f+1$ matching proposals can be identified by matching s and the strings of size s in the respective buffers. Moreover, the voter can apply a successful vote (i.e.,

one reaching $f + 1$ agreement) by performing the m writes to the specified locations. Aside from maintaining the order of writes, we did not see any need for sophisticated consistency models other than register consistency, since voter-initiated writes will typically happen after replicas end their activity in an epoch and before the next epoch starts. In addition, healthy replicas may identify state updates in progress, by means of simple sequence locks in consensual memory.

To interface with the voter, we implement channels and map each channel to one replica. Moreover, we make the voter aware of epochs by exposing two read-only registers to each replica. The first contains the current epoch and is advanced every T. The second contains the epoch number when the last successful vote has happened (see Line 3 in Figure 5(b)). Making the voter aware of epochs avoids costly operations when resetting replicas, which otherwise would require changing the permissions of a replica to use a different channel. Upon receiving a message through the channel, the voter copies the proposal to the respective buffer for this replica and the epoch it is executing in, rotating through buffers as epochs advance.

As indicated in our fault model, we further complicate the case of faulty replicas reaching $f + 1$ agreement by introducing a challenge response mechanism to the voter interface. At the start of each epoch, after rejuvenating all replicas, the voter presents each replica a different random value – called *nonce*, which they are asked to reflect to the voter by xor-ing their proposal with this value. Then, rather than comparing the strings bytewise, the voter first xors the proposed string with the replica's current nonce, which returns the original string and then tries to find $f + 1$ matching proposals. This way, accidentally faulty replicas, in addition to adversaries needing to guess their proposal, must still be sufficiently correct to encode both the state update and the actuation signal using the provided nonce and to propose both to the voter before they are rejuvenated at the end of the epoch, which is highly unlikely. Notice that because the nonce is random and different every epoch, replicas which do not propose in an epoch are automatically considered as faulty replicas.

In preparation for changing the active replica set, we equip voters with more than n channels and with buffers for more than k epochs. This way, the active set can be supported with a subset of the resources available in the voter. A trusted replica manager can change this subset and the parameters n, k, and f over time, should that be necessary. The active subset is encoded in a bit vector with one bit per channel (considering those channels as active whose bit is set). Replicas with access to an inactive channel may already propose, but are ignored until their channel becomes active. This way, additional replicas can already be started and allowed to participate while the previous set of replicas is still in control of the plant. Then, once the new set of replicas are prepared, the trusted replica manager atomically transitions to this set by means of writing a single register. The change will become effective at the beginning of the next epoch. Figure 8 shows the channels, buffers, epoch registers as well as the reconfiguration registers just described.

From the description above, it should be clear that such a voter can be implemented as a service at the application level (waiting for signals from the replicas) as an operating-system service (invoked by system calls) or as a fixed-function custom logic mapped to an FPGA or implemented in silicon. In the latter two cases, replicas interface with the voter through memory-mapped IO registers that are mapped into the replicas' address spaces.

Notice also that while replicas must produce identical actuation signals and state updates to reach agreement (given the same consensually updated state and sensor values), they may (and in fact should to ensure fault independence) compute these proposals in a sufficiently different manner, such that an attack of one replicas does not automatically apply to others.

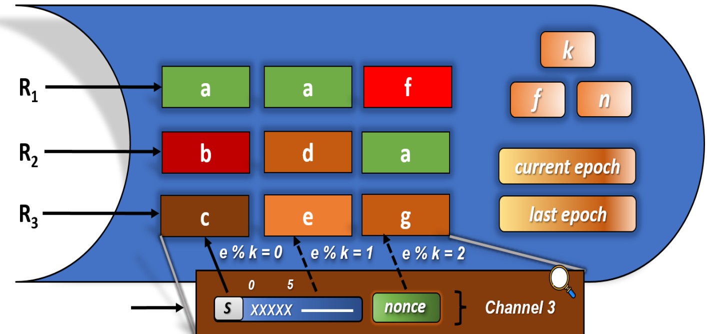


Figure 8 Voter internal structure. The voter provides one buffer per replica and epoch, which the replica can access through a channel. The proposal communicated through the channel is copies into the corresponding buffer of this replica for the current epoch. The voter reveals as well the current epoch and the last epoch where $f + 1$ agreement could be reached and allows k, f and n to be reconfigured by a trusted replica manager (if necessary).

5.4 Bringing it all together

With the above building blocks, we can now bring everything together. The system starts by initializing the plant, the trusted replica-management service (if necessary), and the replicas, which enter the control loop (fn_ctl) as a continuation. When the continuation starts, the RTOS or capture units have already sensed the observable part of the plant and captured that information in a ring-buffer. Therefore if the previous epoch was successful, the replica can proceed with the current sensor values and the current state in consensual memory to produce an actuation signal and update for the state that needs to be carried to the next epoch. Both are proposed to the voter to reach an agreement.

If the previous epoch was unsuccessful, the replica performs the same steps but with the sensor values for the epoch that precedes the last successful one. Notice that in this case, the consensual memory has not been updated and contains the control parameters (e.g., accelerator and previous value for PID control) required for that epoch. Once the voter receives $f + 1$ matching proposals, it marks the current epoch as the last successful and executes the agreed-upon sequence of writes, updating the consensual memory and actuating the plant.

Healthy replicas end their activity in an epoch by sleeping. However, regardless of whether a replica sleeps, the RTOS/non-maskable timer will signal a protected handler in the replicas, which resets the replica by returning to the start of the control-loop continuation (fn_ctl) and by resetting the replica's stack.

5.5 Safe Deployment

In our fault model in Section 4, we have defined constraints for the safe deployment of systems that implement our solution. If these constraints are met, control tasks will reach an agreement, and the agreement is on a correct proposal only. System safety then depends on correct control tasks proposing correct actuations in the given situation, which to ensure is out of the scope of this work.

The constraints highlighted in Section 4 are that (i) no more than f total faults occur and that (ii) the system addresses faults faster than kT with no further faults occurring before that time. In particular, we have discussed that malicious faults must be constrained through diversification such that not all replicas become faulty simultaneously, with the additional

constraint that the time to re-deploy an attack remains above kT. We have also discussed that persistent faults, which cannot be handled at that timescale need to be accounted for. That is, if there are $f_{persistent} < f$ faults, present, only up to $f - f_{persistent}$ faults of another kind may occur within the sliding windows of length at least kT.

With up to f faults over a time kT, at most f proposals may originate from faulty of otherwise compromised replicas. Therefore, when $f + 1$ matching proposals are collected, they are collected from at least one healthy replica. In particular, by the measures we discuss in Section 5.2 and Section 5.3, we ensure that all replicas operate from the same state and are invoked in a reliable manner after rejuvenation. This means (due to our assumption that fused sensor values are correct) that any healthy replica will propose a correct proposal and that agreement can only be reached on such a proposal.

It is always possible to reach agreement on such a proposal because over up to k epochs, $n - fk$ replicas are correct (possibly after rejuvenating them), which because $k = \left\lceil \frac{f+1}{n-f} \right\rceil$, is larger or equal to $f + 1$. Hence the system is life.

To see why our system is also correct in case up to f faults occur during each epoch, but with the additional constraints that (iii) among the fk faults over any sliding window of length kT only up to f are malicious faults, (iv) that malicious faults do not propose the same value as accidental faults and (v) no two accidental faults agree in their value, we have to see that agreement cannot already be reached among faulty replicas. Condition (iii) rules out agreement just by including malicious replicas and (iv) that malicious replicas collude with an accidentally faulty replica, even if up to f malicious replicas agree in their proposal. (v) avoids agreement among accidentally faulty replicas. Notice though that these are probabilistic arguments and that, as mentioned in Section 4, systems cannot safely be deployed if the residual likelihood of agreement among accidentally faulty replicas or if in the targeted environment, the residual likelihood of malicious replicas guessing the fault characteristics of an accidentally faulty replica, cannot be tolerated by the system. Our challenge to require replicas to send their proposals by xor-ing a voter provided nonce, further reduces these likelihoods, as accidentally faults replicas must remain able to do so, despite the fault manifesting.

The above condition also ensures liveness in this setting, since $n - f$ replicas remain correct in each epoch, which, when collecting their proposals over k epochs sums up to at least $f + 1$ proposals from correct replicas.

5.6 Distributed Control

Until now and for our evaluation, we have assumed systems that are sufficiently tightly coupled so that communication through shared, ECC- and RAID-protected memory remains possible. In such a setting, a voter can collect proposals, update consensual memory and actuate the plant. The same applies to closely coupled nodes for the control task, but a remote plant as long as communication between the voter and the plant is reliable.

To support distributed nodes for the control tasks and a remote plant, both the sensor signals and the actuation signals must be communicated reliably to all notes in the system (e.g., by using communication media that already have such a reliability built in [22] or by running a suitable reliable transmission protocol [33, 44]. In such a setting, shared memory will likely not be available and should therefore be replaced by consensually updating locally accessible, read-only mapped memories and by reconstructing the state of such a memory from its peers in case one of the nodes' memory fails. Investigating the tradeoffs of such a solution is out of the scope of this paper.

6 Evaluation

To demonstrate that our approach to consensual resilient control is in fact robust, even in the presence of errors, we have implemented the voter and two control algorithms (LQR and PID) by leveraging Linux's user-level driver infrastructure to sense and actuate our inverted pendulum. In particular, we were interested in whether the ability to tolerate accidental faults in the time domain allows controlling such an application from the less predictable environment that standard Linux offers. For small epochs ($T < 10ms$), we had to use Linux' "silent core" feature to limit OS activity on the cores to which we pinned our control tasks.

All measurements were conducted on a 4-core Raspberry 3 Model B+ with 1GB RAM, running at 1.4 GHz, using the pendulum shown in Section 2. In addition, to evaluate the scalability of our approach, we used a 4x6 core Intel Xeon Gold 6334 CPU, running at 3.4 GHz, and a software emulator of the pendulum (implementing its equation of motion and random turbulence).

We have implemented the voter in software as a user-level task and have pinned replicas and the voter each to a separate core. Replicas communicate with the voter through a dedicated shared memory region, as depicted in Figure 4, which implements the voter's channel interface. We inject faults into randomly selected replicas and consider only faults that manifest in proposing values that are different from those of healthy replicas. For accidental faults, a random value is proposed. For maliciously-induced faults, we as well select a random value but will use the same value for all compromised replicas. In addition, for demonstration purposes, pressing a button on the Raspberry PI will as well cause a random replica to fail.

6.1 Overhead

Since our approach is to re-execute replicas after rejuvenation for up to k epochs, the runtime overhead in terms of time to agreement under faults is dominated by the number of epochs required to collect $f + 1$ faults. In no faults occur, replicas actuate within a single epoch and the worst-case time to agreement is the WCET of the control task plus the overhead to propose and update the state that needs to be preserved across epochs.

We measured this overhead for PID and LQR on the Raspberry PI and with our inverted pendulum ($f = 1, n = 2$) to be 0.39 μs for the time that the replica needs to preserve the state for the next epoch, by proposing the error and accumulator (PID) and the measured angles to calculate angular velocities (LQR). The voter required 0.034 μs to update consensual memory and actuate the plant.

In addition, we performed a series of microbenchmarks on our x86-based simulator of the pendulum to understand these overheads for different controllers that require preserving increasing amounts of state across epochs. Figure 9 shows these results for the same scenario ($f = 1$ and $n = 2$). Shown are the maximum observed (bars), average (green dot) and P95 (top) and P05 (bottom) percentiles of these execution-time overheads.

As can be seen, the overhead of turning control tasks into statelessly-recoverable instants, by pushing all state that needs to be maintained across epochs to consensual memory, is negligible for controllers with small state and well below 2ms for controllers that operate on a significant amount of dynamic state (such as the one from Huo et al. [27]). It should be noted that typical book-keeping tasks can also be performed on consensual memory, with little additional overhead for logging system states in consensual memory.

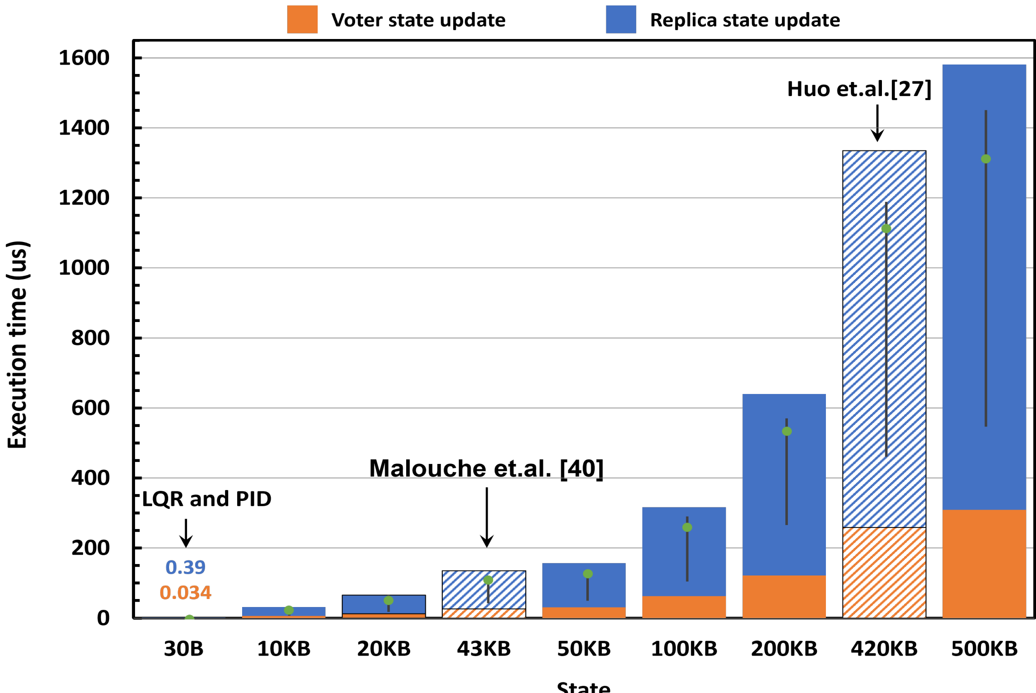


Figure 9 Overhead of consensual resilient control (in μs) broken down into the overhead on the replica side to propose the actuation value and update of the state that should be preserved for the next epoch and into the voter overhead of applying this update and the actuation signal. Shown is the scenario for $f = 1$, $n = 2$.

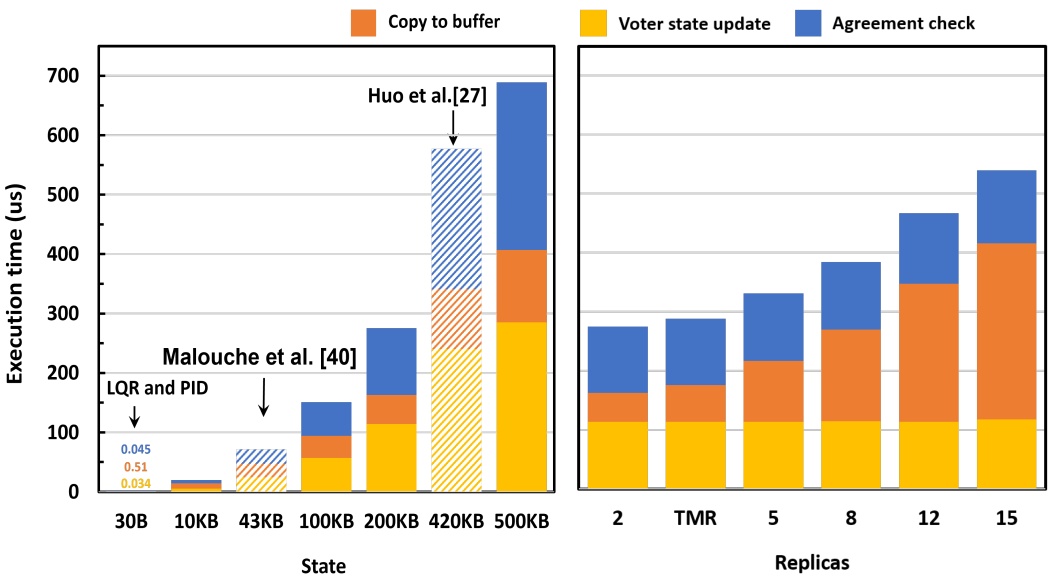
6.2 Breakdown of Voting Overheads

Figure 10 further breaks down the overhead for voting into the different operations that a software-level voter needs to perform. Hardware implementations can avoid buffering costs by directing inputs directly to the current epoch's buffer and they may parallelize agreement checks. Figure 10(a) investigates for $n = 2$ replicas how the voting overhead scales with the size of the state that needs to be preserved across epochs. Figure 10(b) shows these results for increasing n and therefore also for increasing f and a fixed state size of 200KB.

As can be seen, updating consensual memory, copying to the buffer and checking for agreement is linear in the size of the proposal, given that the number of replicas is fixed to $n = 2$ for these measurements. Similarly, updating consensual memory and copying to the buffer are constant for a fixed-size message, irrespective of the number of replicas and the agreement check linear in the number of replicas (and hence faults tolerated) in case no faults occur (as shown in 10(b)) and quadratic $(n \cdot k)$ when $f + 1$ agreement must be collected over up to k epochs. This is because whenever a replica proposes, the voter checks this proposal to all buffers that already contain a proposal for the voted-upon epoch.

6.3 Actuation Signals

Figure 11 shows the sensor (Channel 2–4) and actuation signal (Channel 1) of the inverted pendulum, controlled over three epochs with a consensual resilient PID controller. The scenario depicted in the figure roughly resembles the situation presented in Figure 3. During the first epoch, actuating at vertical line (4), no faults happen and the DC motor gets



(a) Breakdown of the overhead of voting for $n = 2$.

(b) Scalability of voting overhead for larger f and n. Shown are the results for a 200KB cross epoch state.

Figure 10 Voting overhead for the constant invocation of $T = 25$ms.

configured to a 66% duty cycle, as seen in the wider pulse width in Channel 1. As a response to this actuation, the rate of change of the angle drops, as can be seen from the longer distances between the rising and falling edges on Channel 2 and 3. Therefore, the control algorithm selects a lower duty cycle to reduce motor velocity and slow down the cart and thereby also the pendulum motion even further, with the idea of reaching the stable point where the pendulum is pointing straight to the top. Unfortunately, a replica fails during that epoch (since we injected a fault). In consequence, after the voter receives $f + 1$ proposals at the point in time denoted by vertical line (5), no agreement can be reached and the voter will hold the previous duty cycle of 66%. In the following epoch, we again inject a fault into one of the replicas, but this time, $f + 1$ agreement can be reached by combining the proposals of the current and the previous epoch. The voter applies the proposal and adjusts the duty cycle to 8.3%, as can be seen in the change of the pulse-width encoded signal. We also see slight variations between the actuation points. This is due to the control replicas executing with slightly different actual execution times within the $7ms$ epochs.

6.4 Rejuvenation costs

A central contribution of this work is the reduction of rejuvenation costs to just resuming the control loop continuation (fn_ctl) and resetting the stack, which both have overheads in the single to double-digit cycle range. In addition, we induce a maximum observed overhead of $0.39\mu s$ (with LQR and PID) for proposing and updating the state in consensual memory that must be preserved across epochs.

To compare and contrast these costs, we have also measured the average-case overhead when rejuvenating replicas traditionally by creating a new process ($329.06\mu s$), a new thread ($13.28\mu s$) and by mapping the voter interface to this replica ($100.82\mu s$). In addition, such a replica would experience cold start effects and need to catch up to the state of other replicas.

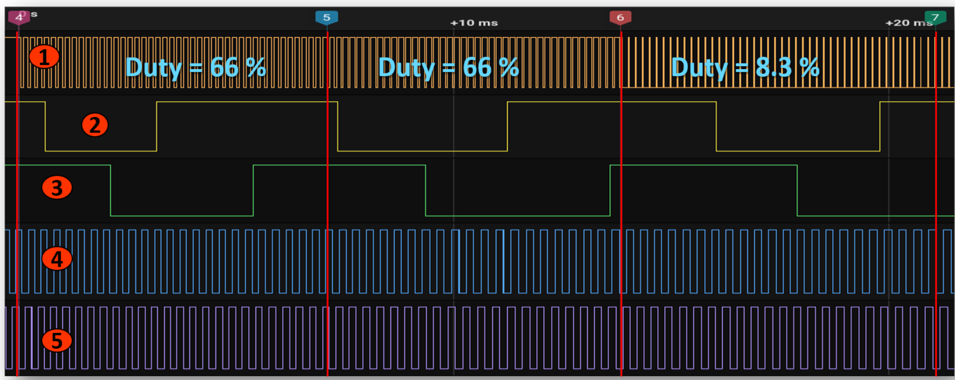


Figure 11 Sensor and actuation signals of the pendulum were evaluated using a logic analyzer. Shown are the points in time of actuation (vertical lines) for three epochs (marked on the top as 4, 5 and 6). The individual channels show DC motor actuation (1), encoded as a pulse-width modulated signal, the two channels of the rotary encoder which measures the angle of the pendulum (2) and (3), as well as the two channels measuring the position (4) and (5).

However, as can already be seen from the reported numbers, the costs of rejuvenating replicas traditionally are significant. It should also be noted that it is difficult to bind these costs from above, which is why typically, real-time systems only use these operations while they have to guarantee timeliness. Notice that rejuvenation will also be required in systems, such as TMR, that are capable of masking faults. This is because persistent attacks exhaust the healthy majority over time. Rejuvenation restores this majority.

7 Conclusion

This paper presents Consensual Resilient Control, a framework specifically designed to ensure the resilience of control tasks to accidental and malicious faults. We have shown how stateful control tasks can be systematically transformed into statelessly-recoverable instances by protecting in consensual memory any state that must be preserved across epochs (such as the PID controller's error and accumulator for derivative and integral control). This allows masking faults just with a detection quorum of $n \geq f+1$ replicas, provided $f+1$ agreement can be collected over $k = \left\lceil \frac{f+1}{n-f} \right\rceil$ epochs and provided the plant can tolerate up to k deadline misses.

We discuss several intricacies of applying consensual resilient control to a real-life application scenario by demonstrating its operation with our self-made inverted pendulum. In addition, we have evaluated our approach in the less predictable setting of running controllers as user-level Linux processes with user-level drivers for sensing and actuating the plant. We have also conducted several microbenchmarks to assess the behavior of consensual resilient control when increasing amounts of state have to be preserved across epochs (up to the 420KB required for the MPC electric microgrid controller by Huo et al. [27]) as well as for increasing f and n.

In the future, we plan to investigate scenarios requiring a change of the control algorithm, as well as simplex/complex controller interplays in a fault-tolerant setting.

References

1 Loveless A, Dreslinski R, Kasikci B, and Phan LT. Igor: Accelerating byzantine fault tolerance for real-time systems with eager execution. In *IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 360–373, May 2021.

2 Leonie Ahrendts, Sophie Quinton, Thomas Boroske, and Rolf Ernst. Verifying weakly-hard real-time properties of traffic streams in switched networks. In *ECRTS 2018-30th Euromicro Conference on Real-Time Systems*, pages 1–22, 2018.

3 Charles W Anderson. Learning to control an inverted pendulum using neural networks. *IEEE Control Systems Magazine*, 9(3):31–37, 1989.

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

5 Günther Bauer and Hermann Kopetz. Transparent redundancy in the time-triggered architecture. In *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, pages 5–13. IEEE, 2000.

6 Robert C Baumann. Soft errors in advanced semiconductor devices-part i: the three radiation sources. *IEEE Transactions on device and materials reliability*, 1(1):17–22, 2001.

7 Guillem Bernat, Alan Burns, and Albert Liamosi. Weakly hard real-time systems. *IEEE transactions on Computers*, 50(4):308–321, 2001.

8 Hristo Bojinov, Dan Boneh, Rich Cannings, and Iliyan Malchev. Address space randomization for mobile devices. In *Proceedings of the Fourth ACM Conference on Wireless Network Security (WiSec'11)*, pages 127–138, 2011. doi:doi:10.1145/1998412.1998434.

9 Erik Buchanan, Ryan Roemer, and Stefan Savage. Return-oriented programming: Exploits without code injection. In *Black HAT USA*, August 2008. URL: https://hovav.net/ucsd/talks/blackhat08.html.

10 Gang Chen, Nan Guan, Kai Huang, and Wang Yi. Fault-tolerant real-time tasks scheduling with dynamic fault handling. *Journal of Systems Architecture*, 102:101688, 2020.

11 Thomas Chen and Saeed Abu-Nimeh. Lessons from stuxnet. *Computer*, 44(4):91–93, 2011.

12 Hongjun Choi, Sayali Kate, Yousra Aafer, Xiangyu Zhang, and Dongyan Xu. Software-based realtime recovery from sensor attacks on robotic vehicles. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 349–364, 2020.

13 Roth E and Haeberlen A. Do not overpay for fault tolerance! In *IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 374–386, May 2021.

14 Haoyang Fan, Fan Zhu, Changchun Liu, Liangliang Zhang, Li Zhuang, Dong Li, Weicheng Zhu, Jiangtao Hu, Hongye Li, and Qi Kong. Baidu apollo em motion planner. *arXiv preprint arXiv:1807.08048*, 2018.

15 Joachim Fellmuth, Thomas Göthel, and Sabine Glesner. Instruction Caches in Static WCET Analysis of Artificially Diversified Software. In Sebastian Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, volume 106 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:23, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECRTS.2018.21.

16 S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Hot Topics in Operating Systems*, pages 67–72, 1997. doi:doi:10.1109/HOTOS.1997.595185.

17 Markus Fras, H Kroha, O Reimann, B Weber, and R Richter. Use of triple modular redundancy (tmr) technology in fpgas for the reduction of faults due to radiation in the readout of the atlas monitored drift tube (mdt) chambers. *Journal of Instrumentation*, 5(11):C11009, 2010.

18 Goran Frehse, Arne Hamann, Sophie Quinton, and Matthias Woehrle. Formal analysis of timing effects on closed-loop properties of control software. In *2014 IEEE Real-Time Systems Symposium*, pages 53–62. IEEE, 2014.

19 Inês Pinto Gouveia, Marcus Völp, and Paulo Esteves-Verissimo. Behind the last line of defense: Surviving soc faults and intrusions. *Computers & Security*, 123:102920, 2022.

20 Blessing Guembe, Ambrose Azeta, Sanjay Misra, Victor Chukwudi Osamor, Luis Fernandez-Sanz, and Vera Pospelova. The emerging threat of ai-driven cyber attacks: A review. *Applied Artificial Intelligence*, 36(1):2037254, 2022.

21 Li H, Lu C, and Gill CD. Rt-zookeeper: Taming the recovery latency of a coordination service. In *ACM Transactions on Embedded Computing Systems (TECS)*, volume 20, pages 1–22, September 2021.

22 Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: Practical accountability for distributed systems. *ACM SIGOPS operating systems review*, 41(6):175–188, 2007.

23 Zain AH Hammadeh, Rolf Ernst, Sophie Quinton, Rafik Henia, and Laurent Rioux. Bounding deadline misses in weakly-hard real-time systems with task dependencies. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 584–589. IEEE, 2017.

24 Richard W Hamming. Error detecting and error correcting codes. *The Bell system technical journal*, 29(2):147–160, 1950.

25 Pengcheng Huang, Hoeseok Yang, and Lothar Thiele. On the scheduling of fault-tolerant mixed-criticality systems. In *Proceedings of the 51st annual design automation conference*, pages 1–6, 2014.

26 Todd E Humphreys, Brent M Ledvina, Mark L Psiaki, Brady W O'Hanlon, Paul M Kintner, et al. Assessing the spoofing threat: Development of a portable gps civilian spoofer. In *Proceedings of the 21st International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS 2008)*, pages 2314–2325, 2008.

27 Yuchong Huo, François Bouffard, and Géza Joós. Integrating learning and explicit model predictive control for unit commitment in microgrids. *Applied Energy*, 306:118026, 2022.

28 Greg Jaffe and Thomas Erdbrink. Iran says it downed u.s. stealth drone; pentagon acknowledges aircraft downing. The Washington Post, December 2011.

29 Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

30 H. Kopetz and G. Grunsteidl. Ttp - a time-triggered protocol for fault-tolerant real-time systems. In *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*, pages 524–533, 1993. doi:10.1109/FTCS.1993.627355.

31 Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

32 C Mani Krishna. Fault-tolerant scheduling in homogeneous real-time systems. *ACM Computing Surveys (CSUR)*, 46(4):1–34, 2014.

33 Leslie Lamport. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*, pages 277–317. ACM, 2019.

34 Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security and Privacy*, 9(3):49–51, 2011.

35 P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated software diversity. In *IEEE Symposium on Security and Privacy*, 2014. doi:doi:10.1109/SP.2014.25.

36 Robert M. Lee, Michael J. Assante, and Tim Conway. German steel mill cyber attack. Industrial Control Systems - avail at: https://ics.sans.org/media/ICS-CPPE-case-Study-2-German-Steelworks Facility.pdf, December 2014.

37 Robert E Lyons and Wouter Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM journal of research and development*, 6(2):200–209, 1962.

38 Michael R Lyu et al. *Handbook of software reliability engineering*, volume 222. IEEE computer society press Los Alamitos, 1996.

39 Martina Maggio, Arne Hamann, Eckart Mayer-John, and Dirk Ziegenbein. Control-system stability under consecutive deadline misses constraints. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

40 Ibtissem Malouche, A Kheriji Abbes, and Faouzi Bouani. Automatic model predictive control implementation in a high-performance microcontroller. In *2015 IEEE 12th International Multi-Conference on Systems, Signals & Devices (SSD15)*, pages 1–6. IEEE, 2015.

41 Aleksandar Matović. Case studies on modeling security implications on safety, 2019.

42 Sibin Mohan, Stanley Bak, Emiliano Betti, Heechul Yun, Lui Sha, and Marco Caccamo. S3a: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems. In *Proceedings of the 2nd ACM international conference on High confidence networked systems*, pages 65–74, 2013.

43 Djob Mvondo, Alain Tchana, Renaud Lachaize, Daniel Hagimont, and Noël De Palma. Fine-grained fault tolerance for resilient pVM-based virtual machine monitors. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 197–208. IEEE, 2020.

44 Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

45 Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete event dynamic systems*, 21:307–338, 2011.

46 Risat Mahmud Pathan. Fault-tolerant and real-time scheduling for mixed-criticality systems. *Real-Time Systems*, 50:509–547, 2014.

47 Krithi Ramamritham and John A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *Proceedings of the IEEE*, 82(1):55–67, 1994.

48 Reza Ramezani and Yasser Sedaghat. An overview of fault tolerance techniques for real-time operating systems. *ICCKE 2013*, pages 1–6, 2013.

49 Michael Riley and John Walcott. China-based hacking of 760 companies shows cyber cold war. *Bloomberg, Dec*, 14, 2011.

50 Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):1–34, 2012.

51 John Rushby. Bus architectures for safety-critical embedded systems. In *International Workshop on Embedded Software*, pages 306–323. Springer, 2001.

52 Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. Loki: Hardening code obfuscation against automated attacks. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3055–3073, Boston, MA, August 2022. USENIX Association. URL: `https://www.usenix.org/conference/usenixsecurity22/presentation/schloegel`.

53 Sarah Scoles. The feds want these teams to hack a satellite – From home. The wired – `https://www.wired.com/story/the-feds-want-these-teams-to-hack-a-satellite-from-home/`, August 2020.

54 Danbing Seto and Lui Sha. A case study on analytical analysis of the inverted pendulum real-time control system. Technical report, Carnegie-Mellon University, 1999.

55 Yanyan Shen, Gernot Heiser, and Kevin Elphinstone. Fault tolerance through redundant execution on cots multicores: Exploring trade-offs. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 188–200, 2019. `doi:10.1109/DSN.2019.00031`.

56 D. Shepard, J. Bhatti, and T. Humphreys. Drone hack. *GPS World*, 23(8):30–33, 2012.

57 Douglas Simoes Silva, Rafal Graczyk, Jérémie Decouchant, Marcus Völp, and Paulo Esteves-Verissimo. Threat adaptive byzantine fault tolerant state-machine replication. In *40th International Symposium on Reliable Distributed Systems (SRDS)*, September 2021.

58 Jill Slay and Michael Miller. Lessons learned from the maroochy water breach. *Critical Infrastructure Protection*, pages 73–82, 2007.

59 Paulo Sousa, Nuno Ferreira Neves, and Paulo Veríssimo. Proactive resilience through architectural hybridization. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 686–690, 2006.

60 Rong Su. Supervisor synthesis to thwart cyber attack with bounded sensor reading alterations. *Automatica*, 94:35–44, 2018.

61 Infineon Technologies. 32-bit aurix™ tricore™ microcontroller. URL: `https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/`.

62 Infineon Technologies. ccu4 capture and compare unit 4. `https://www.infineon.com/dgdl/Infineon-IP_CCU4_XMC-TR-v01_00-EN.pdf?fileId=5546d4624ad04ef9014b0780bb082263&ack=t`.

63 Nils Ole Tippenhauer, Christina Pöpper, Kasper Bonne Rasmussen, and Srdjan Capkun. On the requirements for successful gps spoofing attacks. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 75–86, 2011.

64 Ulf Troppens, Rainer Erkens, and Wolfgang Müller. *Storage networks explained: basics and application of fibre channel SAN, NAS, iSCSI and InfiniBand*. John Wiley & Sons, 2005.

65 Nils Vreman, Anton Cervin, and Martina Maggio. Stability and Performance Analysis of Control Systems Subject to Bursts of Deadline Misses. In Björn B. Brandenburg, editor, *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, volume 196 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:23, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ECRTS.2021.15`.

66 Xin Wang, Keith Holbert, and Lawrence T Clark. Using tmr to mitigate seus for digital instrumentation and control in nuclear power plants. In *7th International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies 2010, NPIC and HMIT 2010*, pages 925–934, 2010.

67 Victor Williams and Kiyotoshi Matsuoka. Learning to balance the inverted pendulum using neural networks. In *[Proceedings] 1991 IEEE International Joint Conference on Neural Networks*, pages 214–219. IEEE, 1991.

68 Aibin Yan, Zhelong Xu, Kang Yang, Jie Cui, Zhengfeng Huang, Patrick Girard, and Xiaoqing Wen. A novel low-cost tmr-without-voter based his-insensitive and mnu-tolerant latch design for aerospace applications. *IEEE Transactions on Aerospace and Electronic Systems*, 56(4):2666–2676, 2019.

69 Kim Zetter. Google hack attack was ultra sophisticated, new details show, January 2010. URL: `https://www.wired.com/2010/01/operation-aurora/`.

70 Kim Zetter. A cyberattack has caused confirmed physical damage for the second time ever. `https://www.wired.com/2015/01/german-steel-mill-hack-destruction`, 2015.

71 Ying Zhang and Krishnendu Chakrabarty. Fault recovery based on checkpointing for hard real-time embedded systems. In *Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 320–327. IEEE, 2003.

72 Junlong Zhou, Min Yin, Zhifang Li, Kun Cao, Jianming Yan, Tongquan Wei, Mingsong Chen, and Xin Fu. Fault-tolerant task scheduling for mixed-criticality real-time systems. *Journal of Circuits, Systems and Computers*, 26(01):1750016, 2017.

73 Xingliang Zou, Albert MK Cheng, and Yu Jiang. P-frp task scheduling: A survey. In *2016 1st CPSWeek Workshop on Declarative Cyber-Physical Systems (DCPS)*, pages 1–8. IEEE, 2016.

Impact of Transient Faults on Timing Behavior and Mitigation with Near-Zero WCET Overhead

Pegdwende Romaric Nikiema
Univ Rennes, Inria, IRISA, CNRS, France

Angeliki Kritikakou
Univ Rennes, Inria, IRISA, CNRS, France

Marcello Traiola
Univ Rennes, Inria, IRISA, CNRS, France

Olivier Sentieys
Univ Rennes, Inria, IRISA, CNRS, France

Abstract

As time-critical systems require timing guarantees, Worst-Case Execution Times (WCET) have to be employed. However, WCET estimation methods usually assume fault-free hardware. If proper actions are not taken, such fault-free WCET approaches become unsafe, when faults impact the hardware during execution. The majority of approaches, dealing with hardware faults, address the impact of faults on the functional behavior of an application, i.e., denial of service and binary correctness. Few approaches address the impact of faults on the application timing behavior, i.e., time to finish the application, and target faults occurring in memories. However, as the transistor size in modern technologies is significantly reduced, faults in cores cannot be considered negligible anymore. This work shows that faults not only affect the functional behavior, but they can have a significant impact on the timing behavior of applications. To expose the overall impact of faults, we enhance vulnerability analysis to include not only functional, but also timing correctness, and show that faults impact WCET estimations. As common techniques to deal with faults, such as watchdog timers and re-execution, have large timing overhead for error detection and correction, we propose a mechanism with near-zero and bounded timing overhead. A RISC-V core is used as a case study. The obtained results show that faults can lead up to almost 700% increase in the maximum observed execution time between fault-free and faulty execution without protection, affecting the WCET estimations. On the contrary, the proposed mechanism is able to restore fault-free WCET estimations with a bounded overhead of 2 execution cycles.

2012 ACM Subject Classification General and reference → Reliability; General and reference → Measurement; Hardware → Error detection and error correction; Hardware → Transient errors and upsets; Hardware → Safety critical systems; Computer systems organization → Real-time system architecture

Keywords and phrases Transient faults, Timing impact, Near-zero WCET error detection and correction, Vulnerability analysis

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.15

Supplementary Material *Software (Source Code)*: `https://gitlab.inria.fr/srokicki/Comet/-/tree/FSR_comet?ref_type=heads`

Funding This work has been funded by the French National Research Agency (ANR) through the FASY research project (ANR-21-CE25-0008).

1 Introduction

1.1 Context

Time-critical systems, such as safety-critical and mixed-criticality systems, consist of hard real-time applications. For such applications, timing guarantees must be provided, i.e., their worst-case response time must be less than their respective deadlines and/or the total

© Pegdwende Romaric Nikiema, Angeliki Kritikakou, Marcello Traiola, and Olivier Sentieys;
licensed under Creative Commons License CC-BY 4.0

execution does not exceed a given latency requirement [47]. To rigorously provide such guarantees, a safe estimation of the Worst-Case Execution Time (WCET) [20] has to be employed during the system design. The estimation of WCET is performed through (i) measurement-based approaches, where the task is executed on the platform under study, and (ii) static analysis approaches, where the software source code and the platform under study are automatically examined, before the application runs [16]. The majority of WCET estimation approaches assumes that the underlying hardware is fault-free, i.e., during WCET estimation no faults occur in the hardware of the target platform [24, 48].

However, in reality, a system is threatened by phenomena that can lead to several permanent or temporary faults, occurring during execution. Especially due to the reduced transistor sizes and lower supply voltages of modern technologies [26, 17], systems are becoming more and more sensitive to environmental sources [39], such as ionization, radiation, and high-energy electromagnetic interference, leading to temporary reliability violations, called transient faults. Transient faults can affect the system behavior by corrupting the system information. Therefore, as systems become more and more prone to faults during execution [21], fault-free WCET bounds cannot be considered safe anymore [48].

1.2 Motivation

To deal with hardware faults, existing approaches apply fault-tolerant techniques to the system. The majority of these works focuses on the impact of faults on the *functional behavior* of the applications. Functional behavior refers to *denial of service*, i.e., no outcome is generated because the application is hanged or crashed, and to *binary correctness*, i.e., the application outcome is different than expected [40].

Fault mitigation considering real-time aspects is usually achieved through scheduling techniques applied at the task-level, such as replication of tasks [27, 5] and task check-pointing/re-execution [12, 51, 50, 27]. When fault tolerance techniques are inserted into the system, their timing impact on WCET has to be taken into account, in order to still provide timing guarantees. To do so, the fault-free WCET is extended with the timing overhead of the applied fault tolerance techniques. However, faults impact not only the functional behaviour, but also the *timing behaviour*, i.e., the application finishes within a given time, but its execution time is different compared to the fault-free execution. This fault impact is bound by the denial of service, i.e., when the execution time exceeds a threshold, it is considered as not responsive. Such application hangs are detected by a watchdog-timer and they are remedied by resetting the system and restarting execution. However, such approaches have significant time overhead, since the transient fault is detected much later than occurred, e.g., when the application finishes execution or the watchdog timer expires. To deal with this limitation, low-level fault-tolerant techniques can circumvent the fault impact, at the time instance when the fault occurs, leading to remedies with significantly lower and bounded time overhead than watchdog timers and less area overhead than replicating the complete processor.

Few approaches address the impact of hardware faults on the timing behaviour of applications. Existing work addresses hardware faults occurring in cache memories, while the rest of the architecture is assumed fault-free. Approaches focus on estimating the timing impact by accounting for the hardware degradation of the cache memory due to the presence of faults, e.g., additional misses due to faulty cache lines [23]. Some approaches have been extended to incorporate the timing impact of inserted fault tolerant techniques to detect, correct or mitigate faults in memories, e.g, when a parity bit is used for error detection [11]. Other works focus on mitigating the hardware degradation in caches, due to occurring faults,

using redundant hardware, e.g., through a shared reliable buffer [24]. As a result, the timing impact of faults on the execution time, and thus the WCET, is mitigated and the timing characteristics of the memory hardware are maintained, leading to a timing behavior close to the fault-free one, despite the presence of faults. However, existing works mainly focus on permanent faults occurring to memories. Nevertheless, with technology size reduction, faults occurring inside the cores cannot be considered negligible anymore [32, 43]. Such faults can significantly affect the execution time of an application.

1.3 Contributions

The contribution of this work is to expose the following key aspect: transient faults affecting the cores impacts not only the functional behavior of an application, but it also has a significant impact on its timing behavior, affecting WCET estimations. To achieve that, we leverage typical fault-free WCET estimations to be fault-aware, by taking into account the impact of transient faults occurring on cores. More precisely, we firstly perform a vulnerability analysis on a target system through extensive fault injection. The analysis verifies not only functional correctness, but also timing correctness of applications, when executed on a core. Then, we apply a typical measurement-based WCET estimation method to verify the impact of faults on WCET estimation. A RISC-V core, named Comet, is used as a case study [41]. Comet is an on open-source High Level Synthesis (HLS) implementation of RV32I base ISA[1].

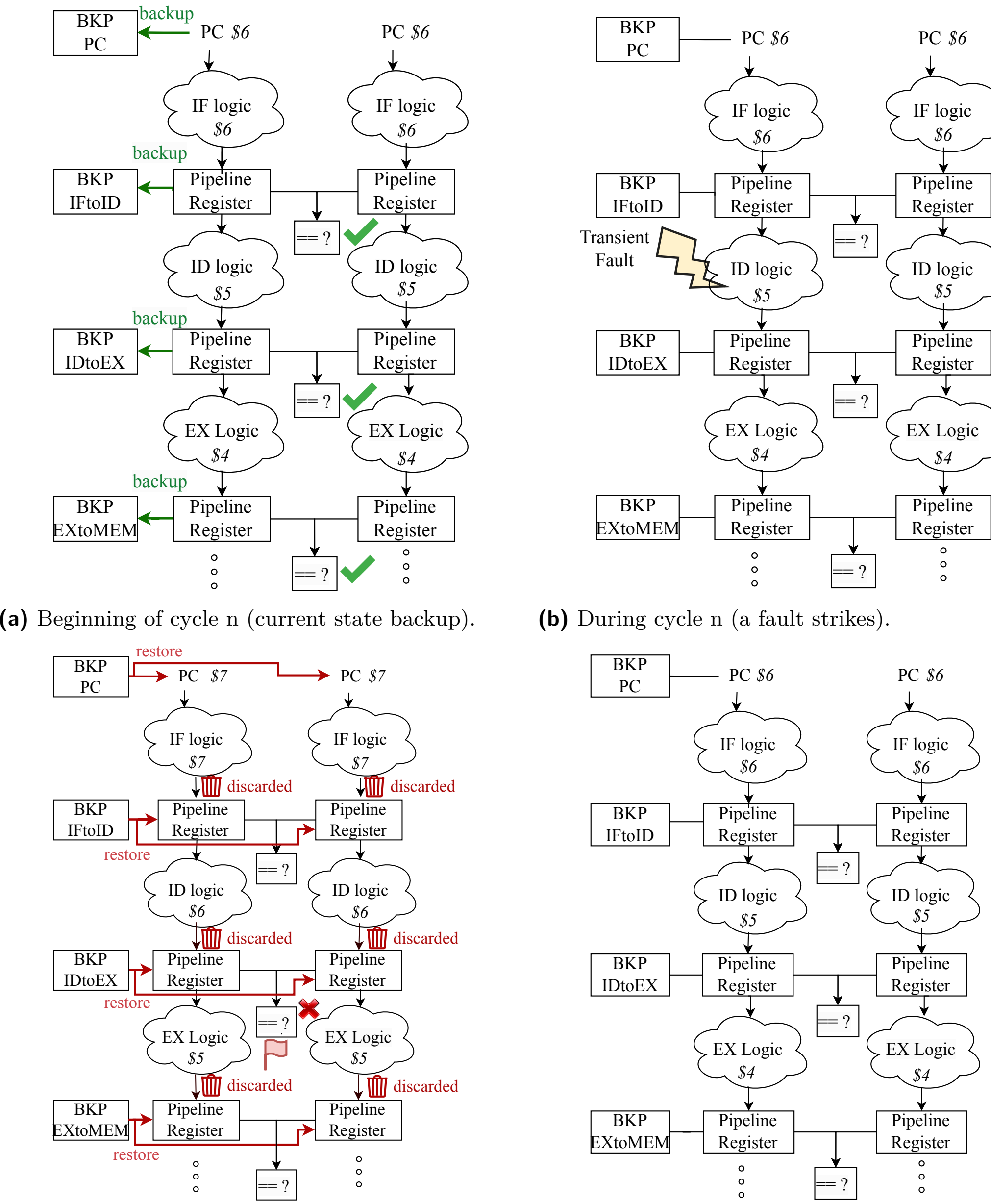
From the obtained results, we observe that the application execution time can be significantly increased under the presence of transient faults, up to 700%, compared to the application execution time without faults. Furthermore, the distribution of execution time traces is significantly modified, compared to the fault-free distribution. The above observations have direct consequences; the time required to finish execution under faults can be significantly higher than the fault-free WCET. Thus, existing approaches should use watchdog timers, in order to bound the impact of transient faults on the application execution time, and keep safe the overall schedule. When the timer expires or an error is detected, the application requires to be re-executed, fully or partially, depending on the approach, leading to high error detection and correction timing overhead. To deal with this limitation, we propose a mechanism with near-zero and bounded overhead (two clock cycles) that circumvents the faults as soon as they occur – before being propagated and affecting the execution time – and thus restores WCET estimations close to the faulty-free one.

The paper is organized as follows. Section 2 describes the methodology followed to obtain fault-aware WCET estimations, based on functional and timing vulnerability analysis combined with a measurement-based WCET approach. Section 3 describes the proposed fault-tolerant mechanism and bounds its timing overhead. Section 4 presents and analyzes the experimental results. Section 5 discusses the related work. Finally, conclusion is presented in Section 6.

2 Fault-aware WCET estimation methodology

This section describes the methodology followed to obtain WCET estimations under transient faults occurring on cores. To obtain realistic fault analysis, hardware fault injection is needed. Thanks to this, faults can be injected in the actual hardware structures, and not only in application variables as done by software fault injection [37]. Hence, a measurement-based WCET estimation method is required in order to be able to analyze the timing impact, when faults are injected in the hardware, compared to a static analysis. Therefore, firstly we perform a vulnerability analysis through hardware fault injection. Then, we apply a

[1] https://gitlab.inria.fr/srokicki/Comet/-/tree/master

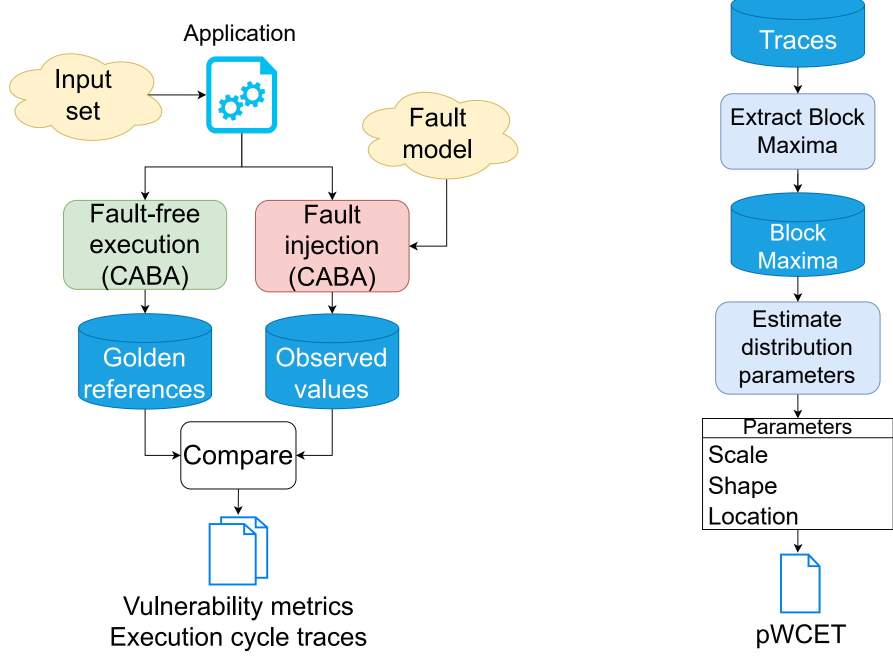


(a) Beginning of cycle n (current state backup). **(b)** During cycle n (a fault strikes).

(c) Beginning of cycle n+1 (error detection & restore). **(d)** Cycle n+2 (error corrected).

Figure 1 Illustration of LESR mechanism.

typical Measurement Based Probabilistic Timing Analysis (MBPTA) to analyze the impact of faults on WCET. The MBPTA is a mathematical method for estimating the extreme values probability of rare events [18, 13]. This method allows us to see the tail behaviour and determine the probabilistic WCET (pWCET) for a set of execution time traces. Note that, the goal of the fault-aware WCET estimation methodology is not to propose a new method to obtain tighter bounds, but to study typical measurement-based WCET estimation approaches in presence of faults. The next paragraphs describe the steps of the fault-aware WCET estimation methodology.



(a) Data collection through vulnerability analysis. **(b)** pWCET estimation flow.

Figure 2 Overview of fault-aware WCET estimation methodology.

2.1 Data collection through vulnerability analysis

During the data collection step, we need to obtain the execution cycles describing the timing behaviour of the application, under transient faults occurring on the core.

To achieve that, we design a *functional and timing vulnerability analysis* and study the impact of transient faults to the functional and timing correctness of an application executed on the core. This is performed through a Cycle-Accurate Bit-Accurate (CABA) simulator, where transient faults are injected based on a given fault model at the pipeline registers of the core. In order to expose the timing impact of faults, we need to monitor any difference between the execution cycles, required for the fault-free execution, and the observed execution cycles under the presence of faults. Therefore, we remove any other source that may lead to variation of the application execution cycles [16], i.e., the application is executed in isolation, with the caches disabled and the initial state of the processor are forced to be the same among executions. Figure 2a illustrates the data collection step. Prior to any fault injection, we execute the application under study with a given set of input data without faults, in order to obtain a set of golden references: i) the application output, ii) the system state (core registers), and iii) the number of cycles required for the execution of the application with the given set of input data. Then, the core simulator is extended with fault injection capabilities in order to execute the application and to inject faults, based on the considered fault model, to the registers, while the application runs. The cycle to inject the faults is chosen randomly between the first cycle and the total number of cycles needed for the fault-free execution for the given set of data. The location, where the faults are injected, is driven by the size of the logic of each pipeline stage. The larger the area, the higher its probability to be selected. After the fault injection and upon application termination, the observed results are compared to the golden references to categorize the impact of faults as:

- *Execution Cycles Mismatch (ECM):* The execution cycles of the application are different than those of the golden reference.
- *Hang (H):* The execution time of the application has exceeded a waiting threshold, and thus, it is assumed that it has entered an infinite loop. A cycle counter is used to stop the current execution, when the counted cycles exceed the given threshold.
- *Crash (C):* The execution of the application has terminated unexpectedly and an exception has been thrown (out of bound memory access, misaligned PC, hardware trap, etc.)
- *Application Output Mismatch (AOM):* The application output is different than the golden reference.
- *Internal State Mismatch (ISM):* The system state (registers) are different than the golden reference.
- *Functionally Masked (FM):* The application has finished execution, with no AOM and no ISM.

By using the aforementioned vulnerability analysis with several random inputs, we obtain the required set of execution cycle traces under faults to be used for the WCET estimation.

2.2 Data grouping, distribution fitting and pWCET estimation

After the collection of the execution cycle traces under faults, the next step is to group the data, so as to select the tail values, perform distribution fitting and estimate the pWCET, as illustrated in Fig 2b.

To select the tail values, we use the Block Maxima (BM) approach, one of the two common methods used, along with Peak-Over-Threshold. Following the BM approach, the data collected from the vulnerability analysis are grouped into blocks of equal size. Note that, grouping of data is performed in the order the values have been collected, without applying any shuffling or sorting. Then, the maximum value is picked from each data block to obtain the BM block, to be used for the distribution fitting. The most commonly used distributions for pWCET estimation are Weibul, Gumbel and Frechet [13], and our approach currently uses the Gumbel distribution, as it is one of the most representative ones [45].

Note that, the way the data is grouped affects the distribution fitting, which affects the pWCET estimation. Selecting a big block size may result into having very few values in the BM block, while selecting a small block size may result into taking into account all the values, some of those may not be representative values as tail values. The proposed approach performs block size exploration in order to select the best representative size considering the Gumbel distribution. In order to qualify the fitting of the distribution, we use the Kolmogorov-Smirnov (KS) test to get the p-value and the ks-statistic value of BM block. The KS test compares the Cumulative Distribution Function (CDF) of the empirical data with the CDF of the theoretical distribution. The p-value tests the null hypothesis H_0 that the data came from the fitted distribution. With a significance level of $\alpha = 0.05$, the H_0 can be rejected, meaning that the data does not come from the fitter distribution, if the p-value is bellow α. However, if the p-value is higher than the significance level, the H_0 cannot be rejected. The ks-statistic value is the maximum absolute difference between the two CDFs, the smaller the value the better the fit. Thus, we select the configuration with the smallest ks-statistic value that does not reject the hypothesis of having a Gumbel distribution, as the most fitting configuration.

The selected configuration gives the distribution parameters, such as the scale (σ), the location (μ) and the shape (ξ). These values are used, along with a given threshold value, to derive the maximum value that we can observe using the Percent Point Function (PPF) (inverse of cdf – percentiles).

3 Fault-tolerant mechanism with near-zero WCET overhead

This section describes the proposed fault-tolerant mechanism with near-zero WCET overhead based on *Lock-step Execution and Shadow Register (LESR)* and reports the upper bound of the error detection and correction time.

Overview. Figure 1 illustrates the proposed LESR mechanism. Two identical cores are working in lock-step, executing the same instruction at each clock cycle. Each pipeline stage stores the result of its logic computation in a pipeline register. The error detection and correction logic is the following: in each clock cycle, we compare the pipeline registers of the two cores, containing the results of the computation of the previous cycle. If no error is detected, all the pipeline registers are copied to a BacKuP copy (BKP) and the execution continues normally (Figure 1a). Otherwise, if a fault impacted the logic during the cycle (Figure 1b) or the pipeline register itself, a wrong result is stored in the register. In this case, a flag is raised, the results of the current computation are discarded and the pipeline registers of both cores are restored with the values in BKP (Figure 1c). In this way, in the next cycle, the pipeline re-executes the cycle that was impacted by a fault (Figure 1d).

To illustrate the proposed mechanism with a simple example, let us consider the C code of listing 1. Listing 2 depicts the assembly code snippet that corresponds to the subtraction ($1 – $4), multiplication ($5 – $6) and the addition ($7 – $9) instructions, considering a RISC-V core with 5 pipeline stages, i.e., Fetch (F), Decode (D), Execute (EX), Memory (MEM) and WriteBack (WB), as the one used in our case study in Section 4.

Listing 1 C program.

```
#include <stdio.h>
int a = 10; int b = 20; int c = 0; int d = 0;
int main() {
    d = a-b;
    c = a+b*4;
    return 0;
}
```

Listing 2 Assembly code of illustration example program.

```
0001018c <main>:
                    ---
    $1 101a8:   lw   a4,a(r0)         ;load word (variable a)
    $2 101ac:   lw   a5,b(r0)         ;load word (variable b)
    $3 101b0:   sub a5,a4,a5          ;subtraction operation
    $4 101b4:   sw   a5,d(r0)         ;store word (variable d)
    $5 101b8:   lw   a5,b(r0)         ;load word (variable b)
    $6 101bc:   slli a5,a5,0x2        ;logical left shift by 2
    $7 101c0:   lw   a4,a(r0)         ;load word (variable a)
    $8 101c4:   add a5,a4,a5          ;addition
    $9 101c8:   sw   a5,c(r0)         ;store word (variable c)
                    ---
```

Table 1 (left part) shows a snapshot of the processor pipeline stages during a fault-free execution of this program. Let us suppose that, at the end of cycle $n-1$, the computation had no errors. As highlighted in the right part of Table 1, at the beginning of cycle n the pipeline registers are compared and no error is detected; hence, the content of the pipeline is

Table 1 Pipeline status for Listing 2 example.

	Fault-free execution					Execution under faults with LESR				
Pipeline stage	**n-1**	**n**	**n+1**	**n+2**	**n+3**	**n-1**	**n**	**n+1**	**n+2**	**n+3**
F	\$5	\$6	\$7	\$8	\$9	\$5	\$6	\$7	*\$6*	\$7
D	\$4	\$5	\$6	\$7	\$8	\$4	\$5	\$6	*\$5*	\$6
EX	\$3	\$4	\$5	\$6	\$7	\$3	\$4	*\$5*	*\$4*	\$5
MEM	\$2	\$3	\$4	\$5	\$6	\$2	\$3	\$4	*\$3*	\$4
WB	\$1	\$2	\$3	\$4	\$5	\$1	\$2	\$3	*\$2*	\$3

copied to the BKP registers. Let us now suppose that a transient fault impacts the D stage logic during cycle n. In cycle $n+1$, the pipeline registers of the two cores are compared and an error is detected, due to the fault in cycle n. In detail, the error is detected by comparing input registers of stage EX, which are also output registers of stage D. Thus, the results of the computations are discarded and the content of BKP is copied back. Finally – in cycle $n+2$ – the cycle impacted by the fault can be re-executed and the computations goes back to normal.

Bound WCET overhead. The LESR approach entails a constant overhead of two clock cycles, namely the cycle where the fault occurred, and the cycle where the fault is detected and the values of the core registers are restored from the BKP registers, for processors with hardware function units that require one cycle to execute the instruction. Further discussion is provided in Section 4.3.

4 Evaluation for RISC-V case study

4.1 Experimental setup

Our case study is Comet, an open-source HLS 32-bit RISC-V processor [41], which supports the RV32I base ISA[2]. Note that, by using HLS, a unique high-level synthesis and simulation C++ model is used to design the processor. The model is used to generate both the hardware target design through High-Level Synthesis, as well as a Cycle-Accurate Bit-Accurate (CABA) simulator through software compilation. The processor consists of a standard 5-stage pipeline, including a forwarding mechanism and a register file with 32 registers in the write-back stage, as illustrated in Figure 3. Table 2 depicts the area of each pipeline stage of the core.

Table 2 Area of RISC-V pipeline stages.

Pipeline stage	Fetch	Decode	Execute	Memory	WriteBack
Area	6.01%	11.02%	35.47%	5.10%	42.41%

The LESR approach has been implemented in the RISC-V CABA simulator, and it is available in `FSR_comet` branch from the Comet repository. We have enhanced both the unprotected and the protected version of the RISC-V core with hardware fault injection capabilities. The used fault model is a bit-flip. A framework based on pythons scripts has been designed in order to perform the data collection with and without fault injection, obtain

[2] https://gitlab.inria.fr/srokicki/Comet/-/tree/master

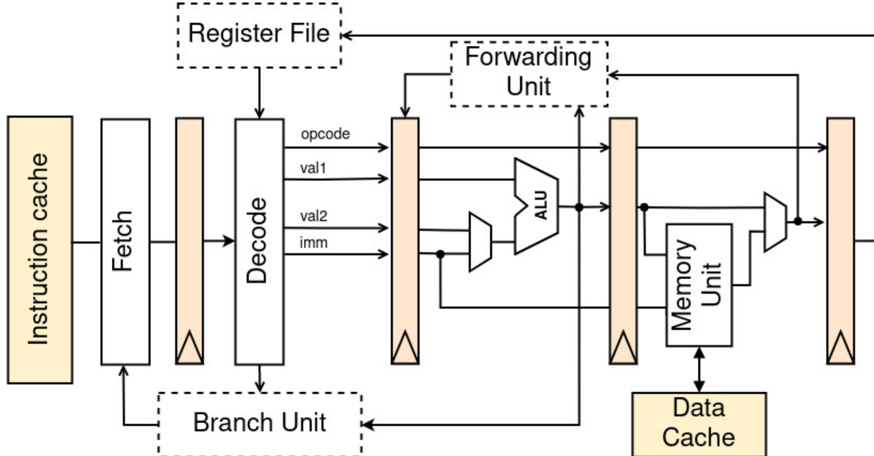


Figure 3 RISC core with 5-stage pipeline, forward mechanism, and data and instruction caches [41].

the vulnerability metrics and execution cycle traces, perform the data grouping, distribution fitting and pWCET estimation. Note that, the threshold for considering that an application is not responsive is set to eight times the execution cycles without faults.

In this first step towards the exploration of the impact on the execution time and WCET estimation of transient faults occurring inside the processor, we used as benchmarks typical kernels, applied in many application domains, such as multimedia, automotive, image processing etc. The goal is to first explore the fault impact on the kernels, before dealing with more complex applications. Five benchmarks with different complexities and execution cycles have been analyzed. More precisely, `Binary Search (BS)` searches an index in a sorted array of a size equal to 15 and `Prime` checks whether two input integers are prime or not. Both benchmarks are taken from the TACLeBench suite. `Qsort` sorts the elements of an array of size 10 and its implementation is inspired from MiBench. `Moving Average (MA)` makes the average of nearby pixels of an 8x8 matrix and is inspired from AxBench. `Matmult` multiplies two 4x4 matrices and it is taken from Polybench. The app, kernel, sequential and test benchmarks from TACLeBench, except those with floating point operations, have been successfully compiled and executed on the proposed lockstep version and fault injection campaigns will be performed in the future. The source code of the benchmarks is available in the `FSR_comet`[3] branch of the Comet repository. For the data collection step, based on [13], we use 650 different inputs for each benchmark, in order to obtain the data for the benchmark timing behavior, leading to 650 fault-free executions per benchmark. The inputs are generated by selecting each integer randomly between the integer range $[INT_{MIN}, INT_{MAX}]$, except for `Prime`, where we used positive numbers.

For the estimation under faults, note that, exhaustive fault injection is not computationally possible, due to the prohibitive number of fault injection points during the execution of an application. The different fault injection points are given by the number of different register bits of the processor and the number of cycles required for the fault-free benchmark execution. Thus, the vulnerability analysis is based on statistical fault injection, as in the the state-of-the-art approaches. The number of faults N to be injected in order to have statistically confident results is defined based on the required confidence level of the statistical analysis

[3] `https://gitlab.inria.fr/srokicki/Comet/-/tree/FSR_comet/tests/basic_tests`

as $N = \frac{t^2 \times p \times (1-p)}{e^2}$, where t is the critical value related to the statistical confidence interval, e the error margin, and p the percentage of the possible fault population individuals that are assumed to lead to errors [29, 53]. With $p = 0.5$, we obtain the maximum number of faults to be injected in order to have statistically confident results, considering infinite number of fault injection points. Based on the above formula, we have injected 250,250 faults per benchmark, which lead to results with a 99.8% confidence interval and a 0.3% error margin. More precisely, we have injected 385 faults per different input, providing 5% confidence interval and a 5% error margin [53] for each input, considering 650 different inputs. Note that, to keep the collected data independent and identically distributed, we keep the maximum clock cycle observed out of the 385 injections on every input generated to be used for the pWCET estimation.

4.2 Experimental Results

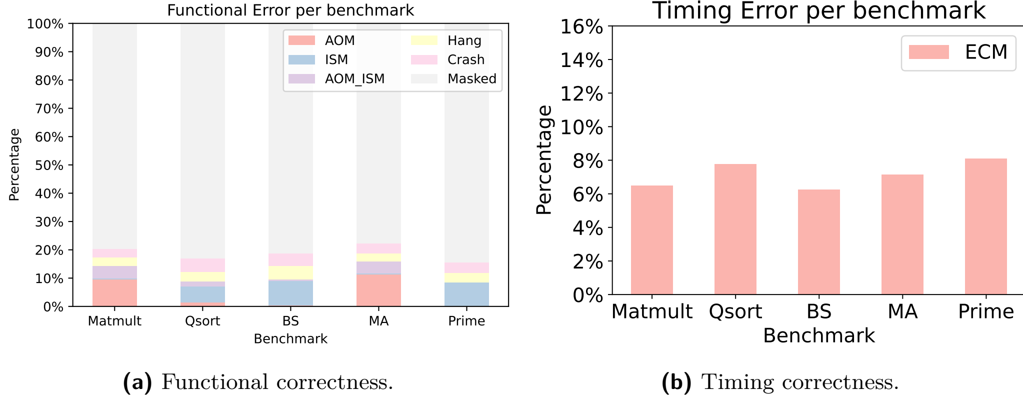
This section presents the execution cycle traces, the best selected configuration for the BM and the WCET estimation for: i) the unprotected version without faults, as currently done in the State-Of-the-Art, ii) the unprotected version under faults, and iii) the protected version using the LESR mechanism under faults. Furthermore, we provide the functional and timing vulnerability metrics, as discussed in Section 2.1, for the last two set-ups.

Table 3 shows the each vulnerability metric for the unprotected version under faults in absolute values and Figure 4 schematically illustrates the corresponding percentages. For instance, for the `Matmult` benchmark, 2.5% of the fault injections has led to application hangs, 2.72% to application crashes, 8.31% to wrong output, 0.32% to wrong internal state, 3.85% to both wrong application output and wrong internal state, and 82.28% were masked. Similar are the results for the rest of the benchmarks. On average, 3.02% of the fault injections has led to application hangs, 3.45% to application crashes, 3.95% to wrong output, 4.16% to wrong internal state, 1.99% to both wrong application output and wrong internal state, and 83.43% were masked. Regarding timing correctness, all benchmarks experienced mismatches in their number of execution cycles. More precisely, the benchmark affected the least is `Binary search`, where 6.25% of the total benchmark executions, under the presence of faults, lead to a different number of execution cycles compared to the fault-free execution. The most affected benchmark is `Prime`, where 8.10% of the benchmark executions under faults lead to ECM. On average, 7.14% of the executions under faults lead to ECM among all benchmarks. For the protected version with LESR, mechanism, all faults have been corrected.

Figures 5, 6, 7, 8 and 9 show the distribution of execution cycles for the five benchmarks. In each figure, the subfigures (a), (b) and (c) correspond to unprotected version without faults, the unprotected version with faults and the protected version with faults, respectively. For the experimental set-up with faults, the distribution shows the execution cycles for $250,250$

Table 3 Functional and timing vulnerability metrics (absolute value).

Benchmark	AOM	ISM	AOM & ISM	Hang	Crash	Masked	ECM
`Qsort`	3,284	12,573	4,216	7,424	10,501	212,252	19,429
`Prime`	107	18,868	262	7,582	8,497	214,934	20,276
`BS`	883	18,975	1,548	10,387	9,719	208,738	15,646
`Matmult`	20,800	816	9,637	6,257	6,829	205,911	16,232
`MA`	24,351	894	9,102	6,105	7,671	202,127	17,832



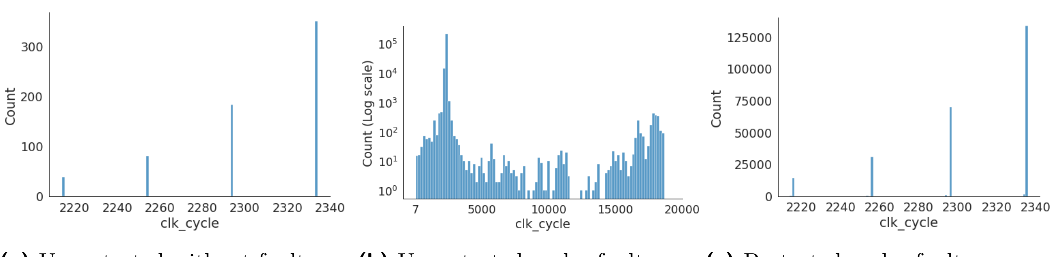
(a) Functional correctness. **(b)** Timing correctness.

Figure 4 Functional and timing Errors for the five benchmarks under study.

executions (excluding the Crash and Hangs cases for the unprotected version). Note that, for the unprotected version as the value variations are high, the histogram is presented in logarithmic scale. The overall observation among all benchmarks is that, when faults impact the unprotected core, the distribution is modified significantly, both in shape and location, as shown by Figures 5b, 6b, 7b, 8b, and 9b. Note that, the x-axis for the unprotected version under faults is significantly larger than the unprotected version without faults and the protected version with faults. Furthermore, the high peak observed in the unprotected version under faults corresponds to the execution cycles obtained for the executions where the faults have been masked.

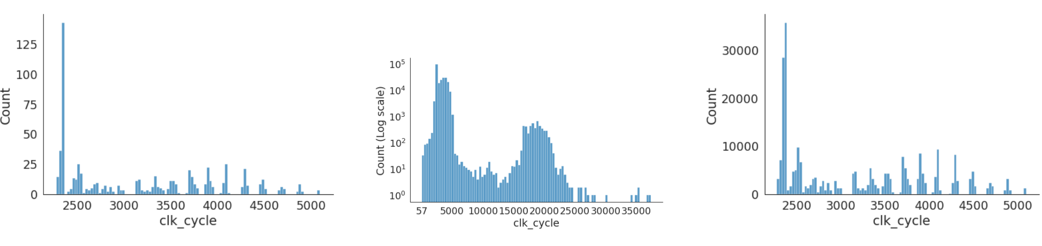
Let's further analyze this impact using the `Binary search`, which is the simplest benchmark. The execution time of binary search depends on the position of the index of the sorted array and it is upper bounded by $log_2(M)$, with M the size of the array. This statement is in line with the observations during the experiments, as `Binary search` searches in an array of 15 elements, and thus, 4 different values are observed during the 650 executions, as depicted in Fig. 5a. However, when faults are injected in the unprotected version, the distribution of collected execution traces is significantly modified. On the contrary, the protected version, using the proposed LESR mechanism, it is able to maintain a distribution very close to the original one under faults, i.e., the number of execution cycles is increased by two cycles.

To illustrate the applied methodology, Figure 10a depicts the histogram of the BM block, along with the Gumbel distribution, and Figure 11a the Quantile-Quantile plot for `Matmult` benchmark, which is one of the benchmarks with higher complexity. We observe a rather good resemblance to the line $x = y$, which means that the collected data follows the Gumbel



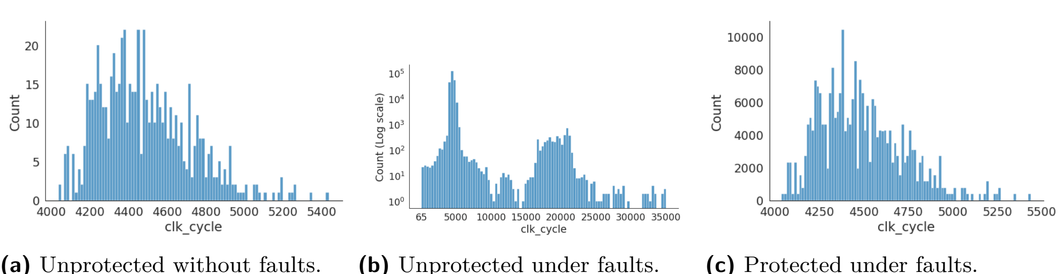
(a) Unprotected without faults. **(b)** Unprotected under faults. **(c)** Protected under faults.

Figure 5 `Binary search:` Collected data regarding execution cycles for all processor versions.



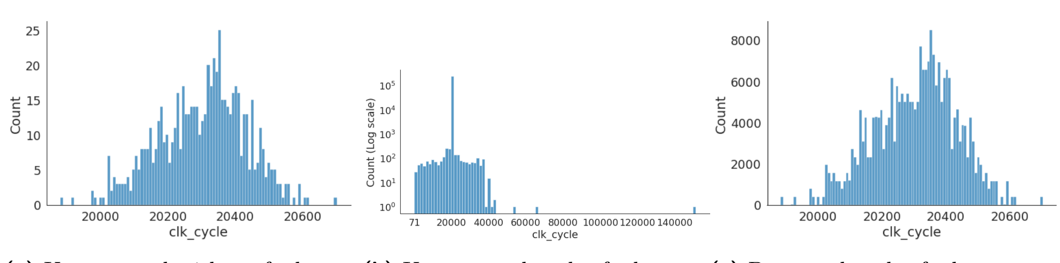
(a) Unprotected without faults. **(b)** Unprotected under faults. **(c)** Protected under faults.

Figure 6 `Prime`: Collected data regarding execution cycles for all processor versions.



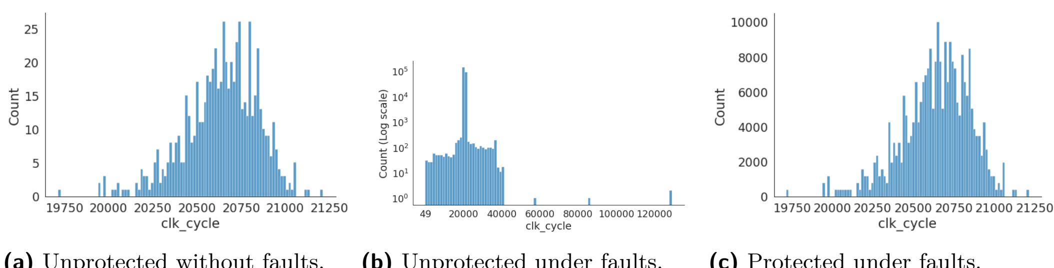
(a) Unprotected without faults. **(b)** Unprotected under faults. **(c)** Protected under faults.

Figure 7 `Qsort`: Collected data regarding execution cycles for all processor versions.



(a) Unprotected without faults. **(b)** Unprotected under faults. **(c)** Protected under faults.

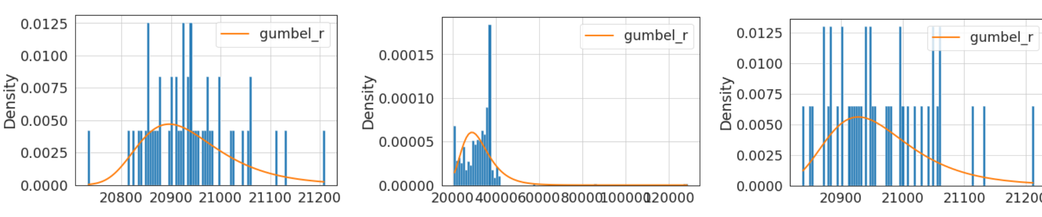
Figure 8 `Moving Average`: Collected data regarding execution cycles for all processor versions.



(a) Unprotected without faults. **(b)** Unprotected under faults. **(c)** Protected under faults.

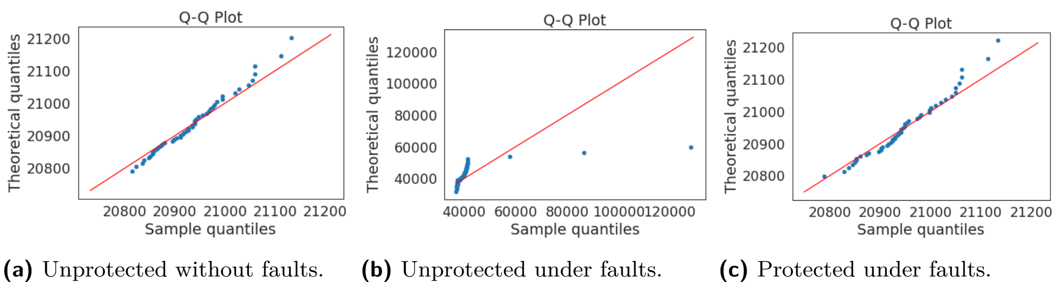
Figure 9 `Matmult`: Collected data regarding execution cycles for all processor versions.

distribution. However, when faults are injected in the unprotected version, the shape of the BM histogram is modified (Figure 10b), as shows Figure 11b. On the contrary, the protected version with LESR is able to keep the shape of the distribution similar to the fault-free distribution (Figure 10c) and obtain a similar fitting (Figure 11c). Table 4 shows the best configuration obtained during experiments.



(a) Unprotected without faults. **(b)** Unprotected under faults. **(c)** Protected under faults.

Figure 10 `Matmult`: Block Maxima and Gumbel distribution for all processor versions for the best configuration.



(a) Unprotected without faults. **(b)** Unprotected under faults. **(c)** Protected under faults.

Figure 11 `Matmul`: Q-Q plot of the distribution for all processor versions for the best configuration.

Table 4 Best BM configuration per version and benchmark.

`Benchmark`	`Binary search`	`Prime`	`Qsort`	`Moving Average`	`Matmult`
Unprotected without faults					
Number of blocks	3	108	650	44	50
Block size	217	6	1	15	13
Unprotected under faults					
Number of blocks	24	18	39	5	3
Block size	28	36	17	130	217
Protected under faults					
Number of blocks	3	81	217	50	41
Block size	217	8	3	13	16

Table 5 illustrates the pWCET estimation, using the best configuration shown in Table 4, and the maximum observed value during experiments, for all versions and benchmarks. The red (green) color highlights pWCET estimations that have a lower (higher) value than the maximum observed one. As long as the pWCET is lower than the maximum observed value, we increase the threshold until we are able to obtain an estimation higher than the maximum observed during experiments. From Table 5, we observe that typical WCET estimation approaches are able to tightly bound the unprotected version without faults.

However, when faults impact the processors, such methods provide less tight bounds with respect to the maximum observed value. This is due to the impact of the faults in the execution cycle distribution, which modifies the shape and the location. Overall, we observe that the difference between the pWCET and the maximum observed value is higher for the majority of the benchmarks. Furthermore, this difference is more significant as the complexity of the benchmark increases, e.g., as shown by the difference that progressively increases with the benchmark complexity, from `Binary Search` to `Matmult`. On the contrary, the proposed LESR protection mechanism is able to restore the execution cycle distribution close to the fault-free distribution, with an impact in the number of execution cycles equal to two. As a result, we are able to obtain pWCET estimations similar to the fault-free execution. Furthermore, the two cycles difference can be observed in the difference between the maximum observed value of the unprotected version without faults and the maximum observed value of protected version under faults.

4.3 Discussion

For the lock-step processor core that we implemented based on Comet [41], the proposed fault-tolerant mechanism entails a number of additional cycles bounded to two (one cycle to detect the fault and one to restore the correct pipeline register values), as confirmed by the experimental results. For other processor versions, the bound of two cycles will hold for similar cores, where the function units require one cycle for the instruction execution. To support a processor with the different extensions, capable of executing more complicated instructions in hardware, two approaches exist, i.e., insert a hardware function unit or implement multi-cycle operations sharing existing function units. In the first case, the proposed mitigation approach will be applied without modifications. Note that, different execution cycles will still be observed in the fault-free execution for applications that have different execution paths, which are selected based on data values. In the second case, the multi-cycle instruction is broken down into small control steps and is expressed as Finite State Machines (FSM). Each state of the FSM corresponds to a computation cycle. For instance, in the case of the multiplication, there is a state for each bit (or group of bits) in the operand. Note that, when a multi-cycle opcode enters the execution stage, the pipeline will be stalled until the FSM has reached its final state and the result is produced. We expect that this behavior will not jeopardize the fact that the proposed approach is bounded. To implement the proposed approach on a processor with a multi-cycle operation, a shadow register is required to be added in the internal register that accumulates the partial results. If the proposed approach is applied as it is, the bound is expected to increase from two cycles to the number of cycles required for the instruction, in the worst case. Therefore, there is a trade-off in the processor design between the overhead of inserting an additional hardware function unit and the overhead of the fault recovery approach.

As future work, we will leverage the proposed approach for different extensions of the RISC-V core and perform extensive fault injection campaigns for more complex applications. We expect that the results will be of similar nature, in the sense that, the more complex the application is, the more execution paths we expect to have, and thus, more execution cycle traces are expected.

Table 5 pWCET estimation (cycles) based on the best fitting configuration for different threshold value, the maximum observed cycle and their difference (%).

Benchmark		threshold 0.9	threshold 0.99	threshold 0.999	Max observed
Unprotected without faults					
Binary Search	Cycles	2,334	2,334	2,334	2,334
	Difference (%)	0	0	0	
Prime	Cycles	4,972	5,976	6,962	5,093
	Difference (%)	-2.37	17.33	36.39	
Qsort	Cycles	4,811	5,267	5,715	5,436
	Difference (%)	-11.50	-3.11	5.13	
Moving Average	Cycles	20,592	20,702	20,810	20,700
	Difference (%)	-0.52	≈ 0	0.53	
Matmult	Cycles	21,073	21,257	21,438	21,211
	Difference (%)	-0.65	0.22	1.07	
Unprotected with faults					
Binary Search	Cycles	18,696	18,826	18,953	18,671
	Difference (%)	0.13	0.83	1.51	
Prime	Cycles	34,873	45,176	55,291	37,381
	Difference (%)	-6.70	20.85	47.91	
Qsort	Cycles	34,444	43,480	52,351	35,085
	Difference (%)	-1.82	23.93	49.21	
Moving Average	Cycles	107,264	163,318	218,353	151,384
	Difference (%)	-29.14	7.88	44.24	
Matmult	Cycles	170,521	267,413	362,545	129,179
	Difference (%)	31.79	107	180	
Protected under faults					
Binary Search	Cycles	2,336	2,336	2,336	2,336
	Difference (%)	0	0	0	
Prime	Cycles	5,047	5,959	6,855	5,095
	Difference (%)	-0.94	16.95	34.54	
Qsort	Cycles	4,984	5,387	5,783	5,438
	Difference (%)	-8.35	-0.94	6.34	
Moving Average	Cycles	20,582	20,686	20,788	20,702
	Difference (%)	-0.58	-0.08	0.41	
Matmult	Cycles	21,077	21,231	21,383	21,213
	Difference (%)	-0.64	0.08	0.80	

5 Related Work

The state-of-the-art, relevant to our work, concerns i) real-time approaches for WCET estimation and fault-tolerant techniques under the presence of faults, ii) lock-step techniques, with focus on RISC-V related implementations, and iii) vulnerability analysis approaches. Table 6 summarizes the related work using the following criteria:

1. Hardware faults under study are Permanent Faults (PF) or Transient Faults (SE).
2. Hardware faults under study impact the Memory (M) or Core (C) of the target platform.

3. The Functional Behaviour (FB) or Timing Behaviour (TB) of the applications is checked. Functional behaviour refers to Denial of Service (DS), i.e., no outcome is generated because the application hanged or crashed, and to Binary Correctness (BC), i.e., the application's outcome is different than expected [40]. Timing behaviour refers to an application execution time that is different than the fault-free execution, due to a hardware fault.
4. Vulnerability analysis is performed through SoftWare (SW) or HardWare (HW) fault injection or placing the platform under Radiation Beam (RB).
5. WCET estimation assumes that hardware faults do not have a timing impact on execution, i.e., Fault-Free (FF), or not, i.e., Fault-Aware (FA).

Table 6 Summary of related work and positioning.

	Fault model		**Fault location**		**Fault detection**			**Vulnerability analysis**			**WCET estimation**	
Ref.	**PF**	**TF**	**M**	**C**	**DS**	**BC**	**TB**	**SW**	**HW**	**RB**	**FF**	**FA**
[52, 22, 1, 16, 57, 44]											√	
[14, 3, 36, 25, 12, 54]		√		√		√					√	
[5, 27, 50]		√		√	√						√	
[19]	√			√	√						√	
[35]	√	√		√	√	√					√	
[23]	√		√									√
[48]	√		√			√						√
[24, 9, 2]	√		√			√						√
[11, 10, 49]	√	√	√			√						√
[56]		√		√	√	√		√				
[33, 34, 30]		√		√		√	√	√				
[38, 55, 8, 37, 4]		√		√	√	√			√			
[28]		√		√	√	√	√		√			
[6]		√		√	√	√						
[46, 31]		√		√	√	√			√			
[59]		√	√	√	√	√				√		
[15, 58]		√	√	√	√	√			√	√		
This work		√		√	√	√	√		√			√

Regarding WCET estimation approaches, the estimation is performed through safe measurements, based on application execution, or static analysis of the programs [16]. For instance, a number of static analysis methods have been proposed, such as [52, 22], focusing on caches, and measurement-based approaches, such as [16, 44, 1]. A more detailed description of WCET estimation methods and tools is available in surveys, such as [57]. The majority of existing approaches does not consider faults, since the hardware of the target platform is assumed to be fault-free, during WCET estimation [24, 48].

To protect the system from faults, real-time approaches apply fault tolerant techniques. The faults under study usually lead to application execution failure or to erroneous outputs. To deal with these issues, the majority of real-time approaches focus on fault mitigation, through scheduling techniques applied at the task-level, such as replication of tasks [14] and task check-pointing/re-execution [19], while the fault detection is assumed to be performed by the hardware. When fault techniques are inserted to the system, their timing impact on WCET has to be taken into account, in order to still provide the timing guarantees.

To do so, existing approaches extend the fault-free WCET with the time overhead of the applied fault tolerant techniques. Works analyse this overhead by exploring how the applied fault tolerant technique impacts schedulability and providing schedulability analysis, e.g., for task replication techniques [5, 3, 36, 25] and task re-execution/check-pointing techniques [12, 27, 50]. Probabilistic worst-case schedulability analysis are also presented, e.g., for active and passive replicas [35]. Last, other works consider faults are rare events, and thus, the WCET should not consider the time overhead for recovery to avoid over-dimensioning the system, and fault recovery is modeled as an overshoot [54]. The above works can have significant time overhead, since the transient fault is detected very late, potentially after fault-free WCET bound is exceeded.

Few approaches address the impact of hardware faults on the timing behaviour of applications. Existing works focus on hardware faults in cache memories, while the rest of the architecture is assumed fault-free [7]. Approaches focus on estimating the timing impact, by accounting for the hardware degradation due to the presence of faults. For instance, static analysis probabilistically quantifies the WCET impact of permanent faults at instruction caches. The probability of an SRAM cell to be faulty is used to evaluate the additional cache misses that may occur [23]. A measurement-based approach for permanent faults occurring to caches provides the WCET impact, when cache lines are disabled due to faults [48]. Such approaches have been extended to incorporate the timing impact of inserted fault tolerant techniques to detect, correct or mitigate faults. For instance, the computation of the worst-case additional misses, due to defected cache lines, and the use of a parity bit for error detection [11]. Static probabilistic timing analysis is performed with fault detection mechanisms that periodically checks caches for faults and disable faulty cache blocks, under permanent faults [9] and also soft errors [10]. The maximum delay, introduced by error detection and correction codes, is computed in [49]. Other approaches focus on mitigating the hardware degradation, due to occurring faults, using redundant hardware. As a result, the timing impact of faults on WCET is mitigated and the timing characteristics of hardware are maintained, leading to WCET estimations close to fault-free WCET ones, despite the presence of faults. For instance, timing analysis considers a reliable victim cache to replace faulty entries [2], an extra reliable cache way per set and a shared reliable buffer [24]. Existing works mainly focus on permanent faults occurring to memories. Nonetheless, with technology size reduction, faults inside the processors cannot be considered negligible anymore [32].

Regarding vulnerability estimation approaches, existing approaches mainly focus only on estimating the functional correctness of the system under study. To achieve that, they apply fault injection at the software level and at the hardware level or put the device under radiation. Software fault injection is hardware agnostic. It is capable of flipping bits only in the data structures of the application [33, 34, 56, 30]. To improve accuracy, vulnerability analysis approaches have to consider the hardware details and perform bit-flips [38, 55, 8, 4, 37, 15, 46, 31]. Other approaches place the platform under radiation to analyze its behavior [15, 59]. However,the majority of existing vulnerability approaches focus on functional behaviour, i.e., checking for functional interruptions and erroneous values of the system under study [56, 38, 55, 8, 4, 37, 15]. However, not only the functional behaviour, but also the timing behaviour must be taken into account during vulnerability analysis for safety-critical systems. Few recent studies explore the impact of soft errors on the timing behaviour. They use software fault injection and their application domain is limited to iterative methods, e.g., the performance impact is given by the number of iterations required for iterative solvers to converge [34, 33] and their execution time [30], and hardware fault injection [28] using a single input. However, such approaches focus on the average behavior, neglecting WCET aspects.

Other fault tolerant approaches exist, which, however, do not focus on WCET aspects. Regarding lock-step execution, it can be based on *non-intrusive* and *intrusive* approaches. *Non-intrusive* approaches do not modify the processor architecture, and are typically used when the internal architecture details are hidden or difficult to modify, e.g., Commercial Off-The-Shelf (COTS) processors. For instance, lockstep approach uses ARM A9 as hard core and RISC-V as soft core [15]. Lockstep execution is achieved by inserting checkpoints in the application, where a synchronisation module is activated to check for mismatch between the status of the cores and apply roll-back. However, to perform lockstep with hard cores, processors should have specific architecture support. However, this functionality is not present on all processors [15]. *Intrusive approaches* modify internally the processor architecture. Hence, when rollback mechanisms are applied, they do not require to insert checkpoints at the application level. For instance, RISC ISA SH-2 processors and rollback are used to achieve error correction [59]. Interleaved multithreated execution is used to implement a dual lockstep approach using two virtual RISC-V cores [46]. Other approaches extend the pipeline registers with error detection and correction codes, e.g., a RISC-V core with Single Error Correction Double Error Detection (SECDED) [31]. Last, approaches triplicate components inside the RISC-V core to enhance its reliability. For instance, Control and Status Registers, Program Counter and the register file [6], FFs, LUTs, BRAMS, and DSPs [58], and the arithmetic and logic unit (ALU) are triplicated [42]. However, existing approaches do not focus on providing simple mechanisms with low WCET bounds regarding the error detection and correction time.

Compared to the state of the art, this work leverages vulnerability analysis approaches with timing correctness for transient faults occurring in processors. Through extended set of experiments, it exposes the fault impact to both functional and timing behavior of the application. Such vulnerability analysis is combined with measurement-based WCET estimation, leading to fault-aware WCET estimations. Last, a mechanism is proposed to remedy the impact of transient faults, with a bounded and near-zero timing overhead, compared to existing approaches, without the need of triplicating the complete processor.

6 Conclusion

This work leverages architectural vulnerability analysis to include not only functional correctness, but also timing correctness, under the presence of transient faults on cores. Using this analysis, we show that the number of execution cycles of an application, under the presence of transient faults, may increase significantly, compared to the fault-free execution. Through a measurement-based WCET estimation approach, we show that impact on the WCET estimation. Compared to common approaches, based on watchdog timers and re-execution with long error detection and correction time, we propose a fault tolerant technique with near-zero WCET overhead that circumvent the fault, as soon as it occurs, before being propagated and affects the execution time.

References

1. J. Abella, M. Padilla, J. Castillo, and F. Cazorla. Measurement-based worst-case execution time estimation using the coefficient of variation. *ACM Trans. Des. Autom. Electron. Syst.*, 22(4), June 2017.
2. J. Abella, E. Quiñones, F. J. Cazorla, M. Valero, and Y. Sazeides. Rvc-based time-predictable faulty caches for safety-critical systems. In *IEEE Int. On-Line Testing Symp. (IOLTS)*, pages 25–30, July 2011.

3 Z. Al-bayati, J. Caplan, B. H. Meyer, and H. Zeng. A four-mode model for efficient fault-tolerant mixed-criticality systems. In *IEEE/ACM Design, Automation Test in Europe Conf. Exhibition (DATE)*, pages 97–102, March 2016.

4 D. Asciolla, L. Dilillo, D. Santos, D. Melo, A. Menicucci, and M. Ottavi. Characterization of a risc-v microcontroller through fault injection. In *Applications in Electronics Pervading Industry, Environment and Society (APPLEPIES)*, Lecture Notes in Electrical Engineering, pages 91–101. Springer Open, 2019.

5 A. Bhat, S. Samii, and R. Rajkumar. Practical task allocation for software fault-tolerance and its implementation in embedded automotive systems. In *IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*, pages 87–98, April 2017.

6 L. Blasi, F. Vigli, A. Cheikh, A. Mastrandrea, F. Menichelli, and M. Olivieri. A RISC-V fault-tolerant microcontroller core architecture based on a hardware thread full/partial protection and a thread-controlled watch-dog timer. In *Applications in Electronics Pervading Industry, Environment and Society (APPLEPIES)*, pages 505–511, 2019.

7 F. Cazorla, L. Kosmidis, E. Mezzetti, C. Hernandez, J. Abella, and T. Vardanega. Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey. *ACM Comput. Surv.*, 52(1), February 2019.

8 C. Chang, S. Lym, N. Kelly, M. B. Sullivan, and M. Erez. Hamartia: A fast and accurate error injection framework. In *IEEE/IFIP Int. Conf. on Dependable Systems and Networks Workshops (DSN-W)*, 2018.

9 C. Chen, J. Panerati, I. Hafnaoui, and G. Beltrame. Static probabilistic timing analysis with a permanent fault detection mechanism. In *IEEE Int. Symp. on Industrial Embedded Systems (SIES)*, pages 1–10, June 2017.

10 C. Chen, J. Panerati, M. Li, and G. Beltrame. Probabilistic timing analysis of time-randomised caches with fault detection mechanisms. *IET Computers & Digital Techniques*, 13(3):129–139, 2019.

11 C. Chen, L. Santinelli, J. Hugues, and G. Beltrame. Static probabilistic timing analysis in presence of faults. In *IEEE Int. Symp. Industrial Embedded Systems (SIES)*, pages 1–10, Krakow, PL, July 2016.

12 G. Chen, N. Guan, K. Huang, and W. Yi. Fault-tolerant real-time tasks scheduling with dynamic fault handling. *Journal of Systems Architecture*, 102:101688, 2020.

13 L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quiñones, and F.J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 91–101, 2012. doi:10.1109/ECRTS.2012.31.

14 M. Cui, A. Kritikakou, L. Mo, and E. Casseau. Fault-tolerant mapping of real-time parallel applications under multiple dvfs schemes. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021.

15 Á.B. de Oliveira, G.S. Rodrigues, F.L. Kastensmidt, N. Added, E.L.A. Macchione, V.A. P. Aguiar, N.H. Medina, and M.A.G. Silveira. Lockstep dual-core arm a9: Implementation and resilience analysis under heavy ion-induced soft errors. *IEEE Trans. Nuclear Science*, 65(8):1783–1790, 2018. doi:10.1109/TNS.2018.2852606.

16 J.F. Deverge and I. Puaut. Safe measurement-based WCET estimation. In *Int. Workshop on Worst-Case Execution Time Analysis (WCET)*, 2007.

17 A. Dixit and A. Wood. The impact of new technology on soft error rates. In *Int. Reliability Physics Symp. (IRPS)*, pages 5B.4.1–5B.4.7, April 2011.

18 S. Edgar and A. Burns. Statistical analysis of wcet for scheduling. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 215–224, 2001. doi:10.1109/REAL.2001.990614.

19 G. Fohler, G. Gala, D. Gracia Pérez, and C. Pagetti. Evaluation of DREAMS resource management solutions on a mixed-critical demonstrator, January 2018.

20 Marc Gatti. Development and certification of avionics platforms on multi-core processors. In *Tutorial Mixed-Criticality Systems: Design and Certification Challenges, ESWeek*, 2013.

21 S. Hamdioui, D. Gizopoulos, G. Guido, M. Nicolaidis, A. Grasset, and P. Bonnot. Reliability challenges of real-time systems in forthcoming technology nodes. In *IEEE/ACM Design, Automation Test in Europe Conf. Exhibition (DATE)*, pages 129–134, March 2013.

22 D. Hardy and I. Puaut. Wcet analysis of multi-level non-inclusive set-associative instruction caches. In *Real-Time Systems Symp. (RTSS)*, pages 456–466, November 2008.

23 D. Hardy and I. Puaut. Static probabilistic worst case execution time estimation for architectures with faulty instruction caches. *Real-Time Systems*, 51:128–152, March 2015.

24 D. Hardy, I. Puaut, and Y. Sazeides. Probabilistic wcet estimation in presence of hardware for mitigating the impact of permanent faults. In *IEEE/ACM Design, Automation Test in Europe Conf. Exhibition (DATE)*, pages 91–96, March 2016.

25 P. Huang, H. Yang, and L. Thiele. On the scheduling of fault-tolerant mixed-criticality systems. In *ACM/EDAC/IEEE Design Automation Conf. (DAC)*, pages 1–6, June 2014.

26 E. Ibe, H. Taniguchi, Y. Yahagi, K. Shimbo, and T. Toba. Impact of Scaling on Neutron-Induced Soft Error in SRAMs From a 250 nm to a 22 nm Design Rule. *IEEE Trans. on Electron Devices*, 57(7):1527–1538, July 2010.

27 J. Kim, G. Bhatia, R. Rajkumar, and M. Jochim. Safer: System-level architecture for failure evasion in real-time applications. In *IEEE Real-Time Systems Symp. (RTSS)*, pages 227–236, December 2012.

28 A. Kritikakou, P. Nikolaou, I. Rodriguez-Ferrandez, J. Paturel, L. Kosmidis, M.K. Michael, O. Sentieys, and D. Steenari. Functional and timing implications of transient faults in critical systems. In *IEEE Int. Symp. On-Line Testing and Robust System Design (IOLTS)*, pages 1–10, 2022.

29 R. Leveugle et al. Statistical fault injection: Quantified error and confidence. In *IEEE/ACM Design, Automation Test in Europe Conference (DATE)*, pages 502–506, April 2009.

30 D. Li, J. S. Vetter, and W. Yu. Classifying Soft Error Vulnerabilities in Extreme-Scale Scientific Applications Using a Binary Instrumentation Tool. In *Int. Conf. on High Performance Computing, Networking, Storage & Analysis (SC)*, pages 1–11, November 2012.

31 J. Li, S. Zhang, and C. Bao. Duckcore: A fault-tolerant processor core architecture based on the risc-v isa. *Electronics*, 11(1), 2022. `doi:10.3390/electronics11010122`.

32 N. N. Mahatme, S. Jagannathan, T. D. Loveless, L. W. Massengill, B. L. Bhuva, S. Wen, and R. Wong. Comparison of Combinational and Sequential Error Rates for a Deep Submicron Process. *IEEE Trans. on Nuclear Science*, 58:2719–2725, December 2011.

33 B.O. Mutlu, G. Kestor, A. Cristal, O. Unsal, and S. Krishnamoorthy. Ground-truth prediction to accelerate soft-error impact analysis for iterative methods. In *Int. Conf. High Performance Computing, Data, and Analytics (HiPC)*, pages 333–344, 2019.

34 B. Ozcelik Mutlu, G. Kestor, J. Manzano, O. Unsal, S. Chatterjee, and S. Krishnamoorthy. Characterization of the Impact of Soft Errors on Iterative Methods. In *IEEE Int. Conf. on High Performance Computing (HiPC)*, pages 203–214, December 2018.

35 Risat Pathan. Real-time scheduling algorithm for safety-critical systems on faulty multicore environments. *Real-Time Systems*, September 2016.

36 R.M. Pathan. Fault-tolerant and real-time scheduling for mixed-criticality systems. *Real-Time Systems*, 50(4), July 2014.

37 J. Paturel, A. Kritikakou, and O. Sentieys. Fast Cross-Layer Vulnerability Analysis of Complex Hardware Designs. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 328–333, Limassol, Cyprus, July 2020. IEEE.

38 A. Ramos, J.A. Antonio Maestro, and P. Reviriego. Characterizing a RISC-V SRAM-based FPGA implementation against Single Event Upsets using fault injection. *Microelectronics Reliability*, 78, November 2017.

39 S. Rehman, M. Shafique, and J. Henkel. *Reliable Software for Unreliable Hardware: A Cross Layer Perspective*. Springer Publishing, 2016.

40 D. Rodopoulos, G. Psychou, M.M. Sabry, F. Catthoor, A. Papanikolaou, D. Soudris, T.G. Noll, and D. Atienza. Classification framework for analysis and modeling of physically induced reliability violations. *ACM Comput. Surv.*, 47(3), February 2015.

41 S. Rokicki, D. Pala, J. Paturel, and O. Sentieys. What You Simulate Is What You Synthesize: Designing a Processor Core from C++ Specifications. In *IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD)*. IEEE, November 2019.

42 D.A. Santos, L.M. Luza, C.A. Zeferino, L. Dilillo, and D.R Melo. A low-cost fault-tolerant risc-v processor for space systems. In *Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–5, 2020. doi:10.1109/DTIS48698.2020.9081185.

43 N. Seifert, B. Gill, S. Jahinuzzaman, J. Basile, V. Ambrose, Q. Shi, R. Allmon, and A. Bramnik. Soft Error Susceptibilities of 22 nm Tri-Gate Devices. *IEEE Trans. Nuclear Science*, 59, 2012.

44 Hardik Shah, Andrew Coombes, Andreas Raabe, Kai Huang, and Alois Knoll. Measurement based wcet analysis for multi-core architectures. In *Int. Conf. on Real-Time Networks and Systems (RTNS)*, pages 257–266, New York, NY, USA, 2014. Association for Computing Machinery.

45 K.P. Silva, L.F. Arcaro, and R. Silva De Oliveira. On using gev or gumbel models when applying evt for probabilistic wcet estimation. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 220–230, 2017. doi:10.1109/RTSS.2017.00028.

46 M.T. Sim and Y. Zhuang. A dual lockstep processor system-on-a-chip for fast error recovery in safety-critical applications. In *Conf. IEEE Industrial Electronics Society (IECON)*, pages 2231–2238, 2020.

47 S. Skalistis and A. Kritikakou. Timely fine-grained interference-sensitive run-time adaptation of time-triggered schedules. In *IEEE Real-Time Systems Symp. (RTSS)*. IEEE, 2019.

48 M. Slijepcevic, L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla. Dtm: Degraded test mode for fault-aware probabilistic timing analysis. In *Euromicro Conf. on Real-Time Systems (ECRTS)*, pages 237–248, July 2013.

49 M. Slijepcevic, L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla. Timing verification of fault-tolerant chips for safety-critical applications in harsh environments. *IEEE Micro*, 34(6):8–19, November 2014.

50 J. Song and G. Parmer. C'mon: a predictable monitoring infrastructure for system-level latent fault detection and recovery. In *IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*, pages 247–258, April 2015.

51 J. Song, J. Wittrock, and G. Parmer. Predictable, efficient system-level fault tolerance in c^3. In *IEEE Real-Time Systems Symp. (RTSS)*, pages 21–32, December 2013.

52 H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise wcet prediction by separated cache andpath analyses. *Real-Time Syst.*, 18(2/3):157–179, May 2000.

53 I. Tuzov, D. de Andrés, and J. Ruiz. Accurate Robustness Assessment of HDL Models Through Iterative Statistical Fault Injection. In *European Dependable Computing Conf. (EDCC)*, pages 1–8, September 2018.

54 G. von der Brüggen, K.H. Chen, W.H. Huang, and J.J Chen. Systems with dynamic real-time guarantees in uncertain and faulty execution environments. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 303–314, 2016.

55 N.J. Wang, A. Mahesri, and S.J. Patel. Examining ace analysis reliability estimates using fault-injection. In *Int. Symp. Computer Architecture (ISCA)*, pages 460–469, New York, NY, USA, 2007. Association for Computing Machinery.

56 J. Wei, A. Thomas, G. Li, and K. Pattabiraman. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *IEEE/IFIP Int. Conf. Dependable Systems and Networks (DSN)*, pages 375–382, 2014.

57 R. Wilhelm, J. Engblom, A. Ermedahl, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3), May 2008.

58 A.E. Wilson, M. Wirthlin, and N.G. Baker. Neutron radiation testing of risc-v tmr soft processors on sram-based fpgas. *IEEE Transactions on Nuclear Science*, pages 1–1, 2023. doi:10.1109/TNS.2023.3235582.

59 J. Yao, S. Okada, M. Masuda, K. Kobayashi, and Y. Nakashima. Dara: A low-cost reliable architecture based on unhardened devices and its case study of radiation stress test. *IEEE Trans. Nuclear Science*, 59(6):2852–2858, 2012. doi:10.1109/TNS.2012.2223715.

Optimal Multiprocessor Locking Protocols Under FIFO Scheduling

Shareef Ahmed
University of North Carolina at Chapel Hill, NC, USA

James H. Anderson
University of North Carolina at Chapel Hill, NC, USA

Abstract

Real-time locking protocols are typically designed to reduce any *priority-inversion* blocking (pi-blocking) a task may incur while waiting to access a shared resource. For the multiprocessor case, a number of such protocols have been developed that ensure *asymptotically* optimal pi-blocking bounds under job-level fixed-priority scheduling. Unfortunately, no optimal multiprocessor real-time locking protocols are known that ensure *tight* pi-blocking bounds under any scheduler. This paper presents the first such protocols. Specifically, protocols are presented for mutual exclusion, reader-writer synchronization, and k-exclusion that are optimal under first-in-first-out (FIFO) scheduling when schedulability analysis treats suspension times as computation. Experiments are presented that demonstrate the effectiveness of these protocols.

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

Keywords and phrases Real-Time Systems, Real-Time Synchronization, Multiprocessors

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.16

Supplementary Material *Software*: `https://github.com/Tamal10/fifo-lock`

Funding Supported by NSF grants CPS 1837337, CPS 2038855, CPS 2038960, and CNS 2151829, ARO grant W911NF-20-1-0237, and ONR grant N00014-20-1-2698.

1 Introduction

In recent years, a number of suspension-based multiprocessor real-time locking protocols have been developed that provide asymptotically optimal bounds on priority-inversion blocking (pi-blocking) under suspension-oblivious (s-oblivious) schedulability analysis, which treats suspension time analytically as computation time [11, 13, 14]. For mutual-exclusion (mutex) sharing, most (if not all) known asymptotically optimal locking protocols ensure a per-task s-oblivious pi-blocking bound of $2m-1$ request lengths on an m-processor platform under job-level fixed-priority (JLFP) scheduling [11, 13].[1] The commonality of this bound is somewhat surprising as these protocols include ones that target different scheduling strategies (e.g., partitioned, global, and clustered scheduling) and employ different mechanisms to cope with pi-blocking (e.g., priority inheritance vs. priority donation [11, 13]).

In contrast, under s-oblivious analysis, the current best lower bound yields a worst-case per-task pi-blocking bound of at least $m-1$ request lengths [11]. This gap between the existing lower bound and upper bound raises an obvious question: *is a pi-blocking bound of* $2m-1$ *request lengths fundamental under JLFP scheduling?*

In this paper, we answer this question negatively by showing that, under s-oblivious analysis, the existing lower bound of $m-1$ request lengths is tight under *first-in-first-out* (FIFO) scheduling. To show this, we give a suspension-based locking protocol for mutex sharing that ensures a per-lock-request s-oblivious pi-blocking bound of at most $m-1$ request

[1] We refine this statement later by distinguishing between *request blocking* and *release blocking*.

© Shareef Ahmed and James H. Anderson;
licensed under Creative Commons License CC-BY 4.0
35th Euromicro Conference on Real-Time Systems (ECRTS 2023).
Editor: Alessandro V. Papadopoulos; Article No. 16; pp. 16:1–16:21
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

lengths under FIFO scheduling, matching the lower bound. Our protocol is designed for clustered scheduling, so it can be applied under global and partitioned scheduling as well. To our knowledge, this is the first *truly optimal* suspension-based multiprocessor locking protocol under any practical scheduling algorithm.

In designing our protocol, we exploit the fact that independent (non-resource-sharing) tasks are non-preemptive under FIFO scheduling. Such non-preemptivity is a property of the scheduler itself and does not have to be otherwise enforced: under FIFO scheduling, a newly released instance of a task cannot cause any other task instance to have insufficient priority to be scheduled. Asymptotically optimal locking protocols such as the C-OMLP [13] enforce such a property via an explicit progress mechanism. We show that such mechanisms are not required under FIFO scheduling.

Our locking protocol strengthens the case for using FIFO scheduling on multiprocessors. In addition to enabling a tight pi-blocking bound, FIFO scheduling has low overheads, ensures bounded response times (and hence bounded deadline tardiness in soft real-time systems) without capacity loss [2, 22], and is *sustainable* with respect to execution times, meaning that it is safe to perform schedulability analysis assuming all instances of a task take its *worst-case execution time* (WCET) to complete. Moreover, non-preemptive execution also eases the determination of WCETs, which is challenging on modern multiprocessors [31]. According to a recent survey, around 30% of industrial respondents reported using FIFO scheduling [3].

Contributions. Our contributions are fourfold.

First, we propose a suspension-based mutex locking protocol called the *optimal locking protocol under FIFO scheduling* (OLP-F). The OLP-F restricts a task from issuing a resource request until it has high enough priority. Together with properties of FIFO scheduling, this ensures that the OLP-F has a tight s-oblivious pi-blocking bound under FIFO scheduling.

Second, we consider an extension of mutex sharing called *k-exclusion* sharing, which permits k simultaneous lock holders. For k-exclusion, we propose the *optimal locking protocol for k-exclusion under FIFO scheduling* (k-OLP-F) and show that it has a tight s-oblivious pi-blocking bound under FIFO scheduling.

Third, we expand even further beyond mutex sharing by considering *reader-writer* (RW) sharing, where exclusive resource usage is only required for write accesses and concurrent read accesses are permitted. For RW sharing, we propose the *read-optimal RW locking protocol under FIFO scheduling* (RW-OLP-F), which provides a tight s-oblivious pi-blocking bound for read requests under FIFO scheduling. Additionally, under the RW-OLP-F, the pi-blocking bound for write requests is just under two request lengths of optimal.

Finally, we provide experimental results that show the benefits of our locking protocols.

Organization. In the rest of this paper, we provide needed background (Sec. 2), delve further into s-oblivious pi-blocking (Sec. 3), establish a FIFO-based progress property for resource sharing (Sec. 4), present the above-mentioned protocols (Secs. 5–7), present our experimental results (Sec. 8), more fully review related work (Sec. 9), and conclude (Sec. 10).

2 System Model and Background

In this section, we provide needed definitions; Tbl. 1 summarizes the notation given here.

Task model. We consider a system of n sporadic *tasks* $\tau_1, \tau_2, \ldots, \tau_n$ to be scheduled on m identical processors by a FIFO scheduler. Each task τ_i releases a potentially infinite sequence of *jobs* $J_{i,1}, J_{i,2}, \ldots$. (We omit job indices if they are irrelevant.) Each task τ_i has a *period* T_i

Table 1 Notation summary.

Symbol	Meaning	Symbol	Meaning
n	Number of tasks	ℓ_q	q^{th} shared resource
m	Number of processors	N_i^q	Maximum number of requests for ℓ_q by τ_i
τ_i	i^{th} task	L_i^q	Maximum request length for ℓ_q by τ_i
$J_{i,j}$	j^{th} job of τ_i	L_{max}^q	$\max_{1\leq i\leq n}\{L_i^q\}$
T_i	Period of τ_i	L_{max}	$\max_{1\leq q\leq n_r} L_{max}^q$
C_i	WCET of τ_i	$\mathcal{R}$	A request
D_i	Relative deadline of τ_i	$r_{i,j}$	Release time of $J_{i,j}$
u_i	Utilization of τ_i	$f_{i,j}$	Finish time of $J_{i,j}$
n_r	Number of resources	$L_{sum,h}^q$	sum of the h highest L_i^q values

specifying the minimum spacing between consecutive job releases. Each task has a *relative deadline* D_i. Task τ_i has an *implicit deadline* if $D_i = T_i$, a *constrained deadline* if $D_i \leq T_i$, and an *arbitrary deadline* if no relationship between D_i and T_i is assumed. Each task has a WCET denoted C_i. Task τ_i's *utilization* is defined as $u_i = C_i/T_i$.

The *release time* (resp., *finish time*) of a job $J_{i,j}$ is given by $r_{i,j}$ (resp., $f_{i,j}$). $J_{i,j}$ is *pending* at time t if $r_{i,j} \leq t < f_{i,j}$. Jobs of a task τ_i are sequential, i.e., $J_{i,j+1}$ cannot commence execution before $J_{i,j}$ finishes. Job $J_{i,j}$ is *eligible* to execute at time t if $J_{i,j}$ is pending at time t and $t \geq f_{i,j-1}$ holds (if $j > 1$). An eligible job is either *ready* (when it can be scheduled) or *suspended* (when it cannot be scheduled).

We assume time to be discrete and a unit of time to be 1.0. All scheduling decisions are taken at integer points in time. We also assume all task parameters to be integers.

Multiprocessor scheduling. Multiprocessor scheduling approaches can be broadly classified into two categories: *partitioned* and *global*. Under partitioned scheduling, a task is statically assigned to a processor and cannot migrate to another processor. Global scheduling allows a task to execute on any of the m processors. *Clustered scheduling* is a hybrid of partitioned and global scheduling. Under clustered scheduling, all m processors are partitioned into $m/c \in \mathbb{N}$ clusters (without loss of generality, we assume m is an integer multiple of c) each containing c processors.[2] Each task is assigned to a cluster and can migrate only among the processors of the cluster. We consider clustered scheduling, as both partitioned and global scheduling are special cases ($c = 1$ and $c = m$, respectively).

Under a *job-level fixed-priority* (JLFP) scheduler, each job is assigned a fixed priority throughout its execution, but a task's priority may change over time. Common JLFP schedulers include *earliest-deadline-first* (EDF), FIFO, and *fixed-priority* scheduling algorithms. When such algorithms are employed with clustered scheduling, the c highest-priority ready jobs (if that many exist) of each cluster are scheduled on the processors of that cluster. In this paper, we consider clustered FIFO (C-FIFO) scheduling where, within a cluster, jobs with earlier release times have higher priority. We assume ties are broken arbitrarily but consistently. Hereafter, we assume all schedules to be C-FIFO unless otherwise stated.

Resource model. We consider a system that has a set $\{\ell_1, \ldots, \ell_{n_r}\}$ of shared resources. For now, we limit attention to *mutual exclusion* (mutex) sharing, although other notions of sharing will be considered later. Under mutex sharing, a resource ℓ_q can be held by at most

[2] Our results can be adapted for non-uniform cluster sizes at the expense of additional notation.

Table 2 Asymptotically optimal locking protocols for mutex locks under s-oblivious analysis.

Scheduling	Protocol	Release blocking	Request blocking
Global JLFP	OMLP [11]	0	$(2m-1)L^q_{max}$
Clustered JLFP	C-OMLP [13]	mL_{max}	$(m-1)L^q_{max}$
Clustered JLFP	OMIP [7]	0	$(2m-1)L^q_{max}$
C-FIFO	OLP-F (This work)	0	$(m-1)L^q_{max}$

one job at any time. When a job J_i requires a resource ℓ_q, it *issues* a *request* $\mathcal{R}$ for ℓ_q. $\mathcal{R}$ is *satisfied* as soon as J_i *holds* ℓ_q, and *completes* when J_i *releases* ℓ_q. $\mathcal{R}$ is *active* from its issuance to its completion. J_i must *wait* until $\mathcal{R}$ can be satisfied if it is held by another job. It may do so either by *busy-waiting* (or *spinning*) in a tight loop, or by being *suspended* by the operating system (OS) until $\mathcal{R}$ is satisfied. We assume that if a job J_i holds a resource ℓ_q, then it must be scheduled to execute.[3] A resource access is called a *critical section* (CS).

We assume that each job can request or hold at most one resource at a time, i.e., resource requests are non-nested. We let N_i^q denote the maximum number of times a job of task τ_i requests ℓ_q, and let L_i^q denote the maximum length of such a request. We define L_i^q to be 0 if $N_i^q = 0$. Finally, we define $L^q_{max} = \max_{1 \le i \le n}\{L_i^q\}$, and $L_{max} = \max_{1 \le q \le n_r}\{L^q_{max}\}$, and let $L^q_{sum,h}$ be the sum of the h largest L_i^q values. We assume all L_i^q and N_i^q to be constant.

Priority inversions. *Priority-inversion blocking* (or *pi-blocking*) occurs when a job is delayed and this delay cannot be attributed to higher-priority demand for processing time. Under a given real-time locking protocol, a job may experience pi-blocking each time it requests a resource – this is called *request blocking* – and/or upon its release and each time it releases a resource – this is called *release blocking*.

On multiprocessors, the formal definition of pi-blocking actually depends on how schedulability analysis is done. Of relevance to suspension-based locks, schedulability analysis may be either *suspension-oblivious* (*s-oblivious*) or *suspension-aware* (*s-aware*) [11]. Under s-oblivious analysis (the focus of this work), suspension time is *analytically* treated as computation time.

Blocking complexity. Request lengths are unavoidable in assessing maximum pi-blocking, as a request-issuing job may have to wait for a current resource-holder to complete before its request can be satisfied. As such, maximum pi-blocking bounds are usually expressed as an integer multiple of the maximum request length, i.e., the number of requests that are satisfied while a resource-requesting job is pi-blocked.

Asymptotically optimal locking protocols. For mutex locks, Brandenburg and Anderson established a lower bound of $m-1$ request lengths on per-request s-oblivious pi-blocking under any JLFP scheduler [11]. Thus, under s-oblivious analysis, an asymptotically optimal locking protocol achieves $O(m)$ per-job pi-blocking. Locking protocols such as the OMLP [11], the OMIP [7], and the C-OMLP [13] are asymptotically optimal under JLFP scheduling. Tbl. 2 provides a summary of existing asymptotically optimal locking protocols.[4]

[3] This is a common assumption in work on synchronization. It is needed for shared data, but may be pessimistic for other shared resources such as I/O devices.

[4] Note that, for the C-OMLP, the $2m-1$ bound mentioned in Sec. 1 comes from a combination of release and request blocking.

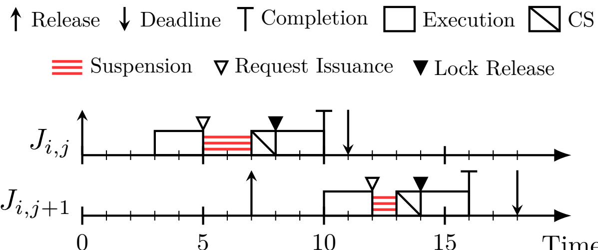


Figure 1 A schedule illustrating s-oblivious pi-blocking for arbitrary-deadline tasks.

Optimal locking protocols. We call a locking protocol *optimal* under a scheduling algorithm if it ensures a pi-blocking bound that is tight, i.e., it matches the lower bound on pi-blocking under that scheduling algorithm.

3 Suspension-Oblivious Pi-Blocking

Under s-oblivious schedulability analysis, each task's WCET is inflated by the amount of worst-case s-oblivious pi-blocking any of its jobs may suffer. Such s-oblivious pi-blocking was originally defined for implicit-deadline hard real-time systems [11]. In this section, we show that this definition needs refinement for systems with arbitrary deadlines or soft timing constraints. We also provide a refined definition that works under such cases. We begin by reviewing the original definition of s-oblivious pi-blocking under clustered scheduling.

▶ **Definition 1** ([11]). *Under s-oblivious schedulability analysis, a job J_i incurs s-oblivious pi-blocking at time t if J_i is **pending** but not scheduled and fewer than c higher-priority jobs are **pending** in its cluster.*

If tasks have arbitrary deadlines or can miss their deadlines due to soft timing constraints, Def. 1 may inappropriately identify certain delays due to the sequential execution of tasks as s-oblivious pi-blocking. The following example illustrates this.

▶ **Example 2.** Fig. 1 illustrates two consecutive jobs $J_{i,j}$, and $J_{i,j+1}$ of a task τ_i with $T_i = 7$ and $D_i = 11$. Job $J_{i,j+1}$ is released at time 7 and job $J_{i,j}$ finishes execution at time 10. Thus, job $J_{i,j+1}$ is pending but not eligible during the time interval $[7, 10)$. Assume that both $J_{i,j}$ and $J_{i,j+1}$ are among the c highest-priority pending jobs in their cluster during $[7, 10)$. Assuming $c > 1$, by Def. 1, $J_{i,j+1}$ is s-oblivious pi-blocked during the interval $[7, 10)$. However, $J_{i,j+1}$'s delay during $[7, 10)$ is not due to a locking-related suspension. Under s-oblivious schedulability analysis, it is not necessary to inflate τ_i's WCET to include such a delay. In fact, doing so may cause a circular problem, i.e., the inflated WCET may cause additional delays, which can then necessitate further inflation.

The above example motivates refining the notion of s-oblivious pi-blocking as follows.

▶ **Definition 3.** *Under s-oblivious schedulability analysis, a job J_i incurs s-oblivious pi-blocking at time t if J_i is **eligible** but not scheduled and fewer than c higher-priority jobs are **eligible** in its cluster.*

▶ **Example 2** (Cont'd). $J_{i,j+1}$ is pending but not eligible during the interval $[7, 10)$. Thus, it is not s-oblivious pi-blocked during that interval. However, $J_{i,j+1}$ is eligible during $[12, 13)$. Assume that $J_{i,j+1}$ is among the c highest-priority eligible jobs during $[12, 13)$, but is suspended. Then, by Def. 3, $J_{i,j+1}$ is s-oblivious pi-blocked during $[12, 13)$.

4 Resource-Holder's Progress Under FIFO Scheduling

Any real-time locking protocol needs to ensure a resource-holding job's progress whenever a job waiting for the same resource is pi-blocked, for otherwise, the maximum per-job pi-blocking can be very large or even unbounded. To ensure that the maximum pi-blocking is reasonably bounded, real-time locking protocols employ *progress mechanisms* that may temporarily raise a job's *effective priority*. One such mechanism is *priority inheritance* [26,28], which raises the effective priority of a job holding resource ℓ_q to the maximum of its priority and the priorities of all jobs waiting for ℓ_q. Another alternative is *priority donation* [14], which ensures that a job J_i can only issue a request when its priority is high enough to be scheduled. Moreover, if a job J_j's release causes J_i to have insufficient priority to be scheduled, then J_j "donates" its priority to J_i. This ensures that a resource holder is always scheduled. This property makes priority donation particularly effective under clustered scheduling.

Progress under FIFO scheduling. The above-mentioned progress mechanisms can be utilized to design multiprocessor locking protocols that are asymptotically optimal under any JLFP scheduling policy [11,14]. Interestingly, for the case of C-FIFO scheduling, no such progress mechanism is required to design optimal locking protocols. In fact, the C-FIFO scheduling policy itself has properties that ensure the progress of a resource-holding job. The key property that enables such progress is given in the following lemma.

▶ **Lemma 4.** *Under C-FIFO scheduling, if a job $J_{i,j}$ becomes one of the c highest-priority eligible jobs in its cluster at time t_h, then it remains so during $[t_h, f_{i,j})$.*

Proof. Assume for a contradiction that t is the first time instant in $[t_h, f_{i,j})$ such that $J_{i,j}$ is not one of the c highest-priority eligible jobs in its cluster. Then, $t > t_h$ holds. By the definition of time t, there are at most $c-1$ (resp., at least c) eligible jobs with higher priority than $J_{i,j}$ at time $t-1 \geq t_h$ (resp., t) in $J_{i,j}$'s cluster. Thus, there is a task τ_u that has an eligible job $J_{u,v}$ with higher priority than $J_{i,j}$ at time t, but has no such job at time $t-1$.

Since $J_{u,v}$'s priority exceeds $J_{i,j}$'s, $r_{u,v} \leq r_{i,j}$ holds. Since $J_{i,j}$ is eligible at time t_h, $r_{i,j} \leq t_h$ holds. Thus, $r_{u,v} \leq t_h$ and $J_{u,v}$ is pending at time $t-1$. We now consider two cases.

Case 1: $v = 1$. In this case, $J_{u,v}$ is also eligible at time $t-h$. Thus, τ_u has an eligible job with higher priority than $J_{i,j}$ at time $t-1$, a contradiction.

Case 2: $v > 1$. Since $J_{u,v}$ is not eligible at time $t-1$, job $J_{u,v-1}$ is eligible at time $t-1$. We have $r_{u,v-1} < r_{u,v} \leq r_{i,j}$. Thus, τ_u has an eligible job with higher priority than $J_{i,j}$ at time $t-1$, a contradiction.

Therefore, we reach a contradiction in both cases. ◀

Utilizing Lemma 4, we have the following lemma.

▶ **Lemma 5.** *If a job $J_{i,j}$ issues a request $\mathcal{R}$ when it is one of the c highest-priority jobs in its cluster, then $J_{i,j}$ is always scheduled from $\mathcal{R}$'s satisfaction to completion.*

Proof. Let t_r, t_s, and t_c be the time instants when $\mathcal{R}$ is issued, satisfied and complete, respectively. Thus, $t_r \leq t_s \leq t_c$ holds. Since $J_{i,j}$ is one of the c highest-priority eligible jobs in its cluster at time t_r, by Lemma 4, $J_{i,j}$ remains one of the c highest-priority eligible jobs in its cluster throughout $[t_r, t_c)$. Since $\mathcal{R}$ is satisfied at time $t_s \geq t_r$, $J_{i,j}$ is ready throughout $[t_s, t_c)$. Thus, $J_{i,j}$ is scheduled during $[t_s, t_c)$. ◀

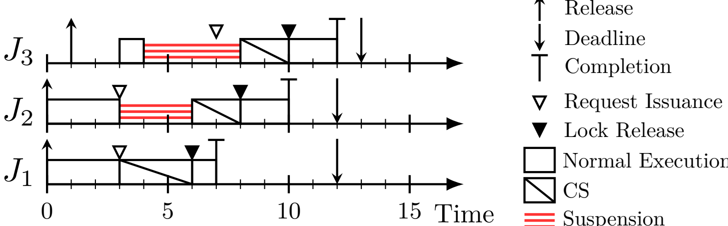


Figure 2 A schedule illustrating the OLP-F.

Thus, by requiring a request to be issued only when the request-issuing job is one of the top-c-priority jobs in its cluster, we can ensure a resource-holder's progress under FIFO scheduling. We exploit this property in designing our protocols. Note that the C-OMLP ensures this property by employing priority donation as its progress mechanism at the expense of additional release blocking that may be incurred by a job even if it does not require any resource [13]. Due to this, our protocols have features in common with the C-OMLP.

5 Mutex Locks

In this section, we introduce the *optimal locking protocol for mutual exclusion sharing under C-FIFO scheduling* (OLP-F), which achieves optimal pi-blocking under C-FIFO scheduling. To match the lower bound on pi-blocking, the OLP-F ensures that each job suffers pi-blocking for the duration of at most $m - 1$ request lengths and incurs no release blocking.

Structures. For each resource ℓ_q, we have a FIFO queue FQ_q that contains requests for ℓ_q. A request $\mathcal{R}$ is satisfied if and only if $\mathcal{R}$ is the head of the FQ_q.

Rules. When a job J_i attempts to issue a request $\mathcal{R}$ for a resource ℓ_q, it proceeds according to the following rules.

M1 J_i is allowed to issue $\mathcal{R}$ only if it is one of the c highest-priority eligible jobs in its cluster. J_i suspends if necessary to ensure this condition.

M2 When J_i issues $\mathcal{R}$, $\mathcal{R}$ is enqueued in FQ_q. If J_i becomes the head of FQ_q, then it is immediately satisfied. Otherwise, it suspends.

M3 $\mathcal{R}$ is satisfied when it is the head of FQ_q. $\mathcal{R}$ is removed from the FQ_q when it is complete.

▶ **Example 6.** Fig. 2 illustrates a C-FIFO schedule of three jobs on a two-processor cluster. J_1 and J_2 are released earlier (hence, have higher priorities) than J_3. Both J_1 and J_2 issue requests for resource ℓ_q at time 3 and J_1's request is enqueued first. Assuming no job in a different cluster holds ℓ_q, J_1 acquires ℓ_q at time 3 by Rule M2. At time 3, since J_2 is suspended, J_3 starts to execute. At time 4, J_3 attempts to issue a request for ℓ_q, but it is suspended due to Rule M1 as it is not one of the top-2-priority jobs at that time. At time 6, J_1 releases ℓ_q and J_2's request is satisfied according to Rule M3. Since J_3 becomes one of the top-2-priority jobs when J_1 completes, it issues a request for ℓ_q at time 7.

Analysis. To derive an upper bound on the pi-blocking suffered by a job, we first show that FQ_q contains no more than m requests at any time.

▶ **Lemma 7.** *Under the OLP-F, at any time, FQ_q contains at most m requests.*

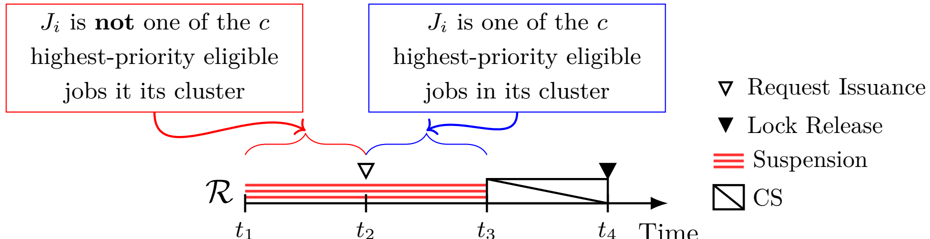


Figure 3 Timeline of a request under the OLP-F.

Proof. Assume that t is the first time instant when FQ_q contains more than m requests. Each job has at most one active request at any time. Thus, at time t, FQ_q must contain a request $\mathcal{R}$ issued by a job J_i that is not one of the c highest-priority eligible jobs in its cluster. Let $t' \leq t$ be the time instant when J_i issues $\mathcal{R}$. By Rule M1, J_i is one of the c highest-priority eligible jobs in its cluster at time t'. Since J_i is not complete at time t, by Lemma 4, it is one of the c highest-priority eligible jobs in its cluster at time t, a contradiction. ◀

We now determine an upper bound on the request blocking suffered by job J_i when it issues a request $\mathcal{R}$ for resource ℓ_q. Fig. 3 depicts the timeline of $\mathcal{R}$ from when J_i attempts to issue $\mathcal{R}$ to when $\mathcal{R}$ completes. Let t_1 be the time instant when job J_i attempts to issue request $\mathcal{R}$. Let t_2 be the first time instant at or after time t_1 when J_i, becomes one of the top-m-priority eligible jobs. Therefore, by Rule M1, $\mathcal{R}$ is issued at time t_2. Let t_3 and t_4 be the time instants when $\mathcal{R}$ is satisfied and completes, respectively.

▶ **Lemma 8.** *During $[t_1, t_3]$, J_i incurs pi-blocking for at most $L^q_{sum,m-1}$ time units.*

Proof. By the definition of t_2, J_i is not one of the top-c-priority eligible jobs in its cluster during $[t_1, t_2)$. Hence, J_i is not pi-blocked during that time. By Lemma 4, J_i is pi-blocked throughout $[t_2, t_3)$. By Lemma 5, J_i is continuously scheduling during $[t_3, t_4)$. Thus, from t_1 to t_4, J_i is only pi-blocked during $[t_2, t_3)$.

By Lemma 7, at most $m - 1$ other requests precede $\mathcal{R}$ in FQ_q at time t_2. By Rule M3 and Lemma 5, each job at the head of FQ_q is continuously scheduled until its request is complete. Since each task has at most one eligible job and each job has at most one request at any time, $t_3 - t_2$ is not more than $L^q_{sum,m-1}$ time units and the lemma follows. ◀

We now show that the OLP-F does not cause any release blocking under C-FIFO scheduling.

▶ **Lemma 9.** *Under the OLP-F, no job incurs release blocking.*

Proof. Since a resource-holding job is scheduled only when its priority is among the top c in its cluster, a resource request $\mathcal{R}$ does not cause pi-blocking to any job (within and across cluster boundaries) that does not issue a request during the time $\mathcal{R}$ is satisfied. ◀

▶ **Theorem 10.** *Under the OLP-F, J_i is pi-blocked for at most $b_i = \sum_{q=1}^{n_r} N_i^q \cdot L^q_{sum,m-1}$ time units.*

Proof. Follows from Lemmas 8 and 9. ◀

Thus, the OLP-F is an optimal locking protocol under C-FIFO scheduling.

Table 3 Asymptotically optimal locking protocols for k-exlcusion locks under s-oblivious analysis.

Scheduling	Protocol	Release blocking	Request blocking
Clustered JLFP	CK-OMLP [11]	$\max_q\{\lceil m/k_q\rceil L^q_{max}\}$	$(\lceil m/k_q\rceil - 1)L^q_{max}$
Global JLFP	OKGLP [18]	0	$(2\lceil m/k_q\rceil + 4)L^q_{max}$
Global JLFP	R^{2}DGLP [30]	0	$(2\lceil m/k_q\rceil - 2)L^q_{max}$
C-FIFO	k-OLP-F (This work)	0	$(\lceil m/k_q\rceil - 1)L^q_{max}$

6 k-Exclusion Locks

k-exclusion generalizes mutual exclusion by allowing up to k simultaneous lock holders; thus, mutual exclusion is equivalent to 1-exclusion. In this section, we give an optimal k-exclusion locking protocol under C-FIFO scheduling. We assume that a resource ℓ_q can be concurrently held by up to $k_q \leq m$ jobs. We begin by reviewing lower bound results for k-exclusion.

Lower bound on pi-blocking. For k-exclusion, Elliot et al. showed that a task system and a release sequence for it exist such that a job requesting a resource ℓ_q incurs s-oblivious pi-blocking for the duration of $\lceil \frac{m-k_q}{k_q} \rceil$ request lengths under any JLFP scheduler [18].

Asymptotically optimal locking protocols. Under s-oblivious analysis, the CK-OMLP [11], the OKGLP [18], and the R^{2}DGLP [30] ensure asymptotically optimal pi-blocking for k-exclusion. Tbl. 3 summarizes these protocols.

The k-OLP-F. We now introduce the *optimal locking protocol for k-exclusion under C-FIFO scheduling* (k-OLP-F), which achieves optimal pi-blocking for k-exclusion under C-FIFO scheduling. The k-OLP-F ensures that a job suffers pi-blocking for the duration of no more than $\lceil \frac{m-k_q}{k_q} \rceil$ request lengths for each request for ℓ_q and incurs no release blocking.

Structures. For each resource ℓ_q, we have a FIFO queue FQ_q that contains *waiting* requests for ℓ_q. We also have a queue SQ_q of length at most k_q that contains the *satisfied* requests for ℓ_q. Initially, both queues are empty. A request $\mathcal{R}$ is satisfied if and only if $\mathcal{R}$ is in SQ_q.

Rules. When a job J_i attempts to issue a request $\mathcal{R}$ for a resource ℓ_q, it proceeds according to the following rules.

K1 J_i is allowed to issue $\mathcal{R}$ only if J_i is one of the c highest-priority eligible jobs in its cluster. J_i suspends if necessary to ensure this condition.

K2 If the length of SQ_q is less than k_q when J_i issues $\mathcal{R}$, then $\mathcal{R}$ is enqueued in SQ_q and is immediately satisfied. Otherwise, $\mathcal{R}$ is enqueued in FQ_q and J_i suspends.

K3 When $\mathcal{R}$ completes, it is removed from SQ_q. If FQ_q is non-empty at that time, then the head of FQ_q is dequeued, enqueued in SQ_q, and satisfied.

▶ **Example 11.** Fig. 4 shows a schedule of five jobs that share a resource ℓ_q with $k_q = 2$. Jobs J_1, J_2, and J_3 (resp., J_4, and J_5) are FIFO scheduled on a two-processor cluster G_1 (resp., G_2). Since SQ_q is initially empty, by Rule K2, J_4 and J_1 acquire ℓ_q at times 2 and 3, respectively. Since both J_2 and J_5 are one of the top-2-priority eligible jobs in their clusters, by Rule K1, they issue requests for ℓ_q at times 4 and 5, respectively. At time 5, J_3 attempts to issue a request for ℓ_q, but is suspended, by Rule K1. At time 5, J_4 releases ℓ_q and is removed from SQ_q by Rule K3. J_2's request is at the head of FQ_q at time 5, so by Rule K3, it is removed from FQ_q, enqueued in SQ_q, and satisfied. At time 7, J_1 completes and J_3 becomes one of the top-2-priority jobs in G_1 and issues its request, by Rule K1.

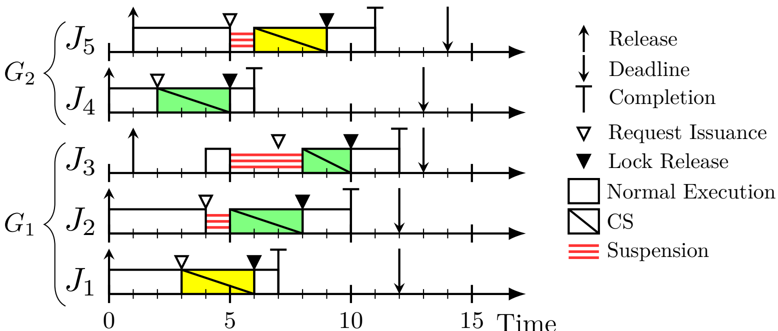


Figure 4 A schedule illustrating the k-OLP-F. Concurrent resource accesses are shaded differently.

Analysis. We now derive an upper bound on the pi-blocking suffered by a job under the k-OLP-F. We first derive an upper bound on the number of waiting requests in FQ_q.

▶ **Lemma 12.** *Under the k-OLP-F, FQ_q contains at most $m - k_q$ requests.*

Proof. Assume otherwise. Let t be the first time instant such that FQ_q contains more than $m - k_q$ requests. Thus, a new request $\mathcal{R}'$ is enqueued in FQ_q at time t. By Rule K2, SQ_q contains k_q requests at time t. Thus, the number of active requests (either satisfied or waiting) is more than $k_q + m - k_q = m$ at time t. Since each job has at most one active request at any time, there is an active request $\mathcal{R}$ issued by a job J_i that is not one of the c highest-priority jobs in its cluster. By Rule K1, J_i is one of the c highest-priority jobs in its cluster when it issues $\mathcal{R}$ at time $t' \leq t$. By Lemma 4, J_i remains as one of the c highest-priority jobs in its cluster at time t, a contradiction. ◀

We now determine an upper bound on request blocking. We consider a job J_i that issues a request $\mathcal{R}$ for resource ℓ_q. As in Fig. 3, let t_1, t_2, t_3, and t_4 be the time instants corresponding to when J_i attempts to issue $\mathcal{R}$, and when $\mathcal{R}$ is issued, satisfied, and complete, respectively.

▶ **Lemma 13.** *For request $\mathcal{R}$, J_i suffers request blocking for at most $L^q_{sum,\lceil \frac{m-k_q}{k_q} \rceil}$ time units.*

Proof. By Def. 3, J_i does not suffer any pi-blocking during $[t_1, t_2)$ and $[t_3, t_4)$. By Lemma 4 and the definition of t_2, J_i suffers pi-blocking during the entire duration of $[t_2, t_3)$, so it suffices to upper bound $(t_3 - t_2)$. If SQ_q contains fewer than k_q requests at time t_2, then $t_3 - t_2 = 0$ holds by Rule K2, so assume otherwise. At time t_2, no two requests in SQ_q and FQ_q are from the same task. By Rule K3, $\mathcal{R}$ is satisfied when it is dequeued from FQ_q. Thus, by Lemma 12, at most $m - k_q$ requests are required to be dequeued to satisfy $\mathcal{R}$. By Rule K2, k_q jobs hold ℓ_q throughout $[t_2, t_3)$. By Rule K1 and Lemma 5, each resource-holding job is always scheduled. Thus, per $L^q_{sum,h}$ time units during $[t_2, t_3)$ at least $h \cdot k_q$ requests complete – and hence, by Rule K3, at least $h \cdot k_q$ requests are dequeued from FQ_q. Dequeuing $m - k_q$ requests from FQ_q thus requires at most $L^q_{sum,\lceil \frac{m-k_q}{k_q} \rceil}$ time units, so $t_3 - t_2 \leq L^q_{sum,\lceil \frac{m-k_q}{k_q} \rceil}$. ◀

Similar to the OLP-F, no release blocking occurs under the k-OLP-F. Therefore, by Lemma 13, we have the following theorem.

▶ **Theorem 14.** *Under the k-OLP-F, J_i suffers pi-blocking for at most $b_i = \sum_{q=1}^{n_r} N_i^q \cdot L^q_{sum,\lceil \frac{m-k_q}{k_q} \rceil}$ time units.*

Thus, the k-OLP-F is optimal for k-exclusion locking under C-FIFO scheduling.

7 Reader-Writer Locks

Some resources can be read without alteration. For such resources, it may be desirable to support *reader-writer* (RW) sharing. Here, *writers* have mutually exclusive access to the resource, but multiple *readers* can access the resource simultaneously.

Under RW sharing, it is often desirable to ensure fast read access. However, enabling fast read access may cause write requests to starve. This can happen under a *read-preference* RW lock that never satisfies a write request if a read request is active. More generally, these observations give rise to an important question: what is the minimum request blocking a read request can incur without causing a write request to starve?

Lower bound on read request blocking. As we show next, ensuring a read request delay of $2L^q_{max} - 2$ time units can in fact cause writer starvation.

▶ **Theorem 15.** *For $m \geq 8$, a task system and a release sequence for it exist such that any locking protocol that ensures request blocking of at most $2L^q_{max} - 2$ time units for read requests causes unbounded request blocking for write requests under any work-conserving scheduler.*

Proof. We give an example task system Γ and a release sequence for it supporting the claim. Let $\tau_1, \tau_2, \ldots, \tau_m$ be m sporadic tasks scheduled on m processors. All tasks have WCETs of $L+1$ time units with $2 \leq L \leq (m-2)/3$. Fig. 5 illustrates this for $m = 8$ and $L = 2$. Each job's execution consists of 1.0 time unit of non-CS execution followed by L time units of CS execution. Tasks $\tau_1, \tau_2, \ldots, \tau_{m-1}$ issue read requests for resource ℓ_q, while τ_m issues a write request for ℓ_q. The periods of all tasks are $m-1$. Each task has an implicit deadline.

Feasibility of Γ. We show that Γ is feasible under a *write-preference* RW lock. Such lock does not satisfy any read request if a write request is waiting. Since τ_m is the only writer task, under a write-preference RW lock, τ_m's jobs acquire ℓ_q immediately (if no reader jobs hold ℓ_q) or immediately after the currently satisfied read requests complete (otherwise). Thus,

E each of τ_m's jobs acquires ℓ_q within L time units of its request issuance.

Since there are m tasks, a processor is always available for τ_m. Thus, with a WCET of $L+1$ and resource acquisition time of at most L, each job of τ_m completes within $L+1+L = 2L+1 \leq 2(m-2)/3+1 < m-2+1 = m-1 = T_m$ time units after its release.

For reader tasks $\tau_1, \tau_2, \ldots, \tau_{m-1}$, a read request $\mathcal{R}$ issued at time t is satisfied immediately if there is no waiting write request. Otherwise, by (E), the pending write request by τ_m's job is satisfied by time $t+L$ and complete by time $t+L+L = t+2L$ (as a processor is available). Since τ_m is the only writer task, after completion of the write request, there is no pending write request. Thus, $\mathcal{R}$ is satisfied by time $t+2L$. With a WCET of $L+1$, the job issuing $\mathcal{R}$ completes within $L+1+2L = 3L+1 \leq 3(m-2)/3+1 = m-2+1 = m-1 = T_i$ time units after its release. Therefore, Γ is feasible.

Release sequence for Γ. τ_m releases its jobs periodically from time 1. τ_1 releases its first job at time 0 and its subsequent jobs' release times are defined as $r_{1,j+1} = f_{m-1,j} - L$. The release times of τ_i's jobs with $2 \leq i < m$ are $r_{i,j} = f_{i-1,j} - L$. Thus, for $2 \leq i < m$, we have

$$\begin{aligned} r_{i,j} &= f_{i-1,j} - L \\ &\geq \{\text{Since } J_{i-1,j} \text{ executes for } L+1 \text{ time units}\} \\ &\quad r_{i-1,j} + L + 1 - L \\ &= \ r_{i-1,j} + 1. \end{aligned} \tag{1}$$

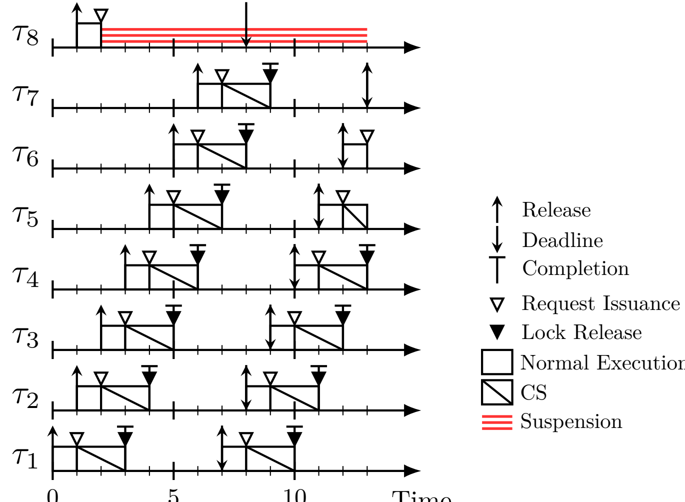


Figure 5 A schedule illustrating Theorem 15.

Similarly, for τ_1, it can be shown that

$$r_{1,j+1} \geq r_{m-1,j} + 1. \tag{2}$$

We now show that consecutive jobs of τ_i with $i < m$ are released at least T_i time units apart. For $2 \leq i < m$, by (1), we have

$$\begin{aligned} r_{i,j+1} &\geq r_{i-1,j+1} + 1 \\ &\geq \{\text{Applying (1) repeatedly for } i-2 \text{ times}\} \\ &\quad r_{1,j+1} + 1 + (i-2) \\ &\geq \{\text{By (2)}\} \\ &\quad r_{m-1,j} + 1 + (i-1) \\ &\geq \{\text{Applying (1) repeatedly for } m-1-i \text{ times}\} \\ &\quad r_{i,j} + (m-1-i) + i \\ &= r_{i,j} + m - 1 \\ &= r_{i,j} + T_i. \end{aligned} \tag{3}$$

Similarly, we can show that consecutive jobs of τ_1 are released at least T_1 time units apart.

We now show that each job of τ_i with $i < m$ is eligible when it is released by showing that $J_{i,j}$ completes before $J_{i,j+1}$'s release. For $2 \leq i < m-1$, in the third step of the derivation of (3), applying (1) repeatedly for $m-2-i$ times instead of $m-1-i$ times, we have $r_{i,j+1} \geq r_{i+1,j} + (m-2-i) + i = r_{i+1,j} + m - 2$. Since $L \leq (m-2)/3 < m-2$ and $r_{i+1,j} = f_{i,j} - L$, we get $r_{i,j+1} > r_{i+1,j} + L = f_{i,j}$. For $i = m-1$, the first step in the derivation of (3) yields $r_{m-1,j+1} \geq r_{1,j+1} + 1 + (m-1-2) = r_{1,j+1} + m - 2 > r_{1,j+1} + L$. Since $r_{1,j+1} = f_{m-1,j} - L$, we get $r_{m-1,j+1} > f_{m-1,j}$. For $i = 1$, applying (1) in (2) repeatedly for $m-3$ times, we have $r_{1,j+1} \geq r_{2,j} + m - 2 > r_{2,j} + L = f_{1,j}$. Thus, $r_{i,j+1} > f_{i,j}$ for $i < m$.

Finishing up. We now prove the theorem by showing that $J_{m,1}$'s write request is never satisfied if the request delay for read requests is at most $2L-2$. Assume that $J_{m,1}$'s request is satisfied at time t. We have $t > 2$, as $J_{m,1}$ issues its request at time 2 and $J_{1,1}$ holds ℓ_q then (under a work-conserving scheduling policy, $J_{1,1}$ acquires ℓ_q at time 1). Since the scheduling policy is work-conserving, a job $J_{i,j}$ must release ℓ_q at time t. Thus, $f_{i,j} = t$.

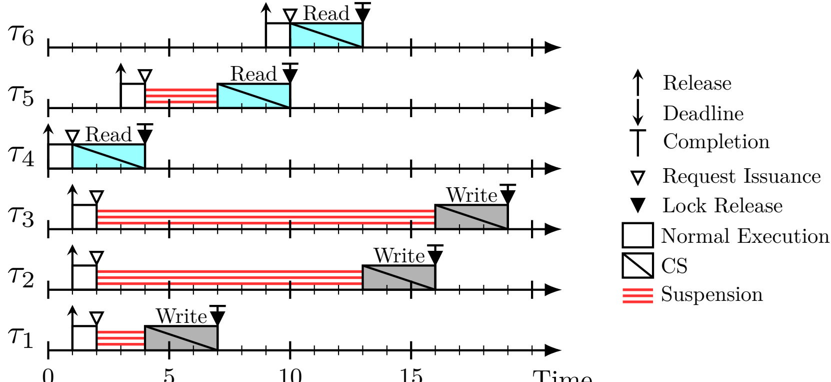


Figure 6 A schedule illustrating Theorem 16. Read and write CSs are shaded differently.

By the job release pattern of $\tau_1, \tau_2, \ldots, \tau_{m-1}$, there exists a job $J_{u,v}$ such that $r_{u,v} = f_{i,j} - L = t - L$. Since each job is eligible when it is released and there are m tasks, $J_{u,v}$ issues a read request $\mathcal{R}$ at time $r_{u,v} + 1 = t - L + 1 < t$ (as $L \geq 2$). Since $J_{m,1}$'s write request is satisfied at time t, $\mathcal{R}$ cannot be satisfied before time $t + L$. Since the task count is m, $J_{u,v}$ is pi-blocked for a duration of at least $t + L - (t - L + 1) = 2L - 1$ time units. Thus, request blocking for read requests exceeds $2L - 2$ time units, reaching a contradiction. ◀

Thus, read request blocking of at least $2L^q_{max} - 1$ time units is fundamental to avoid writer starvation. We now establish a lower bound on write request blocking when read requests suffer request blocking for at most $2L^q_{max} - 1$ time units.[5]

▶ **Theorem 16.** *For $m \geq 4$, there exists a task system and a release sequence for it such that any locking protocol that ensures at most $2L^q_{max} - 1$ read request blocking causes write request blocking of $(2m - 5)L^q_{max} - 1$ time units under any work-conserving scheduler.*

Proof. Let $\tau_1, \tau_2, \ldots, \tau_n$ be n tasks scheduled on $m \geq 4$ processors, where $n = 2m - 4$. Each task has a WCET of $L + 1$ time units with $L \geq 1$. Fig. 6 illustrates this for $m = 5$ and $L = 3$. Each job's execution consists of 1.0 time unit of non-CS execution followed by L time units of CS execution. Tasks $\tau_1, \tau_2, \ldots, \tau_{m-2}$ issue write requests for resource ℓ_q, while $\tau_{m-1}, \tau_m, \ldots, \tau_{2m-4}$ issue read requests for ℓ_q. Each task's period is $T \geq (2m - 4) \cdot (L + 1)$. The task WCETs sum to $(2m - 4) \cdot (L + 1)$, so assuming implicit deadlines, the task system can be scheduled by sequentially executing the jobs on a single processor (i.e., it is feasible).

Tasks $\tau_1, \tau_2, \ldots, \tau_{m-2}$ release their first jobs at time 1. Task τ_{m-1} releases its first job at time 0. For $i > m - 1$, the release time of $J_{i,1}$ is determined as $r_{i,1} = f_{i-1,1} - 1$. Hence, from time 0, there is always an eligible first job of a task until all first jobs are complete. Since all WCETs sum to $(2m - 4) \cdot (L + 1)$, under a work-conserving scheduler, the first job of each task completes by time $(2m - 4) \cdot (L + 1) \leq T$. Subsequent job release times can be easily defined so that each task's consecutive job releases are at least T time units apart.

We now prove that each first job $J_{i,1}$ always incurs pi-blocking when it is waiting for ℓ_q. For any job $J_{i,1}$ with $i > m$, we have $r_{i,1} = f_{i-1,1} - 1 \geq r_{i-1,1} + L + 1 - 1 = f_{i-2,1} - 1 + L$. Since $L \geq 1$, we have $r_{i,1} \geq f_{i-2,1}$. Thus, at most two first jobs of the last $m - 2$ tasks are pending at the same time. Therefore, at most $m - 2 + 2 = m$ first jobs are pending at any time, which implies that a job $J_{i,1}$ incurs pi-blocking if it is waiting.

[5] Assuming higher read request blocking would yield a smaller lower bound on write request blocking. Note that deriving tight lower bounds for RW locks is much more complicated than for the other locks considered in this paper because much leeway exists regarding the interplay between readers and writers.

Table 4 Asymptotically optimal locking protocols for RW locks under s-oblivious analysis.

Scheduling	Protocol	Release blocking	Read request blocking	Write request blocking
Clustered JLFP	CRW-OMLP [11]	$2mL_{max}$	$2L^q_{max}$	$(2m-1)L^q_{max}$
C-FIFO	RW-OLP-F (This work)	0	$2L^q_{max}-1$	$(2m-3)L^q_{max}$

Finally, we prove the claim of the theorem by showing that there is a writer job that incurs pi-blocking for the duration of $(2m-5)L-1$ time units. Job $J_{m-1,1}$ issues a read request at time 1 and acquires ℓ_q (as the scheduling policy is work-conserving). Fig. 6 illustrates this. Each job $J_{i,1}$ with $i < m-1$ issues a write request at time 2.

Each job $J_{i,1}$ with $i > m-1$ (e.g., the jobs of τ_5 and τ_6 in Fig. 6) is released 1.0 time unit before $J_{i-1,1}$ completes and issues a read request when $J_{i-1,1}$ completes. Thus, $J_{i,1}$'s read request cannot be delayed to satisfy two or more pending write requests without incurring read request blocking of at least $2L$ time units. As a result, at most one write request can be satisfied between two consecutive read requests. Thus, there is a write request from a job $J_{u,1}$ with $i < m-1$ (e.g., τ_3's job in Fig. 6) that must be satisfied after all read and write requests of each job $J_{i,1}$ with $i \neq u$ complete.

Since $J_{u,1}$ issues its request at time 2 and $J_{m-1,1}$ (e.g., τ_4's job in Fig. 6) acquires ℓ_q at time 1, $J_{m-1,1}$ pi-blocks $J_{u,1}$ for $L-1$ time units. The stated job release pattern ensures that no two of the remaining $m-3$ read requests (e.g., those by τ_5 and τ_6 in Fig. 6) overlap, so they pi-block $J_{u,1}$ for $(m-3)L$ time units. Finally, $J_{u,1}$ is pi-blocked by each of the other $m-3$ write requests (e.g., those by τ_1 and τ_2 in Fig. 6) for $(m-3)L$ time units. Thus, $J_{u,1}$ incurs pi-blocking for $L-1+(m-3)L+(m-3)L = (2m-5)L-1$ time units. ◀

For simplicity, Theorems 5 and 16 are stated for work-conserving scheduling. However, both theorems are also true under a wider class of schedulers and locking protocols that are *top-c-work-conserving*. On a c-processor cluster, a top-c-work-conserving scheduling ensures that any top-c-highest priority ready job immediately acquires a shared resource (including processor) if such a resource is idle. Note that a work-conserving scheduler and locking protocol combination is also top-c-work-conserving.

Asymptotically optimal RW locking protocols. For RW locks, the CRW-OMLP is an asymptotically optimal locking protocol under clustered JLFP scheduling [11]. The CRW-OMLP is a *phase-fair* RW locking protocol. *Phase-fair* RW locks satisfy read and write requests in alternating phases [12]. At the beginning of a *reader phase*, all waiting read requests are satisfied simultaneously, while at the beginning of a *writer phase*, a single waiting write request is satisfied. Tbl. 4 summarizes the CRW-OMLP.

The RW-OLP-F. We now introduce the *read-optimal RW locking protocol under C-FIFO scheduling (RW-OLP-F)*, which achieves optimal pi-blocking for read requests under C-FIFO scheduling. The RW-OLP-F is a phase-fair RW locking protocol that achieves $2L^q_{max}-1$ (resp., $(2m-3)L^q_{max}$) request blocking for read (resp., write) requests – here, however, we only prove a bound of $2L^q_{max}$ for reads due to space limitation. Unlike the CRW-OMLP, the RW-OLP-F has no release blocking under C-FIFO scheduling.

Structures. For each resource ℓ_q, we have two queues RQ^1_q and RQ^2_q that contain read requests for ℓ_q, and a FIFO queue WQ_q that contains write requests for ℓ_q. One of the read queues acts as a *collecting* queue and the other acts as a *draining* queue. The roles of RQ^1_q and RQ^2_q alternate, i.e., each switches over time between being the collecting queue and being the draining queue. Initially, RQ^1_q is the collecting queue and RQ^2_q is the draining queue.

Reader rules. Assume that a job J_i attempts to issue a read request $\mathcal{R}$ for resource ℓ_q. Let RQ_q^c and RQ_q^d be the collecting and draining queues, respectively, when J_i issues $\mathcal{R}$.

R1 J_i is allowed to issue $\mathcal{R}$ only if it is one of the c highest-priority eligible jobs in its cluster. J_i suspends if necessary to ensure this condition.

R2 If WQ_q is empty when J_i issues $\mathcal{R}$, then $\mathcal{R}$ is immediately satisfied and enqueued in RQ_q^d. Otherwise, J_i suspends and $\mathcal{R}$ is enqueued in RQ_q^c.

R3 If $\mathcal{R}$ is in RQ_q^c, then it is satisfied (along with all other requests in RQ_q^c) when RQ_q^c becomes the draining queue (see Rule W3). If RQ_q^c becomes the draining queue at time t and a read request is issued at time t, then that request is enqueued in RQ_q^c before making it the draining queue. $\mathcal{R}$ is removed from RQ_q^c when it is complete. If RQ_q^c becomes empty because of $\mathcal{R}$'s removal, then the head of WQ_q (if any) is satisfied.

Writer rules. When a job J_w attempts to issue a write request $\mathcal{R}$ for a resource ℓ_q, it proceeds according to the following rules.

W1 J_w is allowed to issue $\mathcal{R}$ only if it is one of the c highest-priority eligible jobs in its cluster. J_w suspends if necessary to ensure this condition.

W2 If RQ_q^1, RQ_q^2, and WQ_q are empty when $\mathcal{R}$ is issued, then $\mathcal{R}$ is immediately satisfied and enqueued in WQ_q. Otherwise, $\mathcal{R}$ is enqueued in WQ_q and J_w suspends.

W3 Let RQ_q^d and RQ_q^c be the draining and collecting queues, respectively, when $\mathcal{R}$ is the head of WQ_q. $\mathcal{R}$ is satisfied when $\mathcal{R}$ is the head of WQ_q and RQ_q^d is empty. When $\mathcal{R}$ is complete, $\mathcal{R}$ is dequeued from WQ_q and if RQ_q^c is non-empty, then RQ_q^c (resp., RQ_q^d) becomes the draining (resp., collecting) queue. Otherwise (RQ_q^c is empty), the new head of WQ_q (if any) is satisfied.

Analysis. We now determine an upper bound on request blocking. For $m \le 2$, by Lemma 4 and Rules R1 and W1, there are at most two active requests and at most one waiting request at any time, so request blocking is at most L^q_{max} time units for both reads and writes. Henceforth, we assume $m \ge 3$. The following lemma follows from Lemma 4 and Rules R1 and W1; we omit its proof as it is similar to Lemma 7.

▶ **Lemma 17.** *The total number of requests in RQ_q^1, RQ_q^2, and WQ_q is at most m.*

We now give two helper lemmas.

▶ **Lemma 18.** *If a write request $\mathcal{R}$ is the head of WQ_q at time t, then it is satisfied by time $t + L^q_{max}$.*

Proof. Let RQ_q^c and RQ_q^d be the collecting and draining queue, respectively, at time t. If $\mathcal{R}$ is not satisfied at time t, then by Rule W3, RQ_q^d is non-empty at time t. By Rule R3, jobs with requests in RQ_q^d hold ℓ_q at time t. Let t' be the time instant when all such requests are complete. By Lemma 5 and Rule R1, $t' \le t + L^q_{max}$. By Rule R2, no read requests are enqueued in RQ_q^d during $[t, t')$. Thus, RQ_q^d becomes empty at time t'. By Rule W3, $\mathcal{R}$ is satisfied at time t'. Thus, the lemma holds. ◀

▶ **Lemma 19.** *If a write request $\mathcal{R}$ is the head of WQ_q at time t, then it is complete by time $t + 2L^q_{max}$.*

Proof. By Lemma 18, $\mathcal{R}$ is satisfied by time $t + L^q_{max}$. By Lemma 5 and Rule W1, $\mathcal{R}$ completes within L^q_{max} time units after being satisfied. Thus, the lemma holds. ◀

We now determine an upper bound on the request blocking suffered by a job when it issues a read request. We consider a job J_i that issues a read request $\mathcal{R}$ for resource ℓ_q. As depicted in Fig. 3, let t_1, t_2, t_3, and t_4 be the time instants corresponding to when J_i attempts to issue $\mathcal{R}$, and when $\mathcal{R}$ is issued, satisfied, and complete, respectively. In the lemma below, for simplicity, we show that request blocking for read requests is at most $2L^q_{max}$. A tight bound of $2L^q_{max} - 1$ can be established by a detailed analysis involving multiple cases.

▶ **Lemma 20.** *For a read request $\mathcal{R}$, J_i suffers request blocking for at most $2L^q_{max}$ time units.*

Proof. J_i suffers pi-blocking for the duration of $[t_2, t_3)$. Let RQ^c_q and RQ^d_q be the collecting and draining queue, respectively, at time t_2. If WQ_q is empty at time t_2, then $t_2 = t_3$ holds according to Rule R2, so assume otherwise. By Rule R2, $\mathcal{R}$ is enqueued in RQ^c_q. Let $\mathcal{R}'$ be the request at the head of WQ_q at time t_2. Let t'_2 be the time instant when $\mathcal{R}'$ completes. By Lemma 19, $t'_2 \leq t_2 + 2L^q_{max}$ holds. By Rule W3, RQ^c_q becomes the draining queue at time t'_2. Thus, by Rule R3, all requests in RQ^c_q, including $\mathcal{R}$, are satisfied at time t'_2, implying $t_3 = t'_2$. Therefore, we have $t_3 - t_2 \leq 2L^q_{max}$. ◀

Finally, we give an upper bound on the request blocking incurred by a job when issuing a write request. Let J_w be a job that issues a write request $\mathcal{R}$ at time t.

▶ **Lemma 21.** *For a write request $\mathcal{R}$, J_w incurs request blocking for at most $(2m-3)L^q_{max}$ time units.*

Proof. If no request holds ℓ_q at time t, then by Rule W2, $\mathcal{R}$ is immediately satisfied. This leaves two cases.

Case 1. *A job with a read request holds ℓ_q at time t.* By Lemma 17, RQ^1_q, RQ^2_q, and WQ_q hold at most m requests at time t. Since there is an active read request, at most $m-2$ write requests precede $\mathcal{R}$ in WQ_q. By Rule W3, each of those write requests becomes the head of WQ_q when its preceding write request completes. By Lemma 19, a write request at the head of WQ_q completes within $2L^q_{max}$ time units from when it becomes the head. Thus, all $m-2$ write requests that precede $\mathcal{R}$ in WQ_q are complete by time $t + 2(m-2)L^q_{max}$. By Lemma 18, after becoming the head of WQ_q, $\mathcal{R}$ is satisfied within an additional L^q_{max} time units. Thus, $\mathcal{R}$ is satisfied by time $t + (2m-3)L^q_{max}$.

Case 2. *A job with a write request $\mathcal{R}'$ holds ℓ_q at time t.* We consider two subcases.

Case 2a. *WQ_q contains m requests at time t.* Thus, $m-1$ requests precede $\mathcal{R}$ in WQ_q. By Lemma 5 and Rule W1, $\mathcal{R}'$ completes within L^q_{max} time units from t. By Lemma 4 and Rules R1 and W1, no requests are issued before $\mathcal{R}'$ completes. Thus, by Rule W3, the write request $\mathcal{R}''$ following $\mathcal{R}'$ is satisfied when $\mathcal{R}'$ is complete. By Lemma 5 and Rule W1, $\mathcal{R}''$ completes within L^q_{max} time from when it is satisfied. Thus, the top two requests in WQ_q are complete by time $t + 2L^q_{max}$. By Lemma 19, each of the remaining $m-3$ write requests preceding $\mathcal{R}$ is complete within $2L^q_{max}$ time units after becoming the head of WQ_q. Thus, $\mathcal{R}$ becomes the head of WQ_q by time $t + 2L^q_{max} + 2(m-3)L^q_{max} = t + 2(m-2)L^q_{max}$. By Lemma 18, $\mathcal{R}$ is satisfied within L^q_{max} time units after becoming WQ_q's head. Thus, $\mathcal{R}$ is satisfied by time $t + (2m-3)L^q_{max}$.

Case 2b. *WQ_q contains at most $m-1$ requests at time t.* Thus, at most $m-2$ requests precede $\mathcal{R}$ in WQ_q. By Lemma 5, $\mathcal{R}'$ completes within L^q_{max} time units from t. By Lemma 19, each of the remaining $m-3$ write requests preceding $\mathcal{R}'$ completes within $2L^q_{max}$ time units

from when it becomes the head of WQ_q. Thus, $\mathcal{R}$ becomes the head of WQ_q within $L^q_{max} + 2(m-3)L^q_{max} = (2m-5)L^q_{max}$ time units from t. By Lemma 18, $\mathcal{R}$ is satisfied within L^q_{max} time units after becoming WQ_q's head. Thus, $\mathcal{R}$ is satisfied by time $(2m-4)L^q_{max}$. ◀

Similar to the OLP-F, no job suffers release blocking due to a resource-holding job under the RW-OLP-F. By Lemma 20 and 21 and letting $N_i^{q,r}$ and $N_i^{q,w}$ denote the maximum number of read and write requests for ℓ_q by τ_i, we have the following.

▶ **Theorem 22.** *Under the RW-OLP-F, J_i is pi-blocked for at most*

$$b_i = \sum_{q=1}^{n_r} (N_i^{q,r} \cdot 2L^q_{max} + N_i^{q,w} \cdot (2m-3)L^q_{max}).$$

As mentioned already, the $2L^q_{max}$ term above can be replaced by $2L^q_{max} - 1$ at the expense of more lengthy analysis. By Rules R1, R2, W1, and W2, FIFO scheduling and RW-OLP-F ensures top-c-work-conserving property. Thus, by Theorems 15 and 16, the RW-OLP-F ensures optimal request blocking for read requests, while ensuring that the request blocking for write requests is just under two request lengths of optimal.

8 Experimental Evaluation

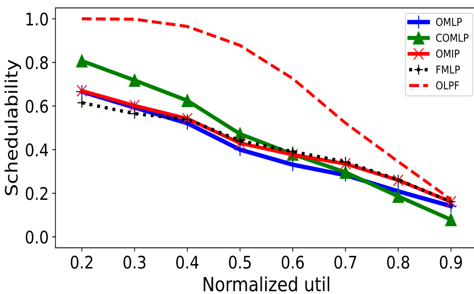
In this section, we present the results of experiments we have conducted using the SchedCAT toolkit [1] to evaluate our proposed locking protocols.

Task system generation. Our task-system generation method is similar to that used in prior locking-related schedulability studies [6, 9, 32]. We generated task systems randomly for systems with $\{4, 8, 16\}$ processors. For each processor count, we generated task systems that have a *normalized utilization*, i.e., $\sum_{i=1}^{n} u_i/m$, from 0.2 to 0.9 with a step size of 0.1. We chose the number of tasks uniformly from $[2m, 150]$. We generated each task's utilization uniformly following procedures from [19]. We chose each task's period randomly from $[3, 33]$ms (*short*), $[10, 100]$ms (*moderate*), or $[50, 500]$ms (*long*). We set each task's WCET C_i to $T_i \cdot u_i$ rounded to the next microsecond.

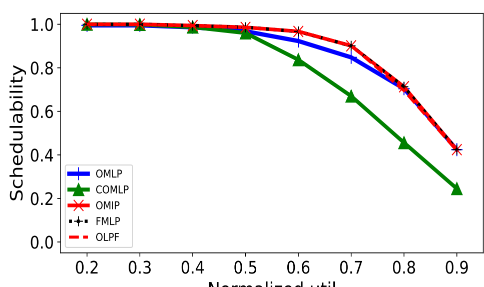
We considered $\{m/4, m/2, m, 2m\}$ number of shared resources. For each τ_i and resource ℓ_q, we selected τ_i to access resource ℓ_q with probability $p^{acc} \in \{0.1, 0.25, 0.5\}$. If so selected, τ_i was defined to access ℓ_q via $N_i^q \in \{1, 2, \ldots, 5\}$ requests. For each $N_i^q > 0$, we chose the maximum request length L_i^q randomly from three uniform distributions ranging over $[1, 15]\mu$s (*short*), $[1, 100]\mu$s (*medium*), or $[5, 1280]\mu$s (*long*). A chosen L_i^q value was decreased accordingly if it caused the sum of all request length of τ_i to exceed C_i. For each combination of m, normalized utilization, T_i, L_i^q, p^{acc}, and n_r, we generated 1,000 task systems. We call each combination of these parameters a *scenario*.

Experiment 1. In our first experiment, we considered mutex sharing. Each task had a *soft* timing constraint, meaning that it was deemed schedulable if its response time was bounded. We considered resource synchronization under the OLP-F, the OMLP [11], the C-OMLP [13], the OMIP [7], and the FMLP [5]. For the OLP-F, each task system's schedulability was tested under global FIFO scheduling [22]. For the remaining protocols, s-oblivious schedulability tests were performed under global EDF scheduling [16].[6] For each scenario, we assessed *acceptance ratios*, which give the percentage of task systems that were schedulable under each locking protocol. We present a representative selection of our results in Fig. 7.

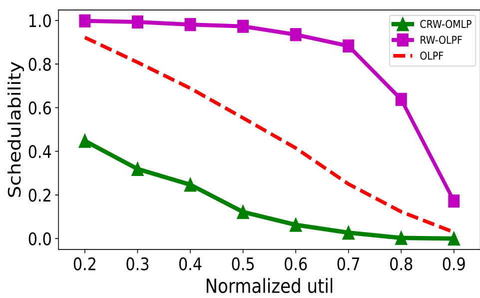
[6] The same schedulability test also applies for a wider class of global schedulers including FIFO.



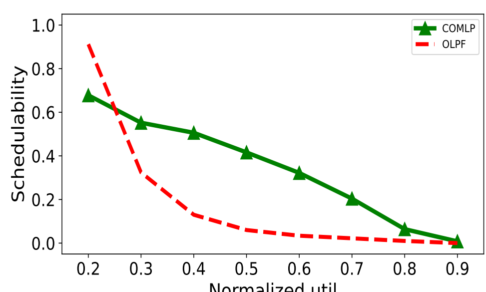
(a) Exp. 1 with $m = 8$, moderate periods, medium requests, $p^{acc} = 0.2$, $n_r = 2$.



(b) Exp. 1 with $m = 16$, short periods, short requests, $p^{acc} = 0.1$, $n_r = 8$.



(c) Exp. 2 with $m = 8$, long periods, medium requests, $p^{acc} = 0.2$, $p^{write} = 0.3$, $n_r = 32$.



(d) Exp. 3 with $m = 4$, long periods, short requests, $p^{acc} = 0.5$, $n_r = 8$.

Figure 7 Experimental results.

▶ **Observation 1.** *The average improvement under the OLP-F over the OMLP, the C-OMLP, the OMIP, and the FMLP was 20.2%, 14.9%, 16.4%, and 27.5%, respectively.*

This can be seen in insets (a) and (b) of Fig. 7. Unsurprisingly, schedulability was improved under the OLP-F because of lower pi-blocking compared to the other protocols. In some cases, as depicted in Fig. 7(b), all protocols had similar schedulability. This can happen when the number of request-issuing jobs for each resource is small (e.g., less than the number of processors), in which case all protocols have similar pi-blocking bounds.

Experiment 2. This experiment pertains to RW sharing. To generate task systems, we used one additional parameter $p^{write} \in \{0.1, 0.2, 0.3, 0.5, 0.7\}$. We defined each resource access to be a write (resp., read) access with probability p^{write} (resp., $1 - p^{write}$). In this experiment, we considered soft real-time scheduling with resource synchronization under the RW-OLP-F, the CRW-OMLP [13], and the OLP-F. Each task system's schedulability was tested under global FIFO scheduling when the OLP-F and the RW-OLP-F were employed, and under global EDF scheduling otherwise. We have the following observation.

▶ **Observation 2.** *The RW-OLP-F improved schedulability over the CRW-OMLP across all scenarios. The RW-OLP-F had less schedulability than the OLP-F when write accesses were more frequent, i.e., high p^{write} values.*

This can be seen in Fig. 7(c). The improved pi-blocking bound enabled higher schedulability under the RW-OLP-F. The RW-OLP-F had better or equal schedulability than the OLP-F across 90% of the total scenarios. Since the RW-OLP-F has higher write request blocking compared to the OLP-F (which does not have optimal read request blocking), the OLP-F had better schedulability than the RW-OLP-F when p^{write} values are high, e.g., $p^{write} = 0.7$.

Experiment 3. In this experiment, we considered hard real-time scheduling under mutex locks. For each task τ_i, we randomly chose a relative deadline between $[T_i, 2T_i]$. We considered partitioned scheduling because of the lack of hard real-time schedulability tests for global FIFO scheduling. We used the *worst-fit* bin packing heuristic to partition each task system. We compared schedulability under the OLP-F and partitioned FIFO scheduling with the partitioned OMLP (the C-OMLP with $c = 1$) and partitioned EDF scheduling.

▶ **Observation 3.** *The partitioned OMLP had better schedulability compared to the OLP-F.*

This can be seen in Fig. 7(d). Despite having lower pi-blocking and bounded response times, the partitioned OMLP enabled better schedulability because of the optimality of uniprocessor EDF in scheduling hard real-time workloads. Note that, unlike for EDF, the employed FIFO schedulability test was non-exact [4].

9 Related Work

The literature on suspension-based multiprocessor real-time locking protocols is quite vast (e.g., [7, 11, 13–15, 17, 20, 21, 23–25, 27, 29]). An excellent recent survey is given in [10]. Below, we comment further on a few specific relevant protocols.

In work on mutex locks, the FMLP [5] was the first multiprocessor locking protocol to be studied under s-oblivious analysis. While relatively simple, the FMLP has $O(n)$ pi-blocking under s-oblivious analysis. The first mutex protocols that were shown to have asymptotically optimal s-oblivious pi-blocking were the OMLP and its variants, which include protocols applicable under partitioned, global, and clustered JLFP scheduling [11, 13, 14]. In later work, the OMIP [7] was presented; it upholds an *independence preserving* property that results in asymptotically optimal s-oblivious pi-blocking under clustered JLFP scheduling.

The first multiprocessor mutex locking protocols were designed to be studied under s-aware analysis. Many of these protocols (e.g., the MPCP [27], the PPCP [17], the PIP [26], etc.) were inspired by classical uniprocessor locking protocols. The FMLP$^+$ [9] is an extension of the FMLP that has been shown to have asymptotically optimal s-aware pi-blocking under clustered JLFP scheduling. In other work, linear-programming techniques were proposed that enable improved s-aware analysis of various protocols, including the PIP, the PPCP, and the FMLP, under global and partitioned fixed-priority scheduling [8, 32].

10 Conclusion

In this paper, we have presented optimal suspension-based multiprocessor locking protocols for mutex, k-exclusion, and RW synchronization. In particular, we have shown that the s-oblivious lower bound of $m - 1$ request lengths for mutex locks is indeed tight under FIFO scheduling. We have also provided a tight s-oblivious lower bound on read-request blocking for RW locks. All three locking protocols presented herein can be used together in the same system without jeopardizing the presented analysis. Moreover, spin-based versions of these protocols can be easily obtained by following the same design principles.

For some non-FIFO JLFP schedulers, it may be possible that $2m - 1$ request lengths is indeed a tight lower bound on s-oblivious pi-blocking for mutex locks. Showing this would require a new lower-bound proof. As seen in Sec. 7, finding task systems that justify such a lower bound can be quite difficult. The results of this paper show that any task system used to justify a $2m - 1$ lower bound must necessarily not be FIFO-scheduled. In some sense, this is unfortunate, as FIFO schedules are somewhat easier to deal with in lower-bound arguments, given that having "top-c" priority is a stable property for FIFO-scheduled jobs.

References

1. SchedCAT: Schedulability test collection and toolkit. `https://github.com/brandenburg/schedcat`. Accessed: 2023-05-07.
2. S. Ahmed and J. Anderson. Tight tardiness bounds for pseudo-harmonic tasks under global-EDF-like schedulers. In *ECRTS'21*, pages 11:1–11:24, 2021.
3. B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. Davis. An empirical survey-based study into industry practice in real-time systems. In *RTSS'20*, pages 3–11, 2020.
4. K. Bedarkar, M. Vardishvili, S. Bozhko, M. Maida, and B. Brandenburg. From intuition to coq: A case study in verified response-time analysis of FIFO scheduling. In *RTSS'22*, pages 197–210, 2022.
5. A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *RTCSA'07*, pages 47–56, 2007.
6. B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-time Operating Systems*. PhD thesis, University of North Carolina at Chapel Hill, 2011.
7. B. Brandenburg. A fully preemptive multiprocessor semaphore protocol for latency-sensitive real-time applications. In *ECRTS'13*, pages 292–302, 2013.
8. B. Brandenburg. Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling. In *RTAS'13*, pages 141–152, 2013.
9. B. Brandenburg. The FMLP+: an asymptotically optimal real-time locking protocol for suspension-aware analysis. In *ECRTS'14*, pages 61–71, 2014.
10. B. Brandenburg. Multiprocessor real-time locking protocols. In *Handbook of Real-Time Computing*, pages 347–446. Springer, 2022.
11. B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *RTSS'10*, pages 49–60, 2010.
12. B. Brandenburg and J. Anderson. Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real Time Syst.*, 46(1):25–87, 2010.
13. B. Brandenburg and J. Anderson. Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In *EMSOFT'11*, pages 69–78, 2011.
14. B. Brandenburg and J. Anderson. The OMLP family of optimal multiprocessor real-time locking protocols. *Des. Autom. Embed.*, 17(2):277–342, 2014.
15. C. Chen, S. Tripathi, and A. Blackmore. A resource synchronization protocol for multiprocessor real-time systems. In *ICPP'94*, pages 159–162, 1994.
16. U. Devi and J. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. *Real-Time Syst.*, 38(2):133–189, 2008.
17. A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *RTSS'09*, pages 377–386, 2009.
18. G. Elliott and J. Anderson. An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems. *Real-Time Syst.*, 49(2):140–170, 2013.
19. P. Emberson, R. Stafford, and R. Davis. Techniques for the synthesis of multiprocessor tasksets. In *WATERS'10*, pages 6–11, 2010.
20. D. Faggioli, G. Lipari, and T. Cucinotta. Analysis and implementation of the multiprocessor bandwidth inheritance protocol. *Real Time Syst.*, 48(6):789–825, 2012.
21. K. Lakshmanan, D. Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *RTSS'09*, pages 469–478, 2009.
22. H. Leontyev and J. Anderson. Tardiness bounds for FIFO scheduling on multiprocessors. In *ECRTS'07*, page 71, 2007.
23. F. Nemati, M. Behnam, and T. Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *ECRTS'11*, pages 251–261, 2011.
24. F. Nemati and T. Nolte. Resource hold times under multiprocessor static-priority global scheduling. In *RTCSA'11*, pages 197–206, 2011.
25. F. Nemati and T. Nolte. Resource sharing among real-time components under multiprocessor clustered scheduling. *Real Time Syst.*, 49(5):580–613, 2013.

26 R. Rajkumar. *Synchronization In Real-Time Systems – A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

27 R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *RTSS'88*, pages 259–269, 1988.

28 L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time system synchronization. *IEEE Trans. Comp.*, 39(9):1175–1185, 1990.

29 Z. Tong, S. Ahmed, and J. Anderson. Overrun-resilient multiprocessor real-time locking. In *ECRTS'22*, pages 9:1–9:23, 2022.

30 B. Ward, G. Elliott, and J. Anderson. Replica-request priority donation: A real-time progress mechanism for global locking protocols. In *RTCSA'12*, pages 280–289, 2012.

31 R. Wilhelm. Real time spent on real time (invited talk). In *RTSS'20*, pages 1–2, 2020.

32 M. Yang, A. Wieder, and B. Brandenburg. Global real-time semaphore protocols: A survey, unified analysis, and comparison. In *RTSS'15*, pages 1–12, 2015.

A Tight Holistic Memory Latency Bound Through Coordinated Management of Memory Resources

Shorouk Abdelhalim ✉
McMaster University, Hamilton, Canada

Danesh Germchi ✉
University of Waterloo, Canada

Mohamed Hossam ✉
McMaster University, Hamilton, Canada

Rodolfo Pellizzoni ✉
University of Waterloo, Canada

Mohamed Hassan ✉
McMaster University, Hamilton, Canada

Abstract

To facilitate the safe adoption of multi-core platforms in real-time systems, a plethora of recent research efforts aim at bounding the delays induced by interference upon accessing the shared memory resources in these platforms. These efforts, despite their value, are scattered, with each one focusing solely on only one of these resources with the premise that latency bounds separately driven for each resource can be added all together to provide a safe end-to-end memory bound. In this work, we put this assumption to the test for the first time by 1) considering a realistic multi-core memory hierarchy system, 2) deriving the bounds for accessing the shared resources in this system, and 3) highlighting the limitations of this widely-adopted approach. In particular, we show that this approach leads to not only excessively pessimistic but also unsafe bounds. Motivated by these findings, we propose GRROF: a novel approach to predictably and efficiently schedule memory requests while traversing the entire memory hierarchy through coordination among arbiters managing all the resources in this hierarchy. By virtue of this novel mechanism, we managed to exploit pipelining upon analyzing the latency of the memory requests for tightly bounding the worst-case latency. We prove in the paper that GRROF enables us to derive a drastically tighter bound compared to the common additive latency approach with more than 18× reduction in the end-to-end memory latency bound for a modern Out-of-Order quad-core platform. The reduction is further improved significantly with the increase in the number of cores. The proposed solution is fully prototyped and tested in a cycle-accurate simulation. We also compare it with real-time competitive state-of-the-art and performance-oriented solutions existing in modern Commercial-off-the-Shelf (COTS) platforms.

2012 ACM Subject Classification Computer systems organization → Real-time system architecture; Computer systems organization → Embedded hardware

Keywords and phrases Predictability, Main Memory, Caches, Real-time

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.17

Supplementary Material *Software (Source Code)*: `https://gitlab.com/FanosLab/endtoend_wcl_cases_matlab/`

Funding This work has been supported in part by NSERC, CMC Microsystems, and TII. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.

Acknowledgements We would like to thank the anonymous reviewers for their valuable feedback, and our shepherd for helping to significantly improve this paper.

© Shorouk Abdelhalim, Danesh Germchi, Mohamed Hossam, Rodolfo Pellizzoni, and Mohamed Hassan; licensed under Creative Commons License CC-BY 4.0

1 Introduction

With the increasing performance requirements and amounts of data to be processed by modern real-time systems, adopting multi-core platforms becomes favorable, if not necessary. One of the main roadblocks to this adoption is the architectural complexity of these platforms, which threatens the timing analyzability of the system and the ability to derive safe yet tight Worst-Case Execution Time (WCET) for real-time tasks. In particular, the several memory resources among the cores pose a significant challenge in bounding the interference-induced delays suffered upon accessing these resources. Therefore, the real-time community has recently invested significant (but somewhat scattered) efforts to address this challenge at each of the different memory resources, including interconnects [14,25,42], caches [8,11,27,28,37,41], and main memories [1, 6, 9, 20]. Each of these efforts focused solely on one of these resources with the premise that latency bounds separately driven for each resource can be added altogether to provide a safe end-to-end memory bound, which we refer to as the *additive latency* approach [10].

In this work, we assess this assumption by conducting the following contributions.

1. We consider a comprehensive and realistic multi-core memory hierarchy system modeled after Commercial-off-the-Shelf (COTS) platforms, where they are independent resources that can be accessed in parallel. This includes a split-transaction interconnect between private L1 caches and the shared Last Level Cache (LLC) composed of a request and a response bus, a realistic cache model with write buffers and non-blocking support to enable several requests to be serviced in parallel, a bankized LLC with several independent banks that can be accessed in parallel, and a system bus to carry requests from the LLC misses and write backs to the memory controller to be sent to the off-chip memory. Section 3 elaborates on this system model.
2. We derive the bounds for the resources in this system following the aforementioned additive latency approach by considering each resource independently. We then use this step to highlight two limitations of this approach. In particular, a) on the one hand, it leads to excessively pessimistic bounds to the level that they reach several thousands of cycles for one request and hence becomes practically useless. This is due to the aggressive reordering and parallelism deployed in these COTS platforms. b) On the other hand, it leads to unsafe bounds due to the fact that each separate resource has to make local assumptions about the ordering of requests that does not necessarily align with the actual request orders in the system. These two limitations are further discussed in Section 2.2 and are illustrated with a numerical example in Section 4.
3. To address this problem, we propose the Global Round Robin Oldest-First (GRROF) as a novel methodology to predictably and efficiently schedule memory requests upon traveling through the entire memory hierarchy. Instead of managing resources independently and analyzing them in isolation from each other, GRROF enables arbiters to operate fully in parallel yet coordinate by sharing and updating a centralized engine that tracks states about requests currently in the system. More details about this approach are provided in Section 4.
4. We use this coordination to conduct a novel analysis that derives a bound, for the first time to the best of our knowledge, on the end-to-end latency suffered by a request upon accessing the memory hierarchy including accessing the interconnect, the LLC, the system bus to the off-chip DRAM, if it is a miss in the LLC, and all the way until it returns to its requesting core and retires. This novel analysis leverages GRROF to pipeline requests among these resource and apply the delay composition theorem originally proposed in [21].

Table 1 DDR4-2400U Timing Constraints [38]. l and s refer to the large (same bank group) and small (different bank groups) timing constraints, respectively.

	Inter-Bank Constraints			Intra-Bank Constraints	
	Description	Cycles		Description	Cycles
t_{RRD}	ACT to ACT	l=6, s=4	t_{RL}	RD to DATA	18
t_{FAW}	4 ACT Window	26	t_{WL}	WR to DATA	12
t_{WTR}	WR DATA to RD	l=9, s=3	t_{WR}	WR DATA to PRE	18
t_{WtoR}	WR to RD	25	t_{RP}	PRE to ACT	18
t_{RTW}	RD to WR	12	t_{RCD}	ACT to CAS	18
t_{BUS}	DATA	4	t_{RTP}	RD to PRE	9
t_{CCD}	CAS to CAS	l=6, s=4	t_{RC}	ACT to ACT	57
			t_{RAS}	ACT to PRE	39

This leads to a drastically tighter bound compared to the common additive latency approach. This reaches more than 18× reduction in a modern Out-of-Order quad-core platform. The analysis is derived in Section 5.

5. We prototype this whole memory system in a cycle-accurate simulator in addition to three other approaches. The first two are modeled after predictable hard-ware real-time solutions, such as Round-Robin (RR) [7,36] and Round-Robin Oldest-First (RROF) [30,33], while the third represents a First-Ready First-Come First-Serve (FRFCFS) approach that reorders requests to increase system performance and is commonly used in COTS platforms. Section 6 discusses the detailed results of these comparisons. And finally, Section 7 is the conclusion.

2 Background and Related Work

2.1 DRAM Memory Background

DRAM device is the off-chip main memory that communicates with on-chip processing elements through a Memory Controller (MC). The device consists of multiple banks of 2D array structure that are indexed by row and column addresses and accessed through data, address, and command buses. Accessing data from a DRAM bank is generally a two-stage process. 1) The row address is provided to activate the requested row through an activation (ACT) command. 2) The column address is provided to conduct the requested read/write operation through a CAS (RD/WR) command. Each DRAM bank also has a row buffer that holds the most recently accessed row from that bank. This enables future accesses to the same row by read/write from the buffer directly without re-activating the row (row hit), and that only requires a CAS command. On the other hand, if a request accesses a row different than the one in the buffer (row miss), the MC has first to pre-charge the row through a PRE command, and then issues the ACT and CAS commands. Those commands (PRE, ACT, and R/W CAS) should be separated by the timing constraints defined in the DRAM JEDEC standard [38] to ensure a correct behavior from the DRAM. Table 1 shows the relevant timing constraints. Some of these constraints apply to the commands of the same bank (intra-bank), while others apply to the commands among different banks (inter-bank). A command is considered intra-ready or inter-ready when it satisfies its intra-bank or inter-bank constraints, respectively, and it becomes ready when both constraints are met.

2.2 Motivation: State-of-The-Art Limitations

The current paradigm to calculate the total WCET of a task in a multi-core platform while accounting for the interference along the memory hierarchy is to use the additive latency approach [10]. In this approach, every resource that is subject to contention is analyzed separately (i.e., independent of other resources) to derive an upper bound on the latency suffered upon accessing that resource. Afterwards, all latency bounds of all resources can be added together to provide an overall safe bound. Most of the existing work in bounding memory-related interference follows that approach; for instance, by focusing on caches [12, 13, 18, 23, 33], DRAM [15, 16, 30, 31], or memory interconnect [14, 17]. Thus, we make two critical observations about this approach.

1) On the one hand, **this analysis conducted separately at each resource has to assume the maximum possible interference at this resource**. This has to be applied to all considered resources leading to very pessimistic bounds when all added together. For example, for a multi-core system with M Out-of-Order (OoO) cores, each of which can have N_{pend} possible outstanding requests, the analysis has to assume the maximum possible interfering requests from all other cores on the resource under analysis, which is $(M-1) \cdot N_{pend}$. For example, existing work in analyzing DRAMs has considered this number of possible competing requests [15, 43]. We make the observation that this is due to the fact that *COTS platforms, to optimize performance, deploy aggressive parallelism among these resource and reorderings among requests targeting them. They do not maintain a global ordering view that is shared by all these resources.* Using this observation, we show that by providing such global ordering, GRROFenables us to derive a considerably tighter bound by making a holistic analysis of all the resources amenable. 2) On the other hand, **the conducted analysis considering only one resource can lead to unsafe assumptions**. In particular, in the case of analyzing requests from an OoO core with multiple outstanding requests, the analysis has to consider the request that arrives first to the resource under consideration to be the oldest from that core. This is, for example, what is conducted in the existing analysis for DRAMs [32] and caches [33]. Although this is true from this resource perspective, it is not necessarily valid from the real (core issuance) perspective. For instance, in a real multi-core platform, where there exists parallelism in the memory hierarchy, a younger request can arrive at a resource before an older one from the same core. This simply destroys the notion of older request from a core perspective, which can entail significant delays to that request upon being arbitrated at one of the resources. The only way to derive a safe bound on such a case is to always assume that the request under analysis arrives at this resource last after the maximum possible number of earlier requests from the same core. This further pushes the pessimism of the analysis leading to extremely significant latency bounds. We will discuss these limitations more with an illustrative example in Section 4 and analytically bound the delays using the additive latency approach in Section 5.3.

2.3 Delay Composition Theorem for Real-Time Pipelines

Our analytical bounds use the delay composition strategy first introduced in [21] and apply it to the whole memory hierarchy. While this analysis method has been introduced to compute the worst-case latency of distributed real-time jobs, it has also been previously applied to obtain upper latency bounds for DRAM requests [15, 16, 43]. In detail, the analysis considers a set of jobs, which we will equate to hardware requests, executing on a given sequence of resources (or pipeline stages). Each request has a known worst-case execution time on each stage; once a request finishes executing on a stage, except the last, it immediately

becomes ready for the next stage. The total latency of a request is defined by the difference between the time it finishes execution on the last stage in the pipeline, and the time at which it arrives on the first stage. At each stage, requests are scheduled according to either a fixed-priority preemptive or a fixed-priority non-preemptive policy; in this paper, we employ the latter since it matches the behavior of hardware resources. The main result in [21] is that pipelining allows us to constrain the interference caused by higher priority requests on a given request *u*nder *a*nalysis r_{ua}: such interference is limited to the longest execution time of a higher priority request r on any stage, rather than the sum of r's execution times on all stages. Intuitively, the idea is that once r "gets ahead" of r_{ua}, it will become ready and start executing on successive stages ahead of it; therefore, it cannot cause maximum interference on r_{ua} on each stage.

However, for this property to hold, the theorem requires the relative priority of requests to remain the same on all stages: if r temporarily drops its priority below that of r_{ua} on a stage, it might be delayed by other lower-priority requests on that stage; and once it regains its higher priority on a later stage, it might interfere again with r_{ua}. One of the main contributions of this paper is proposing a novel architecture that enables the coordinated management of all memory resources (i.e., stages) such that this property is satisfied. As a result, this enables us to leverage the pipelining feature from the theorem to derive a significantly tighter holistic memory latency bound.

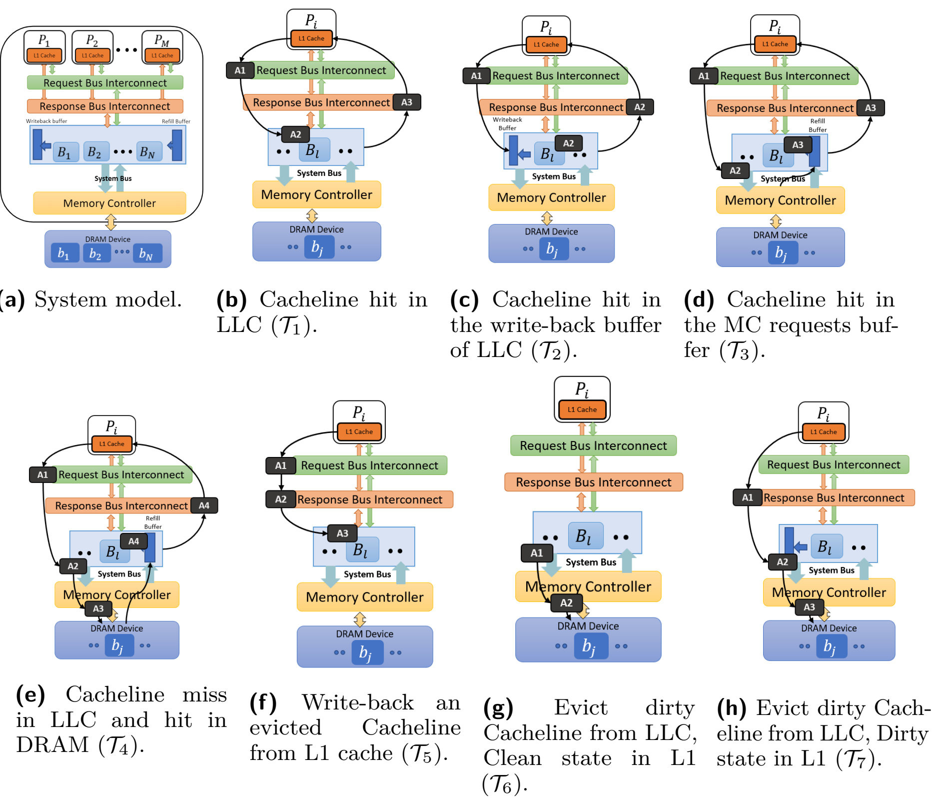
Finally, the original delay composition analysis in [21] assumes that all requests traverse the same pipeline stages. This is not generally the case for a modern memory hierarchy where requests of different types can access resources in a different order. Consider, for example, a demand miss request from a core compared to a write-back request of an evicted line from either L1 or LLC. We will thus use the improved analysis in [22], which supports such an extension. The key idea in the analysis is to split every higher priority interfering request into a set of *segments*: each segment represents the execution of that request on a sequence of consecutive stages in the path of the request under analysis, encountered either in the same or exactly in reverse order.

3 System Model

3.1 Architecture

This section introduces the hardware architecture considered in this paper, as shown in Figure 1a.

- **Processing Cores.** We consider a multi-core system with M cores $P_1, .., P_i, .., P_M$, which can be In-order (IO) or OoO cores. OoO cores can have multiple outstanding memory requests. We denote the maximum number of such in-flight requests as N_{pend}, which is usually determined by the number of available entries in the Miss Status Holding Registers (MSHRs) in the platform's caches [41, 43].
- **Caches.** We assume each core has exclusive access to a private cache (L1), and all the cores share an on-chip Last-Level Cache (LLC). In line with related work, we assume a partitioned LLC to eliminate data interference between cores at the LLC level [8]. Our proposal does not require a particular technique for partitioning; however, for the analysis, we assume a set-partitioned LLC. L1s and LLC are write-back write-allocate caches and implement Least Recently Used (LRU) replacement policy. Unlike state-of-the-art works in cache analysis in the real-time domain, a critical aspect of this paper is that we consider a more realistic cache model that employs several of the optimizations commonly deployed in COTS cache systems to improve system performance. 1) ***MSHRs.*** In order to leverage



(a) System model.

(b) Cacheline hit in LLC ($\mathcal{T}_1$).

(c) Cacheline hit in the write-back buffer of LLC ($\mathcal{T}_2$).

(d) Cacheline hit in the MC requests buffer ($\mathcal{T}_3$).

(e) Cacheline miss in LLC and hit in DRAM ($\mathcal{T}_4$).

(f) Write-back an evicted Cacheline from L1 cache ($\mathcal{T}_5$).

(g) Evict dirty Cacheline from LLC, Clean state in L1 ($\mathcal{T}_6$).

(h) Evict dirty Cacheline from LLC, Dirty state in L1 ($\mathcal{T}_7$).

Figure 1 Considered system model (a) and different request types for both demand (b – e) and write-back (f – h) requests.

the performance gain from the OoO cores, we also consider a non-blocking cache that can service multiple requests at a time. Non-blocking cache is an old concept [26] that is widely adopted in modern COTS platforms. The considered caches allow both a hit-over-miss and a miss-over-miss (i.e., servicing hits and misses while there is a pending miss), subject to the number of MSHR entries. 2) ***Allocate-on-Fill.*** We consider an allocate-on-fill cache, where cache lines that miss in the cache are allocated in the data array only upon receiving the data from the lower memory. This avoids unnecessary early eviction of cache lines and hence can enable more hits [2]. 3) ***Write Buffer.*** Caches employ a write buffer where it places dirty lines upon eviction to be written to the lower memory level. This enables a faster allocation for the new cache line without waiting for the evicted line to be written to that lower memory. Similar to MSHR, write buffering is a standard technique in modern COTS platforms [39] including Intel's [19] and ARM's [3]. It does not only exist in high-end platforms, but even the low-end ones usually used in real-time embedded systems such as the ARM M4 includes some form of a write buffer [4]. 4) **Write Buffer Hits.** Furthermore, the cache controller allows hitting in its write-back buffer if it receives a request to a dirty cache line that exists in the write buffer; hence, preventing unnecessary high miss latency. Upon hitting in the write buffer, the cache controller takes two simultaneous actions to serve such requests: it sends the requested data to the requestor and saves the data again in the cache's data array. 5) ***Bankized LLC.*** We consider a multi-bank LLC where data is distributed over N cache banks; $B_1, .., B_N$. LLC banks are independent and can serve different requests in parallel; hence,

they are modeled as separate resources, and the process time of the data in/out the data arrays of a bank is c_{BANK}. Since each LLC bank is independent, it will have its own set of write buffer and MSHRs. We assume that both write buffers and MSHRs in shared LLC are not source of interference among cores. This is because their sizes in COTS platforms are usually set large enough to accommodate the maximum number of possible outstanding requests from all cores in the system. Even in the case that this assumption does not hold for some particular platform, existing works can be used to eliminate the effect on request latency such as the work in [41] for MSHRs and in [5] for write buffers. 6) ***Miss Forwarding.*** The caches deploy a common optimization to reduce the miss penalty, where requests can be determined to be whether a hit or a miss by checking the tag/status bits. This tag checking can be done in parallel to and independent of accessing the cache data array. Consequently, upon a miss, the cache controller forwards the miss to be filled from the lower level memory without accessing the data array, which reduces miss penalty. Additionally, once the data refill arrives from the lower memory, it can be immediately forwarded to the requesting core on the response bus (subject to arbitration as detailed later on), while simultaneously placed to be written also to the cache data array. 7) ***Immediate Back Invalidation.*** We also assume that back invalidation from a lower level of memory uses a dedicated special bus, and hence, do not interfere with demand requests on the request bus.

- **Interconnect Bus.** The system model considers a split-transaction shared bus between the L1 caches and the LLC. This bus comprises two independent buses: a ***request bus*** for sending requests from the L1s to the LLC and a ***response bus*** for data transmissions. This architecture allows a concurrent operation for the requests and data responses on the buses, where the transmission latency of packets on the request and the response buses are c_{REQ} and c_{RESP}, respectively.
- **System Bus.** The system bus is the interconnect between the LLC and the main memory, and used for transmitting requests and data. A packet on the bus can contain a request, data, or both, and its transmission latency is c_{SBUS}. In our model, we assume a full-duplex system bus, as shown in Figure 2, which consists of two buses: one is for packets that are sent from LLC to the main memory (LLC-DRAM bus), and the second is for the way back from the main memory (DRAM-LLC bus).
- **Memory Controller and DRAM.** Accesses to the off-chip DRAM memory are managed through an on-chip MC, as explained in Section 2.1. The MC stores incoming requests from the system bus in per-requestor buffers. We consider a DRAM with n private banks: $b_1, .., b_n$, where the MC maps every request to a bank that is assigned to its core. Afterwards, the MC translates each request into its corresponding set of commands and buffers them into the per-bank command queues. The MC arbitrates between the ready commands based on the arbitration scheme order. Two more COTS features we consider in our system model. 1) **Write Data Queue Hit.** We assume that the MC allows demanding requests to hit in its request data buffers in order to reduce the memory latency. This means that if a demanding request reaches the request buffers of the MC while the required data is in one of the outstanding write requests, the data is read from the buffers directly. 2) **Clock Domain Crossing (CDC) Effect.** Since off-chip DRAM can generally operate at a different frequency than the on-chip core one, in our end-to-end latency calculations, we have to consider the clock domain crossings that requests suffer upon traversing the memory hierarchy. This can be done simply by doing clock transformations (or calculating latencies at all stages in terms of absolute *nano* seconds). For convention, we refer to the DRAM and core clocks as t_{CLK}^{DRAM} and t_{CLK}^{CPU}, respectively.

- **Arbitration.** Arbiters are required in the system to regulate access to shared resources. The considered resources for arbitration are the request bus, the response bus, each bank of the LLC, the LLC-DRAM bus, and the three stages of the main memory (PRE, ACT, CAS). The DRAM-LLC bus does not require an arbiter as it does not incur any contention. This is because the DRAM-LLC bus is an on-chip bus between the DRAM memory controller and the LLC; therefore, its data transfer time is much lower than that of the off-chip DRAM access time. The arbitration scheme we propose to coordinate all these resources is discussed in detail in Section 4.1.

3.2 Latency Model

For any request r, let t_r^a be the time at which r arrives in the system and t_r^f the time at which r finishes executing. Formally, request r is outstanding in interval $[t_r^a, t_r^f]$. As discussed in Section 3.1, an OoO core can have multiple outstanding requests. Hence, it is essential to clarify how to compute the latency of a request. Using the same approach as in [31], we say that r is *oldest* at time t if it is the earliest arrived request of its core that is still outstanding at t. Note that because our architecture model allows multiple outstanding requests to execute in parallel and complete out-of-order, a request r might never become the oldest. However, if it does, it remains oldest until its finish time t_r^f. Furthermore, it must become oldest either at t_r^a, if there is no other outstanding request of the same core, or at the latest finish time of a request of the same core that arrived before r. The processing latency of a request is then the time during which it is oldest or zero if it never becomes oldest. Intuitively, this ensures that we do not count in the latency of a request the queuing delay caused by other requests of the same core that arrived before it. Note that when a core generates multiple concurrent requests, the time required to complete executing all requests is bounded by the sum of their processing latencies.

3.3 Request Model

In line with the delay composition theorem summarized in Section 2.3, we model each request r as executing on a sequence of stages corresponding to hardware resources in our system where requests are scheduled based on an arbiter. For LLC and interconnections, such resources comprise each of the N LLC banks, which we denote with BANK, the request bus REQ, the response bus RESP, and the LLC-DRAM bus SBUS. Note that the DRAM-LLC bus is not modeled as a pipeline stage since it is not subject to arbitration, as explained in Section 3.1.

Similar to [15, 16, 43], we model DRAM as consisting of three stages: PRE:ACT:CAS. Each stage models the interference of other requests on commands of the corresponding type. Note that because we assume private banks in DRAM, such interference can only be caused by intra-stage DRAM constraints. We define the execution time on any s stage of these stages as c_s. For instance, the execution time on the REQ stage is c_{REQ}, and on the SBUS stage is c_{SBUS}.

3.4 Request Types

Following the described request model, we classify requests into a set of request *types*; the type $\mathcal{T}(r)$ of request r determines the list of stages/resources traversed by r. We notice that requests are issued to the memory system for two main reasons: a load/store request for data that is miss in the L1 cache or a write-back request for a victim dirty cache line to the lower

memory level. Thus, the request types are split into two groups. The first one contains the demanding requests that are miss in the L1 cache, in which a core broadcasts a load/store request on the request bus and waits for the demanding data to be sent on the response bus ($\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$ and $\mathcal{T}_4$). This is denoted in Figures 1b – 1e. On the other hand, Figures 1f – 1h show the write-back requests of a dirty evicted cache line from L1 and LLC, which compose the second group ($\mathcal{T}_5$, $\mathcal{T}_6$ and $\mathcal{T}_7$). We now explain each of the request types in detail.

$\mathcal{T}_1$) REQ:BANK:RESP. This type explains the path for a request demanding a cache line that exists in the LLC. After the core broadcasts a request on the request bus, the data is read from the LLC bank and then sent to the core on the response bus (Figure 1b).

$\mathcal{T}_2$) REQ:RESP/BANK. In this case, the request hits into data that is found in the LLC write buffer. Based on the miss forwarding optimization discussed in Section 3.1, the requested cache line will be sent to the core on the response bus while simultaneously being rewritten to the LLC bank (Figure 1c).

$\mathcal{T}_3$) REQ:SBUS:RESP/BANK. This represents the case of a request that misses in the LLC bank but hits in the MC. This happens when this request targets a cache line that has been recently evicted from the LLC and sent to the DRAM; and hence, not yet written into the DRAM banks. Therefore, the system will fetch the data directly from the memory controller, and send it back to be simultaneously processed on the corresponding LLC bank, while also being sent to the requesting core through the response bus similar to the previous type. This is depicted in Figure 1d.

$\mathcal{T}_4$) REQ:SBUS:PRE:ACT:CAS:RESP/BANK. This represents the case where the request needs to fetch the data from the DRAM device. Accordingly, once the request is issued in the request bus, and misses in the LLC, the request will be sent to the DRAM on the system bus. Afterwards, the data will be fetched from the DRAM bank through the PRE:ACT:CAS stages. Then, the fetched data will be sent to the LLC bank to be written to its data array and simultaneously to the requesting core through the response bus as illustrated in Figure 1e.

$\mathcal{T}_5$) REQ:RESP:BANK. This type corresponds to a write-back request from L1 to the LLC due to the eviction of a dirty line. It first sends a request on the request bus to notify the LLC that it is going to update a cache line, and then it puts the data on the response bus to the LLC. Finally, the LLC bank gets the data and processes it as delineated in Figure 1f.

$\mathcal{T}_6$) SBUS:PRE:ACT:CAS. This type represents the write-back from LLC to DRAM due to the eviction of a dirty up-to-date cache line from the LLC bank. The LLC sends concurrently an invalidation message to the L1 caches and a write-back request to the DRAM through the SBUS. Please note that as aforementioned, the back invalidation to the L1s happens in its dedicated bus, and hence, is not subject to arbitration. Therefore, it does not have a dedicated stage. In contrast, the write to the DRAM after traversing the SBUS, requires the three DRAM stages: PRE:ACT:CAS. This type is shown in Figure 1g.

$\mathcal{T}_7$) RESP:SBUS:PRE:ACT:CAS. Similar to $\mathcal{T}_6$, this type represents an LLC write-back of a dirty evicted line to the DRAM. However, unlike $\mathcal{T}_6$, the evicted line in this case is stale, which means that it is updated in the L1 cache. Thus, when the L1 cache receives the

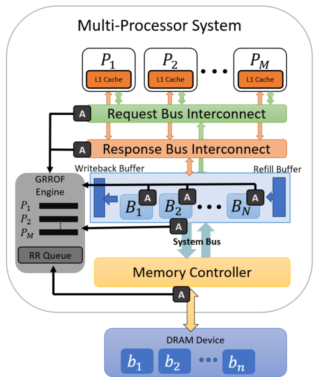


Figure 2 The architecture of GRROFproviding the global view for all arbiters.

invalidation message, it sends the updated version of the line to the LLC on the response bus. The cache controller stores the line in its write buffer until the data is received from L1 and then sends a write request to the main memory. Therefore, it requires RESP stage from L1 to the LLC, SBUS stage from LLC to the DRAM, and then the three DRAM stages: PRE:ACT:CAS. This type is shown in Figure 1h.

Two important general notes to make about the request types. First, requests of type $\mathcal{T}_2$, $\mathcal{T}_3$ and $\mathcal{T}_4$ execute in parallel on two stages ($BANK$ and $RESP$) instead of a linear sequence of stages which is the supported model by the existing pipelining analysis in [21]. To be able to pipeline, we apply the delay analysis to the two possible sequences and take the one that leads to the worst-case latency. More details on how to apply the delay analysis to such requests are in Section 5. Second, for the request types accessing the DRAM (namely, $\mathcal{T}_4$, $\mathcal{T}_6$ and $\mathcal{T}_7$), we assume a request targeting a closed row in the DRAM bank. This assumption is mandatory to provide safe worst-case latency bounds. This is because as explained in Section 2, requests targeting a closed DRAM row suffer larger delays compared to requests targeting an open row.

4 GRROF: Coordinating Management of All Memory Resource

In this section, we introduce the proposed architecture to coordinate all arbiters in the memory hierarchy, enabling us to apply the pipelining idea from the delay composition theorem. The high-level diagram of the proposed architecture is shown in Figure 2, and we use Figure 3 as a running example to explain its operation.

Since one of the motivations of this work is the inherent pessimism in considering each memory resource separately and then applying the additive latency approach, we start with an illustrative example that highlights this pessimism. Figure 3 considers a system with three cores $P_1 - P_3$ where each P_i issues three requests $r_{i,1}$-$r_{i,3}$ to the cache hierarchy. Request's arrival to the system is modeled by the up arrows $\uparrow$. For example, $r_{1,1}$, $r_{1,2}$, and $r_{1,3}$ from P_1 arrive at the timestamps 1, 9, and 17, respectively. The system has multiple arbitration stages REQ, BANK_{0-6}, and RESP. In Figure 3a, each stage employs an independent RR arbiter. Requests $r_{1,1}$, $r_{2,1}$, and $r_{3,1}$ target $BANK_0$, while the other requests are distributed over the other banks. According to the given scenario and the separate RR arbitration, request $r_{1,1}$ incurs a significant delay on RESP stage despite being the oldest request from P_1 and does not finish up until timestamp 58 (modeled by the down arrow $\downarrow$). More importantly,

if we use the additive latency approach naively without considering the fact that $r_{1,1}$ while being the oldest for P_1 might not be the local oldest at each separate resource, the bound will be the maximum possible interference due to requests from other cores in addition to the service time of $r_{1,1}$ itself in each resource. Since this is RR, it will be $2 \times 3 + 10 \times 3 + 5 \times 3=$ 51 cycles. This is less than the actual suffered bound; and hence, is in fact an unsafe bound. The only possible way to derive a safe bound is to assume that the request under analysis is always arriving last to each resource after the maximum number of requests from the same core. As explained in Section 2.2, this provides a safe bound at the expense of being extremely pessimistic.

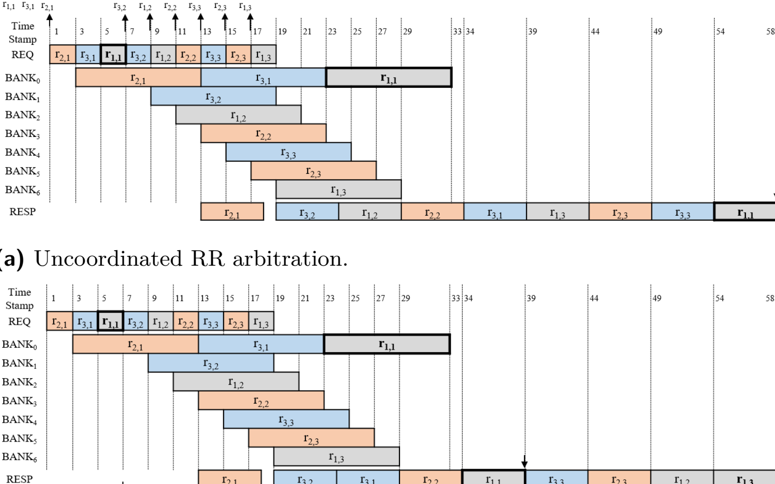
4.1 GRROF: Coordinated Management of All Memory Resources

Motivated by these observations about the existing direction in analyzing memory resources in multi-core real-time systems, we propose GRROF: a methodology to coordinate all the arbitration decisions across all resources in the memory hierarchy of a modern multi-core platform. This includes shared interconnects (request, response, and system buses), shared cache(s), and shared off-chip DRAM. The key idea behind this methodology is to enable all the arbiters in the memory hierarchy to use a shared state of the system to make a coordinated scheduling decision. It is important to emphasize that under GRROF, every resource in the memory hierarchy still deploys its own dedicated arbiter, which is essential for parallelism. This is in contrast to assuming a unified global arbiter that manages all the memory resources. We observe that, for instance, most of the existing works in cache analysis combine all interconnect resources to the shared cache as well as the shared cache itself into one resource that is arbitrated using one arbiter (e.g. [14, 24, 42]). Instead, in GRROF and as explained in Section 3, every resource has its own arbiter. So, there is a dedicated arbiter for the request bus, response bus, each LLC bank, system bus to the DRAM, and the DRAM memory controller. However, all these arbiters operate in coordination using a global view of the state of different requests in the system.

We now detail the operation of GRROF. The global view is maintained using the GRROF Engine in Figure 2. This engine maintains the following state.

1) RR Order. It maintains a RR order among the cores. Arbiters arbitrate among cores according to this RR order. Therefore, all arbiters see the same RR order view. A core gets pushed into the RR order queue if it has issued a request to the system (e.g., upon an L1 miss in our system model). Once a core is at the head of the RR order, it keeps that order until its oldest request retires from the system. For example, in Figure 3b, P_3 maintains its position at the top of the RR order despite having $r_{3,2}$ retired at timestamp 24. This is because the oldest request from P_3 is $r_{3,1}$, which still needs to finish. This is vital to provide tight guarantees for the oldest requests. *The intuition is that the oldest memory request is the one stalling the pipeline [34, 35]; and hence, contributing to the task's WCET [33].* In the same example in the figure, by keeping its RR position, P_3 manages to finish its oldest request $r_{3,1}$ at timestamp 29 compared to what happens for uncoordinated arbitration where P_3 loses its RR order in Figure 3a leading $r_{3,1}$ to wait for another slot for P_3 in the response bus resource and finishes at 39.

2) Per-Core FIFO Order. For arbiters to be able to determine the relative order of requests from the same core, the GRROF Engine maintains one First-In First-Out queue (FIFO) per core. Upon arrival to the system, a request ID is pushed in the corresponding core's FIFO. The request ID is removed from such FIFO upon retiring from the system. Accordingly,



(a) Uncoordinated RR arbitration.

(b) Proposed GRROF solution. The bottom of the figure shows the state tracked by the GRROF Engine, including the per-core FIFO order and the RR order among cores.

Figure 3 An example that shows the behavior of GRROF compared to separated RR. The assumed system contains three cores P_{1-3} that share resources REQ, BANK$_{0-6}$, and RESP. Access latencies for the REQ, BANK, and RESP resources are assumed to be 2, 10, and 5 cycles, respectively.

the FIFOs keep state about this relative order for requests from the same core. Again, all arbiters have access to these FIFOs and hence, can decide accordingly which request to elect for service at the arbitrated resource.

The arbiters deploy Round Robin Oldest First (RROF) arbitration [33]. They first conduct a RR among the oldest requests of the cores. Only if no older requests are ready to be serviced at that resource the arbiter conducts RR among younger requests (in the order of the FIFO for the same core and RR among cores). In the example in Figure 3b, the response bus (RESP) arbiter elects $r_{1,1}$ at timestamp 34 because it is the oldest request from the core at the top of the RR order. On the other hand, at 19, the RESP arbiter cannot issue any of the oldest requests ($r_{1,1}$, $r_{2,2}$, $r_{3,1}$) since none of them is ready for this resource. Accordingly, it picks $r_{3,2}$ as the only ready non-oldest request.

The important and novel aspect here is that this RROF at each resource uses the global system state from the GRROF Engine (Namely, RR Order and per-core request FIFO order). As a result, the relative request priorities remain the same for all arbitrated resources (stages in delay composition theorem terminology); hence, we can apply the pipelining from the theorem. We prove in our analysis in Section 5 how this enables us to significantly reduce the worst-case latencies suffered by the oldest requests in the system. Considering $r_{1,1}$ in Figure 3, we observe that $r_{1,1}$ arrives at the RESP bus resource last among all the requests since it has been delayed by $r_{2,1}$ and $r_{3,1}$ in the Bank$_0$ stage. As a result, in the uncoordinated RR baseline in Figure 3a, according to the local RR arbiter at the RESP stage, this request has to wait for all the requests to finish, including the non-oldest requests from the same core. We see in Figure 3a that $r_{1,1}$ suffers interference from requests from the two other cores in all the stages (REQ, BANK, and RESP) (Observation 1 in Section 2.2). Moreover, $r_{1,3}$ is serviced by the RESP before $r_{1,1}$ since locally, $r_{1,3}$ is considered older (Observation 2 in

Section 2.2). This overall leads $r_{1,1}$ to finish at timestamp 58. On the other hand, GRROF, in Figure 3b, is aware of the global order of all requests, thus once $r_{1,1}$ arrives at the RESP stage, it gets the highest priority among all P_1 requests since it is the oldest. Additionally, P_1 is at the top of the RR queue since its oldest request still needs to finish. As a result, $r_{1,1}$ is serviced at timestamp 39 instead of 58.

4.2 The Proposed Memory Controller

To be able to apply the pipelining from the delay composition theorem across the entire memory hierarchy, the schedulers inside the DRAM MC have to apply the exact arbitration mechanism described in Section 4.1. Therefore, we also propose an MC scheduler that utilizes the global request state from the GRROF Engine. In detail, we employ three distinct RROF arbiters for PRE, ACT, and CAS commands, which follow the global order maintained by the centralized GRROF Engine. If multiple commands from different arbiters are selected at the same time, a command bus conflict may happen. The MC handles this conflict by prioritizing CAS commands over ACT, and ACT over PRE commands, which is the common approach followed by COTS MCs. The intuition is that CAS commands are for row hit requests and hence prioritizing them will increase the overall system performance. However, the proposed MC strictly applies the RROF scheme between CAS commands and handles reads and writes equivalently. This is in contrast to DuoMC [32], which deploys read/write batching, where several read (write) commands are scheduled together as a batch and executed in a read (write) round, and rounds are alternating types. The reason for avoiding read/write batching is we find this to break the property required by the delay composition theorem. Basically, by read/write batching, requests no longer keep their relative priority across the different stages in the memory hierarchy. For example, a write might have a higher priority than a read in one of the cache levels (according to GRROF orders) but the write gets deprioritized at the MC (e.g. if it is executing a read batch upon its arrival to its request queues). Breaking this property hinders our goal of being able to apply pipelining to the memory latency analysis. Instead, by handling reads and writes in a similar fashion, we are able to apply the pipelining among DRAM command stages. This pipelining at the DRAM has been explored by previous works [15, 16, 43], albeit they considered DRAM only.

It is important to notice that deploying the same GRROF arbitration methodology at all memory resources including within cache hierarchy as well as in DRAM enables us to apply the delay composition theorem since all requests maintain their relative order throughout the resources. Without this coordination, 1) suffered delays at different stages cannot be overlapped by applying to the pipeline, and 2) the oldest request at one resource can become non-oldest at another resource and hence can be significantly delayed. This is similar to the clarified example in Figures 3a and 3b.

5 Latency Analysis

In this section, we show how to obtain a latency bound L_{ua} on the worst-case processing latency of the oldest request under analysis r_{ua} of a given type, following our described GRROF arbitration in Section 4.1 and MC in Section 4.2, and based on the delay composition framework.

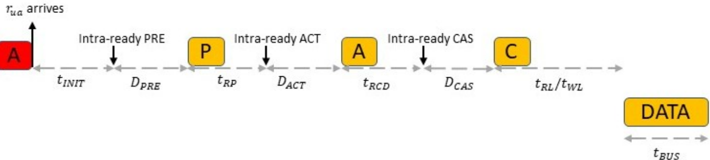


Figure 4 The delay composition for the request under analysis.

5.1 Main Memory Latency Analysis

Based on the proposed MC arbitration, we next determine an upper bound to the delay that the oldest request under analysis r_{ua} can suffer in the main memory. In the worst-case scenario, r_{ua} and all interfering requests require to issue three commands: PRE, ACT, and CAS. Figure 4 shows how to decompose the latency of r_{ua} based on two types of terms: 1) intra-bank constraints between commands issued to the same bank, or between command and data, and 2) inter-bank constraints caused by the same type of command issued to other banks. Since we assume private banks, intra-bank constraints can only be caused by commands of the core that issued r_{ua}, while requests of other cores cause inter-bank constraints. This section aims to derive the worst-case latency components of the three DRAM stages, which we will then use in Section 5 in order to calculate the total worst-case request latency. These components are as follows:

1. **For the PRE stage:** D_{PRE} is the maximum latency of PRE from the time it becomes intra-ready until it is issued, caused by interfering PRE commands of other cores.
2. **For the ACT stage:** D_{ACT} is the maximum latency of ACT from the time it becomes intra-ready until it is issued, caused by interfering ACT commands of other cores.
3. **For the CAS stage:** D_{CAS} is the maximum latency of CAS from when it becomes intra-ready until it is issued, again caused by interfering CAS commands of other cores.

We next derive bounds on $\mathcal{D}_{PRE}$, $\mathcal{D}_{ACT}$ and $\mathcal{D}_{CAS}$ as a function of the numbers N_{PRE}, N_{ACT} and N_{CAS} of higher priority requests whose PRE, ACT and CAS commands, respectively, interfere with the commands of r_{ua}. However, since, as aforementioned applying the pipelining from the delay composition theory to these three DRAM stages is not novel, and the latency components have already been derived in several previous works [15, 43], we do not formally prove their derivation and instead use the values directly from those works. That said, for comprehensiveness in the paper, we intuitively explain each equation.

For the PRE stage, Equation 1 calculates $\mathcal{D}_{PRE}$, which uses the same analysis as in [43] (Equation 2). The intuition behind Equation 1 is as follows. Since there is no inter-bank timing constraint between PRE commands, each interfering PRE contributes one clock cycle of delay; however, we have to add an additional cycle per command to account for the effects of command bus conflicts.

$$\mathcal{D}_{PRE}(N_{PRE}) = 2 \cdot N_{PRE} \cdot t_{clk}^{DRAM} \tag{1}$$

For the ACT stage, ACT commands have two inter-bank timing constraints, t_{RRD}^{l} which applies between successive commands, and t_{FAW} which applies every 4 commands. Hence, the bound must consider the maximum of the two constraints. The value of each timing constraint is increased by one clock cycle to account for bus conflicts caused by CAS commands. That is one difference between Equation 2 below and Equation 3 in [43]. In the latter, each timing constraint is instead increased by two clock cycles because ACT commands can suffer bus conflicts due to both PRE and CAS commands, while for our proposed controller, as

explained in Section 4.2, ACT will not suffer command bus contention from PRE. Finally, in Equation 2, $t_{FAW} - 3 \cdot t^l_{RRD} - t^{DRAM}_{clk}$ represents the maximum delay caused by ACT commands of lower-priority requests issued as late as possible before the ACT of r_{ua} becomes intra-ready.

$$\mathcal{D}_{ACT}(N_{ACT}) = t_{FAW} - 3 \cdot t^l_{RRD} - t^{DRAM}_{clk} + \max\Big(N_{ACT} \cdot (t^l_{RRD} + t^{DRAM}_{clk}), \lfloor N_{ACT}/4 \rfloor \cdot (t_{FAW} + t^{DRAM}_{clk}) + (N_{ACT}\%4) \cdot (t^l_{RRD} + t^{DRAM}_{clk})\Big) \tag{2}$$

Finally, **for the CAS stage,** Equations 3 and 4 calculate $\mathcal{D}_{CAS}$ for read (RD) and write (WR) requests, respectively. Since in our controller, CAS commands have the highest priority for accessing the command bus; they cannot suffer command bus conflicts. Also, since in our controller RD and WR commands are scheduled fairly, contrary to [43] and similar to [15], the N_{CAS} interfering commands can comprise both RD and WR. Inter-bank timing constraints between CAS commands are longer when switching between RD-to-WR (t_{RTW}) and WR-to-RD ($t_{WL} + t_{Bus} + t_{WTR}$) compared to issuing two RD or two WR commands back-to-back (t^l_{CCD}); this is because the data bus needs this time to change the direction of the data sent on the bus. Hence, to bound the worst-case latency for a CAS command, we consider the maximum alternation between RD and WR commands. Again, a lower-priority CAS command can be issued one clock cycle before the CAS of r_{ua} becomes intra-ready; hence, the total number of interfering requests is $N_{CAS} + 1$. Noticing that the last constraint must be a WR-to-RD switch if r_{ua} is an RD (Equation 3), and an RD-to-WR switch if r_{ua} is a WR (Equation 4).

$$\mathcal{D}^{RD}_{CAS}(N_{CAS}) = \left\lfloor \frac{N_{CAS}+1}{2} \right\rfloor \cdot t_{RTW} + \left\lceil \frac{N_{CAS}+1}{2} \right\rceil \cdot (t_{WL} + t_{Bus} + t_{WTR}) - t^{DRAM}_{clk} \tag{3}$$

$$\mathcal{D}^{WR}_{CAS}(N_{CAS}) = \left\lceil \frac{N_{CAS}+1}{2} \right\rceil \cdot t_{RTW} + \left\lfloor \frac{N_{CAS}+1}{2} \right\rfloor \cdot (t_{WL} + t_{Bus} + t_{WTR}) - t^{DRAM}_{clk} \tag{4}$$

In addition to the latency of each of these stages, there are additional across-stage delays a request can suffer. These are as follows. 1) t_{INIT} is the worst-case latency from the time r_{ua} arrives at the MC to PRE becoming intra-ready. In the worst case depicted in the figure, a non-oldest request r of the same core as r_{ua} could issue an ACT command one cycle before r_{ua} arrives at the MC. In such a case, although r_{ua} preempts r, the ACT command imposes an ACT-to-PRE timing constraint t_{RAS}. Thus, in the worst case we have $t_{INIT} = t_{RAS} - t^{DRAM}_{clk}$. Additionally, intra-bank constraints impact when a request can become ready at a particular stage. Namely, 2) t_{RP} is the PRE-to-ACT timing constraint, and 3) t_{RCD} is the ACT-to-CAS timing constraint. We now show how to use all these components to derive the total end-to-end worst-case latency of a memory request.

5.2 Holistic Memory Latency Bound

To maximize L_{ua}, we assume that r_{ua} becomes oldest at the earliest possible time $t^a_{r_{ua}}$. Since we have several request types for r_{ua} as well as for interfering requests. Proving the worst-case L_{ua} for all scenarios is impossible within the paper space. Instead, we developed a brute-force algorithm that covers all possible scenarios and calculates their latency to ensure that we correctly compute the latency bound for every request type and values of timing parameters and corresponding valid values of $N_{REQ}, N_{SBUS}, N_{PRE}, N_{ACT}, N_{CAS}, N_{RESP}, N_{BANK}$[1]. It

[1] source code is available here: `https://gitlab.com/FanosLab/endtoend_wcl_cases_matlab/`

is worth noting that the running time of this brute-force machinery is in the range of few seconds. That said, it is done offline; therefore, the exact run-time complexity is irrelevant to the proposed solution. In the rest of this section, we focus on deriving the global worst-case L_{ua} across all scenarios, which corresponds to a r_{ua} of type $\mathcal{T}_4$, which traverses the most stages and has the highest latency for our system. First note that as pointed out in Section 3.4, requests of type $\mathcal{T}_4$ (as well as those of type $\mathcal{T}_2, \mathcal{T}_3$) execute simultaneously on BANK and RESP, rather than traversing a linear sequence of stages. Since BANK and RESP are the last stages on which r_{ua} executes, its latency depends on which of the two stages it last finishes execution on. If r_{ua} finishes executing on BANK after RESP, then we can obtain a latency bound by analyzing its execution along path (REQ, SBUS, PRE, ACT, CAS, BANK); otherwise, by analyzing path (REQ, SBUS, PRE, ACT, CAS, RESP). Therefore, we can compute L_{ua} by applying the delay composition analysis to both stage sequences and taking the maximum obtained bound.

Second, as discussed in Section 2.3, we use the improved delay composition analysis in [22] to support requests of different types. This analysis's key idea is to split every higher priority interfering request r into a set of *segments*: each segment represents the execution of r on a sequence of consecutive stages in the path of the request under analysis encountered either in the same or exactly in reverse order. We consider the path (REQ, SBUS, PRE, ACT, CAS, RESP) as an example since we find it to lead to the maximum possible L_{ua} in our system. Then, the following segments must be considered:

- Each request of type $\mathcal{T}_1, \mathcal{T}_2$ or $\mathcal{T}_5$ is split into a segment (REQ) and a segment (RESP). Note that no segment can represent execution on BANK since this stage is not part of the analyzed path of r_{ua}.
- Each request of type $\mathcal{T}_3$ is split into a segment (REQ, SBUS) and a segment (RESP).
- Each request of type $\mathcal{T}_4$ corresponds to a single segment (REQ, SBUS, PRE, ACT, CAS, RESP).
- Each request of type $\mathcal{T}_6$ corresponds to a single segment (SBUS, PRE, ACT, CAS).
- Each request of type $\mathcal{T}_7$ is split into a segment (RESP) and a segment (SBUS, PRE, ACT, CAS).

L_{ua} is obtained by summing the following terms:

1. L_{ua}^{trav}: this is the time required by r_{ua} to traverse its required stages (based on type), assuming it suffers no interference at all. For each stage, this is the maximum time required to move to the next one along the path or the time needed to finish executing the last stage.
2. L_{ua}^{lp}: this is the latency component caused by low-priority requests. For each stage, the maximum interference is caused by a single lower-priority request. In GRROF, such request must start executing no later than one clock cycle before r_{ua} or any higher-priority request becomes ready at that stage, otherwise the arbiter will not select the lower-priority request; therefore, the maximum interference is equal to the execution time of any lower-priority request on that stage minus one clock cycle.
3. L_{ua}^{hp}: this is the latency component caused by higher-priority requests. For every segment, its maximum execution time on any one stage on which it executes.

We begin by discussing L_{ua}^{trav}. Here, we are interested in the time between a request starting execution on a stage and becoming ready to be arbitrated on the next stage. Since the request becomes ready on SBUS immediately after finishing executing on REQ, the time from REQ to SBUS is simply the execution time c_{REQ} on REQ. Similarly, the time to finish executing on RESP is c_{RESP}. However, in the case of DRAM stages, we have to consider the

effect of intra-bank DRAM constraints. As discussed in Section 5.1, the time from PRE to ACT is t_{RP} and the time from ACT to CAS is t_{RCD}. For the time from SBUS to PRE, we have to consider three time components: (a) the time c_{SBUS} required to execute on SBUS; (b) the time t_{cross} required to cross clock domains since the system bus and the DRAM controller use different clocks. In the worst case, we assume that such time equals one clock cycle of the destination domain, i.e., $t_{cross} = 1$. (c) The time $t_{INIT} = t_{RAS} - 1$ required for the request to become ready on PRE after arriving at the memory controller. Hence, the required time is equal to $c_{SBUS} + t_{RAS}$. Finally, for CAS to RESP, the time includes $t_{RL} + t_{BUS}$ to obtain the data after issuing the CAS command, $t_{cross} = 1$ to cross back into the CPU clock domain [2], and c_{SBUS} to send the data back through the DRAM-LLC bus, for a total of $t_{RL} + t_{BUS} + 1 + c_{SBUS}$. Summing over all stages, for our example, we obtain a total of:

$$L_{ua}^{trav} = c_{REQ} + c_{SBUS} + 1 + \big((t_{RAS} - 1) + t_{RP} + t_{RCD} + t_{RL} + t_{BUS}\big) \cdot (t_{CLK}^{DRAM}/t_{CLK}^{CPU}) + c_{SBUS} + c_{RESP}. \tag{5}$$

Next, we consider the interference of lower and higher-priority requests/segments ($L_{ua}^{intf} = L_{ua}^{hp} + L_{ua}^{lp}$). Let $N_{REQ}, N_{SBUS}, N_{PRE}$, N_{ACT}, N_{CAS} to denote the number of segments that interfere on the corresponding stage, subject to the constraint that each segment interferes on only one stage. Then including the effect of a lower-priority request, the total interference on REQ, SBUS and RESP is equal to $\mathcal{D}_{REQ}(N_{REQ}) = c_{REQ} - 1 + N_{REQ} \cdot c_{REQ}$, $\mathcal{D}_{SBUS}(N_{SBUS}) = c_{SBUS} - 1 + N_{SBUS} \cdot c_{SBUS}$, $\mathcal{D}_{RESP}(N_{RESP}) = c_{RESP} - 1 + N_{RESP} \cdot c_{RESP}$; while the interference on PRE, ACT and CAS is equal to $\mathcal{D}_{PRE}(N_{PRE}), \mathcal{D}_{ACT}(N_{ACT})$ and $\mathcal{D}_{CAS}^{RD}(N_{CAS})$ as computed in Equations 1, 2, 3. The total interference is thus calculated by Equation 6 maximized over all possible values of $N_{REQ}, N_{SBUS}, N_{PRE}, N_{ACT}, N_{CAS}, N_{RESP}$.

$$L_{ua}^{intf} = \mathcal{D}_{REQ}(N_{REQ}) + \mathcal{D}_{SBUS}(N_{SBUS}) + \mathcal{D}_{RESP}(N_{RESP}) + \big(\mathcal{D}_{PRE}(N_{PRE}) + \mathcal{D}_{ACT}(N_{ACT}) + \mathcal{D}_{CAS}^{RD}(N_{CAS})\big) \cdot (t_{CLK}^{DRAM}/t_{CLK}^{CPU}) \tag{6}$$

Note that based on our GRROF arbitration, at most $M-1$ requests can have higher priority than r_{ua}. For the system settings employed in our evaluation, Equation 6 is maximized when all $M-1$ requests are of type $\mathcal{T}_7$, yielding $M-1$ segments of type (RESP) with $N_{RESP} = M-1$ and $M-1$ segments of type (SBUS, PRE, ACT, CAS) interfering on CAS (i.e., $N_{CAS} = M-1$), for a resulting interference:

$$L_{ua}^{intf,GRROF} = \mathcal{D}_{REQ}(0) + \mathcal{D}_{SBUS}(0) + + \mathcal{D}_{RESP}(M-1) + \big(\mathcal{D}_{PRE}(0) + \mathcal{D}_{ACT}(0) + \mathcal{D}_{CAS}^{RD}(M-1)\big) \cdot (t_{CLK}^{DRAM}/t_{CLK}^{CPU}). \tag{7}$$

Summing Equations 5 and 7 then yields a latency bound for path (REQ, SBUS, PRE, ACT, CAS, RESP), which again based on our system setting, is the latency bound L_{ua} for requests of type $\mathcal{T}_4$ (Equation 8).

$$L_{ua}^{GRROF} = L_{ua}^{trav} + L_{ua}^{intf,GRROF} \tag{8}$$

[2] Note that for the system in Section 6, we assume that the CPU clock has double the frequency of the DRAM clock and that the two clocks are synchronized. Under such assumption, we can take $t_{cross} = 0$.

Finally, note that when analyzing a path through BANK, the maximum interference is similarly equal to $\mathcal{D}_{BANK}(N_{BANK}) = c_{BANK} - t_{CLK}^{CPU} + N_{BANK} \cdot c_{BANK}$. It is important to consider that here N_{BANK} represents the number of segments that interfere on the *same* LLC bank as r_{ua}. However, a higher-priority request might also target a *different* LLC bank. For example, when $\mathcal{T}(r_{ua}) = \mathcal{T}_1$, a higher priority request r also of type $\mathcal{T}_1$ would yield a single segment (REQ, BANK, RESP) if it targets the same bank as r_{ua}, and two segments (REQ) and (RESP) if it targets a different bank. If $c_{BANK} \geq c_{REQ} + c_{RESP}$, then the first case results in higher interference; otherwise, the second.

5.3 Effect of Additive Latency Approach

After deriving the latency bounds for GRROF and using the pipelining analysis from the delay composition theorem, this section shows how pessimistic the resulting bounds are upon considering resources separately and follow the additive latency approach even when pipelining has been considered at one or more of the components in the system but not at the holistic level of the memory. In doing so, we first derive this bound for two systems that apply the latency additive theorem at different scale. Please note that to derive a safe bound using this approach, we follow the direction discussed in Section 2.2 by assuming the maximum possible delay from younger requests of the same core at each resource. As explained in Section 2.2, this provides a safe bound at the expense of being extremely pessimistic. We first define the two systems as follows. Discrete-RR is a system that deploys traditional RR arbitration at the REQ, RESP, and BANK stages, while it uses the MC model proposed in Section 4.2 deploying the RROF arbitration at the three DRAM stages: PRE, ACT, and CAS. Split-RROF is a system that deploys RROF arbitration locally at DRAM stages using the MC model proposed in Section 4.2 and at the cache stages. However, there is no coordination between the DRAM subsystem and the cache subsystem stages. We use a DRAM with a pipelined stages model for the two systems since the state-of-the-art analysis in DRAM already applies the delay composition theorem [15,16,43]. The Discrete-RR system, on the other hand, is using a traditional RR arbiter at each of the remaining three stages: REQ, BANK, and RESP. In contrast, the Split-GRROF goes one step further and even pipelines these three stages together. The reason for choosing this model is to show that even when pipelining resources at one of the levels and not at the system level, latency bounds are still quite pessimistic.

The latency bound L_{ua} for Discrete-RR, L_{ua}^{DRR} can be calculated as follows. First, the latency of each of the REQ, RESP, BANK, and SBUS stages has to be separately calculated. Since each of these stages adopts a RR arbiter and each core has a maximum of N_{pend} pending requests. The worst-case for r_{ua} is to assume that it arrives at the stage after $N_{pend} - 1$ requests from P_{ua} and that P_{ua} is last in the RR order. This yields a total of $M \cdot N_{pend} \cdot c_s$, where s is any of the REQ, RESP, BANK, or SBUS stages, which includes the execution of r_{ua} itself in s. For the DRAM stages, there is one clock cycle for domain crossing. Afterwards, in contrast to GRROF since the relative priority of requests can no longer be assumed to be the same between the cache and the DRAM subsystems, r_{ua} has to assume that it arrives at the DRAM in worst-case as the last request similar to all other stages. Therefore, it has to wait for $N_{pend}-1$ requests to finish from P_{ua}, which in worst-case are all write requests. This is the second line in Equation 9. Afterwards, r_{ua} itself can suffer $M - 1$ requests from other cores due to RR order. This is the third line in Equation 9. Since DRAM is pipelined, we maximized the delay over the ACT, PRE, and CAS stages, which

happens to be that in worst-case all the $M-1$ interfering requests are contributing to the CAS stage similar to GRROF in Equation 7. Applying the additive latency theorem, this gives a total delay for the Discrete-RR as follows:

$$L_{ua}^{DRR} = M \cdot N_{pend} \cdot (c_{REQ} + c_{SBUS} + c_{BANK}) + 1 + c_{SBUS}$$
$$\big((N_{pend}-1)(\mathcal{D}_{PRE}(0) + t_{RP} + \mathcal{D}_{ACT}(0) + t_{RCD} + \mathcal{D}_{CAS}^{WR}(M-1) + t_{WL} + t_{BUS} + t_{WR}) +$$
$$\mathcal{D}_{PRE}(0) + \mathcal{D}_{ACT}(0) + \mathcal{D}_{CAS}^{RD}(M-1) + t_{RL} + t_{BUS}\big) \cdot (t_{CLK}^{DRAM}/t_{CLK}^{CPU}). \tag{9}$$

The latency bound L_{ua} for Split-RROF, L_{ua}^{SRROF} can be calculated as in Equation 10. One important observation to highlight is that because of the pipelining effect in the cache subsystem, Split-RROF has to account only for interference from $M-1$ requests instead of the $M \cdot N_{pend}$ in Discerete-RR case. However, because Split-RROF does not pipeline the cache and the DRAM subsystems together, a special consideration has to be paid for what requests interfere with r_{ua} in the cache subsystem. In particular, because there are now requests that will hit in the BANK and requests that miss and go to the DRAM, requests will take different paths in the cache pipeline stage. The hit requests will be REQ, BANK, and RESP, while the miss requests (from the cache pipeline perspective) will be REQ, then go off the system (to DRAM), then come back to execute RESP and BANK in parallel. Due to this fact, r_{ua} can indeed suffer interference both at the REQ stage and at the BANK stage. This is accounted for in the second line in Equation 10. The first line represents the traversing latency similar to Equation 5 for GRROF. It basically goes through REQ, SBUS, then cross to DRAM (one cycle for clock domain crossing), and then comes back from DRAM through SBUS and then is processed in BANK. The third and fourth lines are accounting for DRAM interference very similar to Equation 9. The last line accounts for one low-priority request at each stage r_{ua} traverses.

$$L_{ua}^{SRROF} = c_{REQ} + c_{SBUS} + 1 + c_{SBUS} + c_{BANK} +$$
$$\mathcal{D}_{REQ}(M-1) + \mathcal{D}_{SBUS}(0) + \mathcal{D}_{BANK}(M-1) +$$
$$\big((N_{pend}-1)(\mathcal{D}_{PRE}(0) + t_{RP} + \mathcal{D}_{ACT}(0) + t_{RCD} + \mathcal{D}_{CAS}^{WR}(M-1) + t_{WL} + t_{BUS} + t_{WR}) +$$
$$\mathcal{D}_{PRE}(0) + \mathcal{D}_{ACT}(0) + \mathcal{D}_{CAS}^{RD}(M-1) + t_{RL} + t_{BUS}\big) \cdot (t_{CLK}^{DRAM}/t_{CLK}^{CPU}) +$$
$$(c_{REQ}-1) + (c_{SBUS}-1) + (c_{BANK}-1) \tag{10}$$

6 Evaluation Results

We implement the proposed solution as well as the two systems we compare against (Discrete-RR and Split-RROF from Section 5.3) on a cycle-accurate simulation platform integrating the cache subsystem simulator provided in [17] with MCsim as a main memory system simulator [29], in order to mimic the whole path of a request. By this way, we are able to accurately obtain end-to-end latency for memory accesses.

Experimental Setup. Unless otherwise specified, in all our experiments we use a quad-core system clocked at 2.4 GHz, where each core is OoO with up to 16 N_{pend} and a 32 KB 4-way set-associative private L1 cache. Through a split-transaction interconnect, the cores share access to a 4 MB 8-ways set-associative bankized LLC that comprises 8 separate banks. Both L1s and the LLC are write-back write-allocate non-blocking caches with a cache line size of 64 bytes and an LRU replacement policy. The LLC is set-partitioned such that each core has its own private sets. Nonetheless, all cores can access all LLC banks. To map the requests to the different LLC banks, the cache controller uses the Least-Significant-Bits (LSBs) in the

tag address bits to denote the bank number, such that a core can access all banks within its private sets. We use a DDR4-2400U for the main memory with a single-channel single-rank DRAM device. DRAM banks are partitioned such that each core accesses its own private set of banks. We assume that processing L1 hit requests takes a single cycle and processing data in the LLC data array takes ten cycles ($c_{BANK} = 10$). We also configure processing time on request and response buses to 2 and 5 cycles, respectively ($t_{REQ} = 2$, $t_{RESP} = 5$). The system bus between the LLC and the MC has a latency of ($t_{SBUS} = 5$) in either direction LLC-DRAM or DRAM-LLC.

Workloads. We use SPEC CPU benchmark [40]. While running a SPEC workload on one of the cores, the other cores are running a stressing microbenchmark to generate the most interference on the task under analysis. For these stressors, we use the *latency* benchmark from the IsolBench suite [41].

Compared Systems. In addition to the two real-time systems of (Discrete-RR and Split-RROF, we compare GRROF against COTS high performance arbiter using FRFCFS arbitration for all resources.

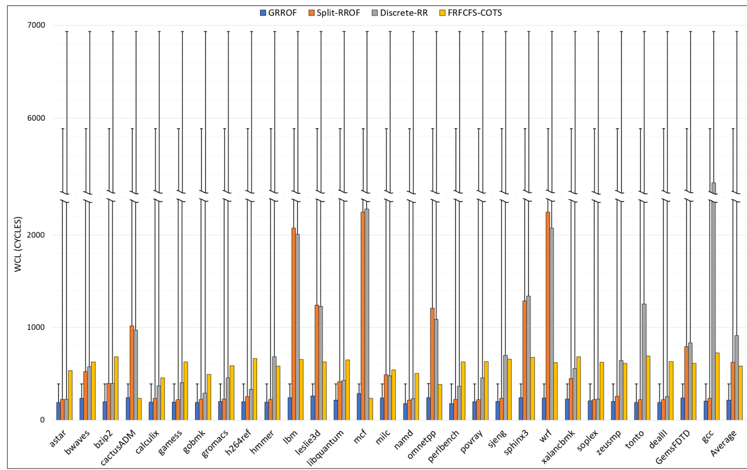


Figure 5 Per-request worst-case latency both Observed (solid bars) and analytical (T-shape) for SPEC workloads.

6.1 Per-Request Worst-Case Latency

Figure 5 delineates both the observed and analytical WCL suffered by any memory request for all the systems. Note that the high performance FRFCFS is theoretically unbounded (assuming that there is no threshold for requests reordering), thus there is no estimated bound for it. The figure shows that: 1) the analytical bounds for Split-RROF and Discrete-RR are very pessimistic. Compared to the observed WCL, they reach up to 26× (*namd*) and 30× (*astar*) for Split-RROF and Discrete-RR, respectively. This clearly shows the pessimism of the additive latency approach, as discussed throughout the paper. 2) GRROF manifests the lowest observed WCL per-request for all the workloads. For the worst-case observed latency across all the workloads, GRROF shows 8× and 18.4× reduction compared to Split-RROF and Discrete-RR, respectively. 3) The analytical bound of GRROF is the tightest latency bound which does not exceed than 2× of the experimental latency (*namd*). In fact, GRROF

achieves a 14.8 and 17.5 reduction on the analytical latency bound compared to Split-RROF and Discrete-RR, respectively. This experiment clearly emphasizes that using conventional per-resource real-time arbitration schemes alongside the additive latency approach suffers from excessively pessimistic latency bounds and that coordinating these resources such that pipelining analysis can be applied has a huge potential as an alternative.

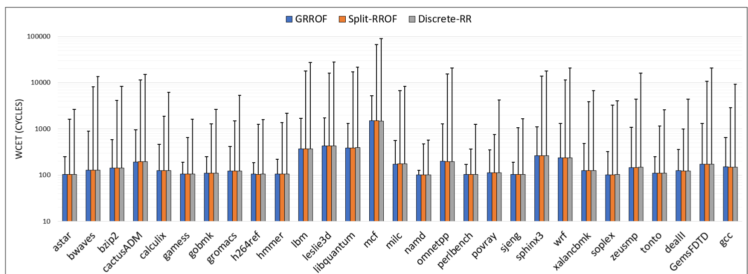


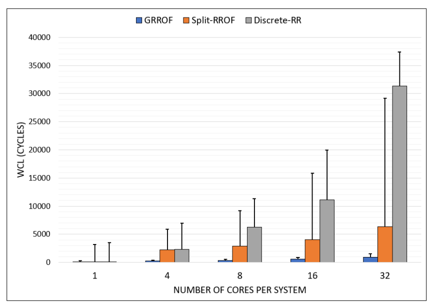
Figure 6 Per-task observed memory latency for SPEC benchmarks (solid bars), compared to the analytical memory processing time (T-shape). Values in y-axis are in logarithmic scale.

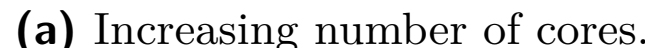
6.2 Per-Task Worst-Case Memory Latency

In this experiment, we evaluate the total task Memory Latency and compute the analytical total task's worst-case memory latency (WCML). Figure 6 shows the experimental total memory latency for SPEC benchmarks. Additionally, it shows the analytical WCML as T-bar. The analytical bound for each task is obtained by summing up the following components: 1) the number of L1 hit requests multiplied by the L1 hit latency, 2) the number of LLC hit requests multiplied by the WCL of a request hitting in LLC (this is type $\mathcal{T}_1$ in this case), and 3) the number of DRAM access requests multiplied by WCL of a miss (this is type $\mathcal{T}_4$ as driven in Equation 8). Please note that write-backs are not considered in this task analysis since they are neither stalling core pipeline nor in the critical path of the requests based on the considered system architecture in Section 3.1. From the figure, we observe that, the observed total memory time for the three systems are very close for the SPEC benchmarks. When investigating the reason for this, we found that most of the SPEC BMs exhibit a very high L1 hit rate. However and more importantly from a real-time prespective, in terms of predictability, the calculated bounds for Split-RROF and Discrete-RR are drastically pessimistic. In case of Split-RROF, the analytical WCML varies between 2×-36× of the observed latency. And for Discrete-RR, it varies between 4×-48×. This wide variability makes them poor in predictability and entails bounds not very useful. By looking at GRROF bounds, on the other hand, it is clear that it provides the tightest WCML, which does not overrun 1.5× and can be as close as 16% of the actual experimental latency.

6.3 Sensitivity Test

In this section, we conduct two experiments that study the request worst-case latency while: 1) increasing number of cores in the system (Figure 7a), and 2) increasing the size of MSHR buffer entries (and hence, the $N_{pending}$ from each core) (7b). In both figures, we show the results for only one SPEC benchmark (*mcf*). We observe similar trend for all the other benchmarks. For the first experiment, we experiment with 1, 4, 8, 16 and 32 core systems. Figure 7a emphasizes that the bound of GRROF increases linearly with number of cores. Likewise the previous experiment, we run SPEC workload aside with a stressing workloads. It




(a) Increasing number of cores.

(b) Increasing N_{pend}.

Figure 7 Per-request worst-case latency of *mcf* workload from SPEC benchmark. Column and T-shape bars denote experimental and analytical values, respectively.

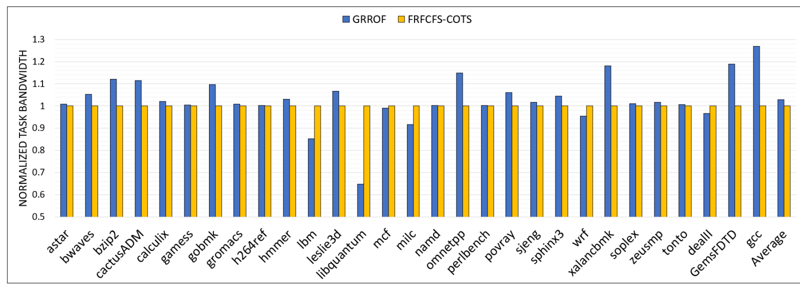


Figure 8 The bandwidth of SPEC workloads in a quad-core system.

is clear that GRROF's estimated latency is tightly bounded and very close to the experimental results, which does not exceed 0.6× of the observed WCL, in the most interfering setup of 32-cores. However, Split-RROF and Discrete-RR systems shows very large execution time which increases dramatically with increasing number of interfering cores, up to 8× and 32× of the observed WCL of GRROF. For the second experiment, it conveys the results for increasing number of N_{pend} entries as 1, 4, 8 and 16. In Figure 7b, the latency for GRROF is fixed and independent of the number of the N_{pend} entries. This is because, regardless of N_{pend}, GRROF ensures that the latency of any request can suffer interference delays from only one request from every other core as shown in Section 5. On the other hand, although Discrete-RR can provide a bound on the WCL for memory accesses, it is quadratically increased by the increasing number of outstanding requests on OoO systems. Requests may suffer up to 165× of their latency on an IO system. Split-RROF reduces this large variance, however the requests can suffer up to 12× of their latency on an IO system.

6.4 Average Performance

In this experiment, we evaluate the average-performance of GRROF. Figure 8 shows the average memory bandwidth of SPEC benchmarks running on GRROF and FRFCFS-COTS, and normalized on FRFCFS-COTS performance. Comparing the average-performance with the high-performance system, we make these observation points: 1) the memory bandwidth of GRROF is on-par with the COTS solution and performs even slightly better (2.9%) on average results. The reason for this improvement is that GRROF introduces more fairness to all benchmarks by prioritizing the oldest requests from all cores over younger ones. This protects tasks from severe interference from other co-running aggressor tasks.

7 Conclusions

In this paper, we introduce a coordinating management mechanism for the holistic memory system in order to sustain priorities of requests across all the shared resources in the memory system. By virtue of this novel mechanism, we could tightly bound the estimated per-request and per-task memory latency. And by comparing the proposed solution to the conventional real-time solutions, we made the point their analysis model is not convenient nor reliable for tightly bounding the latencies. In addition, we show that our system drastically reduces the WCL with more than 18× reduction in memory latency and tightly bounds the estimated latency by not exceeding 16% of the experimental latency.

References

1. Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: A predictable sdram memory controller. In *2007 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 251–256, 2007. doi:10.1145/1289816.1289877.
2. Intel® iris® plus graphics and uhd graphics open source. programmer's reference manualintel® 64 and ia-32 architectures optimization reference manual. https://01.org/sites/default/files/documentation/intel-gfx-prm-osrc-icllp-vol07-memory_cache_0.pdf.
3. ARM. Arm926ej-s™ revision: r0p5 technical reference manual. https://developer.arm.com/documentation/ddi0198/e, 2008.
4. ARM. Cortex-m4 technical reference manual r0p0. https://developer.arm.com/documentation/ddi0439/b, 2010.
5. Michael G Bechtel and Heechul Yun. Denial-of-service attacks on shared cache in multicore: Analysis and prevention. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 357–367, 2019. doi:10.1109/RTAS.2019.00037.
6. Leonardo Ecco, Sebastian Tobuschat, Selma Saidi, and Rolf Ernst. A mixed critical memory controller using bank privatization and fixed priority scheduling. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10, 2014. doi:10.1109/RTCSA.2014.6910550.
7. Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, and Lothar Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 1–15, 2013. doi:10.1109/EMSOFT.2013.6658595.
8. G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni. A survey on cache management mechanisms for real-time embedded systems. *ACM Comput. Surv.*, 48(2), 2015. doi:10.1145/2830555.
9. Danlu Guo, Mohamed Hassan, Rodolfo Pellizzoni, and Hiren Patel. A comparative study of predictable DRAM controllers. *ACM Transactions on Embedded Computing Systems (TECS)*, 2018. doi:10.1145/3158208.
10. 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)*, pages 299–308, 2016. doi:10.1145/2997465.2997471.
11. Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches. In *2009 30th IEEE Real-Time Systems Symposium (RTSS)*, pages 68–77, 2009. doi:10.1109/RTSS.2009.34.
12. Mohamed Hassan. Heterogeneous mpsocs for mixed-criticality systems: Challenges and opportunities. *IEEE Design & Test*, pages 47–55, 2017. doi:10.1109/MDAT.2017.2771447.

13 Mohamed Hassan. Disco: Time-compositional cache coherence for multi-core real-time embedded systems. *IEEE Transactions on Computers (TC)*, pages 1163–1177, 2022. doi:10.1109/TC.2022.3193624.

14 Mohamed Hassan and Hiren Patel. Criticality- and requirement-aware bus arbitration for multi-core mixed criticality systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11, 2016. doi:10.1109/RTAS.2016.7461327.

15 Mohamed Hassan and Rodolfo Pellizzoni. Bounding dram interference in cots heterogeneous mpsocs for mixed criticality systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, pages 2323–2336, 2018. doi:10.1109/TCAD.2018.2857379.

16 Mohamed Hassan and Rodolfo Pellizzoni. Analysis of memory contention in heterogeneous cots mpsocs. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 23:1–23:24, 2020. doi:10.4230/LIPIcs.ECRTS.2020.23.

17 Salah Hessien and Mohamed Hassan. The best of all worlds: Improving predictability at the performance of conventional coherence with no protocol modifications. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 218–230, 2020. doi:10.1109/RTSS49844.2020.00029.

18 Mohamed Hossam and Mohamed Hassan. Predictably and efficiently integrating cots cache coherence in real-time systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 17:1–17:23, 2022. doi:10.4230/LIPIcs.ECRTS.2022.17.

19 Intel. Write combining memory implementation guidelines. https://download.intel.com/design/PentiumII/applnots/24442201.pdf, 1998.

20 Javier Jalle, Eduardo Quinones, Jaume Abella, Luca Fossati, Marco Zulianello, and Francisco J Cazorla. A dual-criticality memory controller (dcmc): Proposal and evaluation of a space case study. In *2014 IEEE Real-Time Systems Symposium (RTSS)*, pages 207–217, 2014. doi:10.1109/RTSS.2014.23.

21 Praveen Jayachandran and Tarek Abdelzaher. Delay composition in preemptive and non-preemptive real-time pipelines. *Real-Time Systems*, pages 290–320, 2008. doi:10.1007/s11241-008-9056-3.

22 Praveen Jayachandran and Tarek F. Abdelzaher. End-to-end delay analysis of distributed systems with cycles in the task graph. In *In Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS)*, pages 13–22, 2009. doi:10.1109/ECRTS.2009.15.

23 Anirudh M. Kaushik, Mohamed Hassan, and Hiren Patel. Designing Predictable Cache Coherence Protocols for Multi-Core Real-Time Systems. *IEEE Transactons on Computers (TC)*, pages 1–23, 2020. doi:10.1109/TC.2020.3037747.

24 Anirudh Mohan Kaushik, Paulos Tegegn, Zhuanhao Wu, and Hiren Patel. Carp: A data communication mechanism for multi-core mixed-criticality systems. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 419–432, 2019. doi:10.1109/RTSS46320.2019.00044.

25 Ondrej Kotaba, Jan Nowotsch, Michael Paulitsch, Stefan M. Petters, and Henrik Theiling. Multicore in real-time systems – temporal isolation challenges due to shared resources. In *Design, Automation and Test in Europe (DATE)*, 2013.

26 David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture (ISCA)*, pages 81–87, 1981.

27 NG Chetan Kumar, Sudhanshu Vyas, Ron K Cytron, Christopher D Gill, Joseph Zambreno, and Phillip H Jones. Cache design for mixed criticality real-time systems. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pages 513–516, 2014. doi:10.1109/ICCD.2014.6974730.

28 Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54, 2013. doi:10.1109/RTAS.2013.6531078.

29 Reza Mirosanlou, Danlu Guo, Mohamed Hassan, and Rodolfo Pellizzoni. Mcsim: An extensible dram memory controller simulator. *IEEE Computer Architecture Letters (LCA)*, pages 105–109, 2020. doi:10.1109/LCA.2020.3008288.

30 Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Drambulism: Balancing performance and predictability through dynamic pipelining. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 82–94, 2020. doi:10.1109/RTAS48715.2020.00-15.

31 Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Duetto: Latency guarantees at minimal performance cost. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1136–1141, 2021. doi:10.23919/DATE51398.2021.9474062.

32 Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. DuoMC: Tight DRAM Latency Bounds with Shared Banks and Near-COTS Performance. In *ACM International Symposium on Memory Systems (MEMSYS)*, pages 1–14, 2021. doi:10.1145/3488423.3519322.

33 Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Parallelism-Aware High-Performance Cache Coherence with Tight Latency Bounds. In *34th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 16:1–16:27, 2022. doi:10.4230/LIPIcs.ECRTS.2022.16.

34 Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 146–160, 2007. doi:10.1109/MICRO.2007.21.

35 Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *ACM SIGARCH Computer Architecture News (ISCA)*, pages 63–74, 2008. doi:10.1109/ISCA.2008.7.

36 Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for cots-based embedded systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 269–279, 2011. doi:10.1109/RTAS.2011.33.

37 Martin Schoeberl, Wolfgang Puffitsch, and Benedikt Huber. Towards time-predictable data caches for chip-multiprocessors. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems (SEUS)*, pages 180–191, 2009. doi:10.1007/978-3-642-10265-3_17.

38 DDR4 SDRAM Standard, JEDEC JESD79-4, 2012.

39 John Paul Shen and Mikko H Lipasti. *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

40 Sarabjeet Singh and Manu Awasthi. Memory centric characterization and analysis of spec cpu2017 suite. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 285–292, 2019. doi:10.1145/3297663.3310311.

41 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)*, pages 1–12, 2016. doi:10.1109/RTAS.2016.7461361.

42 Man-Ki Yoon, Jung-Eun Kim, and Lui Sha. Optimizing tunable wcet with shared resource allocation and arbitration in hard real-time multicore systems. In *2011 IEEE 32nd Real-Time Systems Symposium (RTSS)*, pages 227–238, 2011. doi:10.1109/RTSS.2011.28.

43 Heechul Yun, Rodolfo Pellizzoni, and Prathap Kumar Valsan. Parallelism-aware memory interference delay analysis for cots multicore systems. In *2015 27th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 184–195, 2015. doi:10.1109/ECRTS.2015.24.

Replication-Based Scheduling of Parallel Real-Time Tasks

Federico Aromolo
Scuola Superiore Sant'Anna, Pisa, Italy

Geoffrey Nelissen
Eindhoven University of Technology, Eindhoven, The Netherlands

Alessandro Biondi
Scuola Superiore Sant'Anna, Pisa, Italy

Abstract

Multiprocessors have become the standard computing platform for real-time embedded systems. To efficiently leverage the computational power of such platforms, software tasks are often characterized by an internal structure where concurrent subtasks can execute in parallel on different processors. Existing strategies for the scheduling of parallel real-time tasks on multiprocessor platforms, such as partitioned, global, and federated scheduling, were inspired by earlier techniques that were not conceived to explicitly support parallel tasks, thus carrying advantages but also well-known limitations. This paper introduces replication-based scheduling, a specialized scheduling paradigm for parallel real-time DAG tasks. Replication-based scheduling leverages the internal structure of the parallel tasks to assign replicas of the subtasks to different processors, while ensuring that exactly one replica of each subtask will be executed at runtime for every task instance. This approach aims at preserving the advantages of partitioned scheduling while simplifying the timing analysis. The replication-based scheduling framework is first defined, together with a strategy for implementing replication-based scheduling in real-time operating systems. Then, offline allocation strategies for subtask replicas and a response-time analysis are presented. In the provided experiments, the schedulability achieved with replication-based scheduling is compared with that of existing techniques for the scheduling of parallel real-time tasks on multiprocessors.

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

Keywords and phrases Real-Time Systems, Scheduling Algorithms, Schedulability Analysis, Parallel Tasks

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.18

Acknowledgements The authors would like to thank José Fonseca for the fruitful discussions that triggered the idea for the approach presented in the paper.

1 Introduction

With the emergence of multiprocessor systems as the standard enabling platform for high-performance real-time embedded computing systems, computational workloads have evolved towards highly parallel structures to match the enhanced processing capabilities offered by the underlying hardware. Numerous models exist to capture and analyze the timing behavior of the scheduling system and guide the allocation of the computational activities to the available processing elements, both at design time and at runtime, in order to maximize resource usage while ensuring timely execution of all software activities in the system. However, existing scheduling solutions for parallel tasks are characterized by either achieving low resource utilization levels, or by excessive complexity in their runtime behavior and implementation, leading to conservative analyses and significant runtime overheads [9, 8, 11, 23, 19].

© Federico Aromolo, Geoffrey Nelissen, and Alessandro Biondi;
licensed under Creative Commons License CC-BY 4.0

Contributions. This paper presents the replication-based scheduling paradigm (RBS) for managing the execution of sporadic parallel real-time tasks on multiprocessor computing platforms, which aims at improving the achieved system utilization and schedulability performance by employing a flexible allocation and execution scheme based on subtask replication. The main contributions are as follows:

1. Definition of the replication-based scheduling approach and discussion on its distinguishing features in comparison with existing approaches.
2. Description of a pattern of implementation of replication-based scheduling in real-time operating systems.
3. Real-time analysis for parallel tasks executing on a multiprocessor platform under preemptive fixed-priority replication-based scheduling.
4. Experimental evaluation of the schedulability performance of replication-based scheduling, in comparison with existing techniques for the scheduling of real-time parallel tasks.

2 System model

We consider a set $\tau = \{\tau_1, \ldots, \tau_n\}$ of n sporadic parallel real-time tasks, to be scheduled on a multiprocessor platform consisting of m identical processors $P_1, \ldots, P_m$ under preemptive fixed-priority scheduling. Each task τ_i releases a potentially infinite sequence of jobs, each separated from the next by at least a minimum inter-arrival time T_i, and subject to a constrained relative deadline D_i, such that $D_i \leq T_i$. The parallel computational structure of each task τ_i is modeled as a directed acyclic graph (DAG) $G_i = (V_i, E_i)$, where $V_i = \{v_{i,1}, v_{i,2}, \ldots, v_{i,n_i}\}$ is a set of n_i nodes (or vertices), and $E_i \subseteq V_i \times V_i$ is a set of directed edges between nodes in V_i. Each node $v_{i,a} \in V_i$ represents a sequential computational unit, or subtask, of the task τ_i, and is characterized by a worst-case execution time (WCET) $C_{i,a}$. Each edge in E_i represents a precedence constraint between two nodes of the DAG G_i. If $e_i^{a,b} = (v_{i,a}, v_{i,b})$ is an edge connecting the vertices $v_{i,a}$ and $v_{i,b}$, then, for every job of τ_i, subtask $v_{i,b}$ cannot execute before $v_{i,a}$ is completed. Each task τ_i is assigned an unique fixed scheduling priority π_i. Subtasks inherit the priority of the corresponding task. The set of tasks with priorities higher than or equal to that of a task τ_i, excluding τ_i itself, is denoted by $\text{hep}(\tau_i)$. Overall, a task τ_i is characterized by the tuple (G_i, T_i, D_i, π_i).

The cumulative worst-case execution time (WCET) C_i of a task τ_i is defined as $C_i = \sum_{v_{i,a} \in V_i} C_{i,a}$. The utilization factor U_i of τ_i is defined as $U_i = C_i / T_i$. The response time of a job of a task τ_i is defined as the difference between its finishing time, that is, the time at which the job completes its execution, and its arrival time. The worst-case response time (WCRT) R_i of a task τ_i is defined as the maximum response time across all possible jobs of τ_i in all possible schedules of task set τ, with respect to the adopted scheduling algorithm. Analogously, the response time of an instance of a subtask $v_{i,a}$ within a job of τ_i is defined as the difference between the finishing time of that instance and the arrival time of the corresponding job, while the WCRT $R_{i,a}$ of a subtask $v_{i,a}$ is defined as the maximum possible response time of $v_{i,a}$ across all possible jobs of τ_i in all possible schedules of task set τ.

Whenever an edge from $v_{i,a}$ to $v_{i,b}$ exists, $v_{i,a}$ is said to be an immediate predecessor of $v_{i,b}$, whereas $v_{i,b}$ is said to be an immediate successor of $v_{i,a}$. The set of immediate predecessors of $v_{i,a}$ is denoted by $\text{ipred}(v_{i,a})$, while the set of immediate successors of $v_{i,a}$ is denoted by $\text{isucc}(v_{i,a})$. When the immediate predecessor and the immediate successor definitions are applied transitively starting from a node $v_{i,a}$ over the topology of the DAG G_i, the set of predecessors and the set of successors of $v_{i,a}$ are obtained, respectively. Two different nodes are said to be independent from each other if neither is a predecessor or a

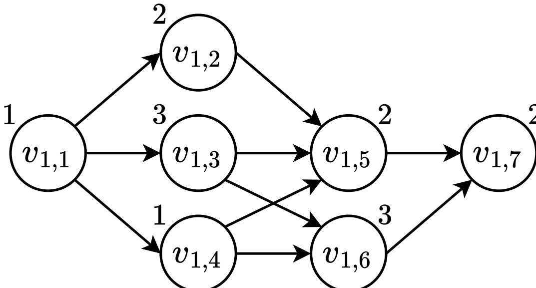


Figure 1 Example of computational structure of a parallel task τ_1, modeled as a DAG G_1.

successor of the other. A node with no incoming edge is referred to as a *source* node, while a node with no outgoing edge is referred to as a *sink* node. The set of sink nodes in a DAG G_i is denoted by $\text{sink}(G_i)$. In the following we assume, without loss of generality, that a single source node, denoted by $v_{i,S}$, is present in each DAG G_i. A *path* in a DAG G_i is defined as an ordered sequence of nodes where a directed edge exists between any two adjacent nodes in the sequence, each node in the sequence is an immediate predecessor of the following node, and the sequence starts from a source node and ends on a sink node. Given a path λ, $V(\lambda)$ represents the set of nodes belonging to the path. The set of all paths in a DAG G_i is denoted by $\text{path}(G_i)$.

Running example. Figure 1 depicts the DAG topology G_1 of an example parallel task τ_1 composed of $n_1 = 7$ nodes. In the figure, the WCET $C_{1,a}$ of each subtask $v_{1,a}$ is reported next to the corresponding node.

2.1 Scheduling requirements

Designing a suitable scheduling paradigm for parallel tasks requires satisfying the following requirements for each task set τ scheduled under that paradigm.

- **Requirement 1.** For each task $\tau_i \in \tau$, in all jobs of τ_i, the precedence constraints in G_i must be properly enforced, meaning that each node in G_i cannot start executing before all of its predecessors have completed.
- **Requirement 2.** For each task $\tau_i \in \tau$, in all jobs of τ_i, each node in G_i must execute exactly once.

3 Background and motivations

Several parallel task models have been proposed in the literature to represent the different forms of workload generated by well-known parallel programming models. In the fork-join model [22, 27, 4], tasks are represented as an interleaved sequence of sequential and parallel segments, where synchronization is assumed at the boundary of every segment. The sporadic DAG model was introduced by Saifullah et al. [28] to support less restrictive parallel structure structures. A number of works demonstrated that the DAG task model resembles commonly used parallel programming models such as OpenMP [30, 25].

The following techniques are considered the primary options when dealing with the scheduling of sporadic DAG tasks on multiprocessor platforms.

Global scheduling. Under fixed-priority global scheduling, the m highest-priority pending subtasks are scheduled at any given time. If tasks are preemptive, the execution of a low-priority subtask may be preempted by a higher-priority one. When resuming its execution, the preempted subtask may migrate to a different processor than that on which it was preempted. Therefore, global scheduling requires the underlying system to be capable of moving the execution context of a job from one processor to another. Migrations are typically costly and increase the execution time of jobs. An advantage of global scheduling is that all scheduling decisions are taken at runtime, and no design time resource assignment is required. Global scheduling for parallel tasks was investigated in numerous works, such as those by Bonifaci et al. [12], Baruah [5], and Fonseca et al. [17, 18].

Partitioned scheduling. Under partitioned scheduling, each subtask is statically assigned to a specific processor at design time, and can only execute on that processor at runtime. Execution of the subtasks allocated to each processor is then managed by a dedicated uniprocessor scheduler. As a result, the partitioned scheduling approach entails solving a complex allocation problem to map subtasks on processors. On the other hand, partitioned scheduling can be easily implemented in a real-time operating system by reusing techniques from uniprocessor scheduling. Parallel tasks under partitioned scheduling were analyzed by Fonseca et al. [19], Casini et al. [13], and Aromolo et al. [2] by means of model transformation techniques to self-suspending task models [14].

Federated scheduling. Federated scheduling, originally proposed by Li et al. [23], splits the task set into two disjoint sets: the set of *high-utilization tasks*, which contains all tasks τ_i such that $U_i \geq 1$, and the set of *low-utilization tasks*, which contains all tasks τ_i such that $U_i < 1$. The two sets of tasks are then treated separately. First, each high-utilization task τ_i is assigned a set of $m_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$ *dedicated* processors, with $L_i = \max_{\lambda \in \mathrm{path}(G_i)} \left\{ \sum_{v_{i,a} \in V(\lambda)} C_{i,a} \right\}$. Each high-utilization task is scheduled on its dedicated m_i processors by any work-conserving scheduler. Low-utilization tasks are treated as sequential tasks and executed with any multiprocessor scheduling algorithm for sequential tasks on the processors that were not assigned to high-utilization tasks. Subsequent works, such as those by Baruah [6, 7], Jiang et al. [21], Ueter et al. [29], Dinh et al. [15], and Jiang et al. [20], explored the application of federated scheduling under different assumptions.

3.1 Motivations

The scheduling approaches mentioned above come with both advantages (e.g., simplicity or load balancing) and disadvantages (e.g., resource over-provisioning or limited analyzability), which were properly documented in previous work [9, 8, 11, 2]. It is worth highlighting the benefits of partitioned scheduling, which include the possibility of accurately controlling the contention for memory resources and the typically lower overheads implied by its implementations compared to schedulers that support job migrations. Nevertheless, partitioned scheduling of parallel tasks proved to introduce significant complexity in the response-time analysis, which inevitably also affects the performance of partitioning algorithms [19].

Motivated by these observations, this work investigates a specialized scheduling approach for parallel tasks that aims at preserving the overall philosophy of partitioned scheduling on a per-job basis, while at the same time drastically simplifying the timing analysis.

The proposed replication-based scheduling algorithm leverages the internal structure of each parallel task to assign replicas of its nodes to different processors, while ensuring that exactly one replica of each node will be executed at runtime for every job.

Other scheduling approaches leveraging replication have been investigated in the context of high-performance computing, with the aim of reducing communication costs and improving the expected response times, but are characterized by different scheduling behaviors. For instance, duplication-based scheduling [1] statically assigns nodes of parallel tasks redundantly to different processors in order to minimize the overheads incurred due to inter-processor communication. Duplication-based scheduling was also adopted to devise a specialized partitioning strategy for real-time parallel DAG tasks which aims at eliminating inter-processor dependencies between subtasks, thus simplifying the resulting schedulability analysis [16]. However, in duplication-based approaches, all copies of each node are executed for each job, whereas the replication-based scheduling approach ensures that exactly one node replica executes for each job, meaning that the overall computational workload of the task is not increased. Concerning distributed server systems, replication-based load balancing techniques were proposed to minimize the expected response time for the incoming job requests by creating multiple replicas of the job on different servers [26]. Unlike replication-based load balancing, which aims at minimizing the expected latency for job requests in distributed systems, our solution focuses on ensuring that precedence constraints and real-time properties are satisfied in the scheduling of parallel real-time tasks.

4 Replication-based scheduling

This section presents the replication-based scheduling strategy for parallel tasks, for the specific case of preemptive fixed-priority systems. The aim of the proposed replication-based scheduling paradigm is to mitigate the limitations and performance loss suffered by existing techniques by leveraging the knowledge of the internal computational structure of the parallel tasks, in terms of their DAG topology.

4.1 Overview

As with partitioned and federated scheduling, replication-based scheduling consists of two phases; namely, an allocation phase at design time, and a dynamic scheduling phase at runtime. The core feature of replication-based scheduling is that, during the allocation phase, each node of the DAG of a given parallel task can be replicated and made available for execution on a subset of processors. For each job, a single replica of each node is then selected for execution at runtime, depending on the dynamic scheduling situation.

4.1.1 Allocation phase

In the system design phase, the computational parallel structure of each task is first decomposed into a set of linear node sequences by means of a specialized *sequence expansion* algorithm, specified later in Section 4.2. Each node sequence generated by this algorithm represents a *subpath* in the DAG which should be executed sequentially and *without suspension* on a single processor. Then, in a *sequence allocation* step, each node sequence is allocated to a specific processor, meaning that it can only execute on that specific processor at runtime. In the following, let $S_{i,q} = \langle v_{i,a}, v_{i,b}, \dots \rangle$ represent an ordered sequence of nodes of task τ_i, and let $\mathcal{S}_i = \{S_{i,1}, S_{i,2}, \dots\}$ represent the set of node sequences generated by the sequence expansion algorithm for a task τ_i. Then, $P(S_{i,q})$ represents the processor to which a sequence $S_{i,q}$ of a task τ_i is assigned.

Running example. Figure 2 illustrates the four linear node sequences $\{S_{1,1}, S_{1,2}, S_{1,3}, S_{1,4}\}$ obtained for the example task τ_1 of Figure 1 with the *sequence expansion* algorithm specified later in Section 4.2. Sequence $S_{1,1} = \langle v_{1,1}, v_{1,2}, v_{1,5}, v_{1,7}\rangle$ contains all nodes in the upper path of τ_1. Sequence $S_{1,2} = \langle v_{1,3}, v_{1,5}, v_{1,7}\rangle$ contains the subpath starting at node $v_{1,3}$. $S_{1,3}$ starts at node $v_{1,4}$ and $S_{1,4}$ at node $v_{1,6}$. The arrows in Figure 2 represent the precedence constraints between nodes in the sequences, inherited from the DAG G_1 of τ_1.

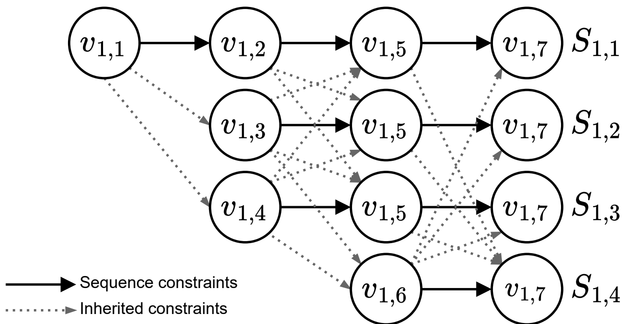


Figure 2 Example set of node sequences $\mathcal{S}_1$ obtained from the decomposition of parallel task τ_1.

As can be seen from the above example, each node of the original DAG of a given parallel task can be present in multiple sequences (e.g., $v_{1,5}$ and $v_{1,7}$ appear 3 and 4 times, respectively), which are then potentially allocated to different processors. Therefore, the nodes that belong to multiple sequences allocated to different processors are said to be replicated.

4.1.2 Runtime phase

As in the partitioned approach, the scheduling of the node sequences on each processor is managed by a dedicated uniprocessor scheduler. The runtime scheduler is designed in a way that ensures that exactly one replica of each node is executed for each job of a parallel task while enforcing the precedence constraints of the original DAG.

To do so, whenever a sequence completes the execution of a node, it checks whether all the precedence constraints of the next node in the sequence are satisfied. If they are, then the next node in the sequence is executed. If they are not, the execution of the sequence is terminated. This provides two properties. First, during system execution, a node sequence can be modeled as a sequential task without suspensions but with varying execution time executed on a single processor. Second, the combination of structural properties observed on the sequences obtained in the decomposition algorithm and of the early termination mechanism guarantees that nodes do not execute before their corresponding precedence constraints are satisfied (i.e., a sequence is ended if precedence constraints of the next node are not respected), and that exactly one replica of each node will execute for each job of a task (i.e., the replica in the last sequence reaching that node).

This means that resources are initially set aside for the execution of a specific node on multiple processors, but those resources will only be utilized in one processor for each job of the task, depending on the ongoing dynamic scheduling situation.

Running example. Assume that the sequences of Figure 2 are each assigned to a different processor of a multicore platform, so that $P(S_{1,1}) = P_1$, $P(S_{1,2}) = P_2$, $P(S_{1,3}) = P_3$, and $P(S_{1,4}) = P_4$. Note that there are four replicas of $v_{1,7}$ and three replicas of $v_{1,5}$ in this case.

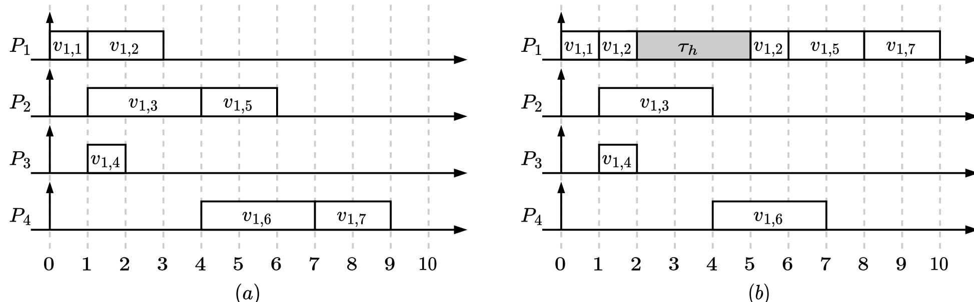


Figure 3 Example schedules of parallel task τ_1 under replication-based scheduling, in isolation (a) and with preemption by a higher-priority task τ_h (b).

Figure 3(a) provides an example schedule of a job of task τ_1 when all its subtasks execute for their WCET in isolation according to the algorithm explained above. The figure depicts the schedule of each sequence in $\mathcal{S}_1$ on the corresponding processor. In the example, the job of τ_1 is released at time 0. Therefore, the corresponding sequences arrive on the associated processor at time 0, as represented by the upward arrows. Since it corresponds to the only source node in the DAG G_1, subtask $v_{1,1}$ starts executing on processor P_1 as part of the sequence $S_{1,1}$. Once $v_{1,1}$ terminates at time 1, subtasks $v_{1,2}$, $v_{1,3}$, and $v_{1,4}$ can start their execution and sequences $S_{1,2}$ and $S_{1,3}$ are released on processors P_2 and P_3. At time 2, subtask $v_{1,4}$ terminates. At this point, the incoming precedence constraints towards the next subtask in the sequence $S_{1,3}$, i.e., $v_{1,5}$, are not yet satisfied. For this reason, the execution of the sequence $S_{1,3}$ is forcibly terminated for this job, and it will not execute its replicas of the nodes $v_{1,5}$ and $v_{1,7}$. Similarly, at time 3, the execution of subtask $v_{1,2}$ terminates, but the next subtask in the sequence $S_{1,1}$, i.e., $v_{1,5}$, cannot start executing at this time because the precedence constraint incoming from node $v_{1,3}$ is not yet satisfied. Therefore, the execution of sequence $S_{1,1}$ is also terminated early for this job. $S_{1,1}$ does not execute its replicas of the nodes $v_{1,5}$ and $v_{1,7}$. When, at time 4, subtask $v_{1,3}$ terminates its execution as part of the sequence $S_{1,2}$, all precedence constraints towards node $v_{1,5}$ are satisfied. This means that the replica of node $v_{1,5}$ belonging to the sequence $S_{1,2}$ can start its execution at time 4. At the same time, the precedence constraints towards node $v_{1,6}$ are satisfied, so that sequence $S_{1,4}$ can start its execution on processor P_4. When subtask $v_{1,5}$ terminates at time 6, node $v_{1,7}$ cannot start executing because the precedence constraint incoming from $v_{1,6}$ is not yet satisfied, therefore the corresponding sequence $S_{1,2}$ is terminated early. Finally, subtask $v_{1,7}$ starts executing on processor P_4 once subtask $v_{1,6}$ terminates at time 7, since all of its precedence constraints are satisfied at that time. The job of τ_1 finishes at time 9, when $S_{1,4}$ completes the execution of $v_{1,7}$.

Figure 3(b) provides an example schedule of a job of task τ_1 and another higher-priority task τ_h executing some workload on processor P_1 between time instants 2 and 5. This example highlights how the overall scheduling scenario on the multiprocessor system dynamically affects the selection of the replica to be executed for a job of any parallel task in the system.

In this scenario, node $v_{1,2}$ is preempted at time 2 on processor P_1, and cannot execute until time 5, when the processor becomes again available for execution of τ_1. Since node $v_{1,2}$ is the last of the predecessors of $v_{1,5}$ to terminate in this schedule, the replica of $v_{1,5}$ to be executed in this case is the one in $S_{1,1}$, instead of $S_{1,2}$ as in the previous schedule (Figure 3(a)). Similarly, the replica which is executed for the sink node $v_{1,7}$ is the one in $S_{1,1}$, again differently from the previous case.

Note that, within the schedules in Figure 3(a) and Figure 3(b), (1) each subtask of τ_1 is executed exactly once, (2) replicated nodes ($v_{1,5}$ and $v_{1,7}$) are always executed by the last sequence that reaches that node, (3) all precedence constraints between nodes of the DAG are respected, and (4) sequences may suffer release jitter and early-termination but never suffer self-suspension. These observations will be leveraged in Section 5 in order to derive a real-time analysis for replication-based scheduling.

4.2 Specification of the allocation algorithms

In the following, we specify the two algorithms that are part of the design phase of replication-based scheduling; namely, sequence expansion and sequence allocation.

4.2.1 Sequence expansion

The sequence expansion algorithm performs a decomposition of the DAG G_i of each task τ_i into sequences corresponding to subpaths in the DAG topology. The purpose of this algorithm is to generate a set of node sequences that can be executed as sequential sporadic tasks with release jitter and execution time variation.

A possible approach to perform the sequence expansion step for a given task $\tau_i \in \tau$ is described in Algorithm 1, where head($S_{i,q}$) and tail($S_{i,q}$) represent the first and the last node in a sequence $S_{i,q}$, respectively. First, the set $\mathcal{S}_i$ is initialized with a single sequence $S_{i,1}$, initially only including the source node $v_{i,S}$ of G_i (Lines 2-3). Then, the set $\mathcal{S}_i$ is extended in an incremental procedure (Lines 6-20). In this procedure, each sequence in $\mathcal{S}_i$, starting with $S_{i,1}$, is expanded by appending nodes to the sequence, following one path of the DAG G_i until a sink node of G_i is reached (Lines 7-10). When expanding a sequence $S_{i,q}$, all immediate successors of the last node in $S_{i,q}$ that are not added to $S_{i,q}$ initiate new sequences in $\mathcal{S}_i$ that are added to $\mathcal{S}_i$ (Lines 11-17). The sequences in $\mathcal{S}_i$ are expanded in the order in which they were created. A pair of indices, q and c, is used to keep track of the sequence that is currently being explored and of the last sequence added to $\mathcal{S}_i$ (Lines 4-5), and the procedure continues until all sequences in $\mathcal{S}_i$ have been expanded. Whenever a node is added to sequence, one of the successors isucc($v_{i,L}$) of the last node $v_{i,L}$ of $S_{i,q}$ is selected according to the policy implemented in the SelectSuccessor procedure and is appended to the sequence $S_{i,q}$ (Lines 8-10); then, for all the other nodes $v_{i,K}$ in isucc($v_{i,L}$), an additional sequence, initially containing $v_{i,K}$ only, is added to $\mathcal{S}_i$ if no other sequence starting with node $v_{i,K}$ exists in $\mathcal{S}_i$ (Lines 11-17).

The selection of the successor to be appended to the sequence $S_{i,q}$ that is being explored (SelectSuccessor at Line 8) can be performed according to different policies. In the following, we assume that a *static* successor selection policy is adopted, meaning that, for each node $v_{i,a} \in V_i$, the node selected to follow a replica of $v_{i,a}$ must be the same for all sequences in $\mathcal{S}_i$ in which $v_{i,a}$ appears. For instance, the immediate successor node $v_{i,r}$ with smallest index r might be selected to be added as the next element in $S_{i,q}$. Another option could be that the next selected node is the node among the candidate successors which comes first in a fixed topological ordering of the nodes of the DAG G_i. Different policies may bring to different outcomes of the sequence expansion algorithm in terms of number and structure of the resulting node sequences. In future work, one may propose a non-static successor selection policy which, for example, may consist in keeping track of the number of times each node is visited as a successor of other nodes within the sequence expansion algorithm, and then selecting the node which was visited the least number of times. Figure 2 shows the sequences resulting from applying Algorithm 1 to the DAG of Figure 1, when SelectSuccessor selects the immediate successor with smallest index.

Algorithm 1 Sequence expansion algorithm for a task τ_i.

```
1:  procedure SequenceExpansion(τ_i)
2:      S_{i,1} ← ⟨v_{i,S}⟩
3:      𝒮_i ← {S_{i,1}}
4:      q ← 1                              ▷ Index of the next sequence to be expanded
5:      c ← 1                              ▷ Index of the last sequence added to 𝒮_i
6:      while q ≤ c do
7:          while tail(S_{i,q}) ∉ sink(G_i) do
8:              v_{i,L} ← tail(S_{i,q})
9:              v_{i,A} ← SelectSuccessor(G_i, v_{i,L})
10:             S_{i,q} ← S_{i,q} ∥ v_{i,A}                ▷ Append v_{i,A} to S_{i,q}
11:             for all v_{i,K} ∈ isucc(v_{i,L}) \ v_{i,A} do
12:                 if ∀S_{i,p} ∈ 𝒮_i, v_{i,K} ≠ head(S_{i,p}) then
13:                     c ← c + 1
14:                     S_{i,c} ← ⟨v_{i,K}⟩
15:                     𝒮_i ← 𝒮_i ∪ S_{i,c}
16:                 end if
17:             end for
18:         end while
19:         q ← q + 1
20:     end while
21:     return 𝒮_i
22: end procedure
```

4.2.2 Sequence allocation

Following the sequence expansion, each sequence in the set $\mathcal{S}_i$ for each task τ_i in τ must be allocated to a specific processor, on which it is bound to execute at runtime. This procedure is akin to the partitioning problem for partitioned scheduling of sequential and parallel tasks, and can be approached with different techniques.

A possible sequence allocation scheme is described in Algorithm 2. Under this approach, tasks are allocated in order of decreasing utilization (Line 2), and each sequence is allocated in topological order of the first node of the sequence (Line 3). The choice for the allocation of each sequence is determined based on the impact on the schedulability of a partial version of the task set following a tentative allocation of the sequence on each available processor, thus determining the allocation of each task incrementally. The schedulability can be evaluated with the response-time analysis that will be presented in Section 5. In Algorithm 2, the choice for the allocation of each sequence $S_{i,q}$ of a task τ_i is determined by tentatively allocating $S_{i,q}$ to each of the processors $P_1, \ldots, P_m$, one after the other, followed, for every such allocation, by applying the schedulability analysis to all the sequences in the partial task set including the tentatively allocated sequence $S_{i,q}$ and all the other sequences that were already allocated to a processor (Lines 4-12). Note that, since the allocation of tasks does not follow a priority order, the schedulability results obtained for tasks that were already allocated cannot be reused; therefore, the schedulability of the partial task set composed of all the sequences that were already allocated must be reevaluated for each allocation attempt. In case none of the allocations produces a schedulable task set, the task set is deemed not schedulable, and the allocation returns a failure (Lines 13-15). Otherwise, the preferred allocation is selected according to a specific policy among those that result in a schedulable partial task

Algorithm 2 Sequence allocation algorithm for a task set τ.

```
1: procedure SequenceAllocation(τ)
2:     for all τ_i ∈ τ in decreasing utilization order do
3:         for all S_{i,q} ∈ 𝒮_i in topological order of the first node head(S_{i,q}) do
4:             𝒫_{i,q} ← ∅                          ▷ Set of schedulable allocations of S_{i,q}
5:             for all P_k ∈ P do
6:                 P(S_{i,q}) ← P_k                  ▷ Tentatively allocate S_{i,q} to P_k
7:                 Test the schedulability of S_{i,q} assuming it is allocated to P_k
8:                 For all S_{j,p} that have already been allocated, test the schedulability of S_{j,p}
                       assuming S_{i,q} is allocated to P_k
9:                 if P(S_{i,q}) = P_k yields a schedulable task set then
10:                    𝒫_{i,q} ← P_k
11:                end if
12:            end for
13:            if 𝒫_{i,q} = ∅ then
14:                return Failure (no valid allocation was found)
15:            end if
16:            P(S_{i,q}) ← SelectProcessor(𝒫_{i,q})   ▷ Select a schedulable allocation
17:        end for
18:    end for
19:    return Success (all sequences were allocated)
20: end procedure
```

set (SelectProcessor at Line 16). One possible selection strategy is to first determine the slack $\overline{S}_i$ of the partial version of task τ_i within the partially allocated task set, computed as $\overline{S}_i = D_i - \overline{R}_i$, where $\overline{R}_i$ is an upper bound on the WCRT of τ_i (computed with respect to the sequences that were already allocated), and then apply a Worst Fit, Best Fit, or First Fit heuristic (or a combination of them) with respect to the available slack to select the allocation. Specifically, Worst Fit and Best Fit select the allocation producing, respectively, the largest and the smallest slack $\overline{S}_i$, while First Fit simply selects the first processor that fits the sequence.

A variant of this approach is inspired by the dual allocation scheme proposed in federated scheduling. In this case, tasks are allocated in order of decreasing utilization, where the sequences of the high-utilization tasks, i.e., those tasks τ_i with utilization factor $U_i \geq 1$, are allocated as in the above approach. Instead, for sequences of low-utilization tasks, i.e., those tasks τ_i with utilization factor $U_i < 1$, an attempt is first made to allocate the full task to a processor as a single linearized sequence of τ_i, corresponding to a topological sorting of the nodes in G_i, similar to how low-utilization tasks are treated in federated scheduling. If the attempt fails, the task is allocated as in the above approach leveraging the slack.

4.3 Runtime phase and implementation pattern

The runtime phase of replication-based scheduling decides which replica of each subtask should be executed at runtime. As discussed in Section 4.1, the executed replica varies for each job of a task and depends on runtime properties like the actual execution time of predecessors or the interference suffered by subtasks. In order to realize a runtime mechanism for replication-based scheduling that is consistent with the requirements for the scheduling of parallel tasks, the following rules govern the execution of the task sequences.

- **Rule 1.** Each sequence $S_{i,q} \in \mathcal{S}_i$ of every task $\tau_i \in \tau$ is scheduled on the assigned processor $P(S_{i,q})$ according to a preemptive fixed-priority policy using the priority π_i of the corresponding task τ_i. If two or more sequences have equal priority, then the one that was released the earliest is considered as having higher priority.
- **Rule 2.** For all tasks $\tau_i \in \tau$ and each sequence $S_{i,q} \in \mathcal{S}_i$, when a job of τ_i is released, the sequence $S_{i,q}$ arrives on the corresponding processor, but is released and becomes eligible for execution only once all the precedence constraints incoming into the first node of the sequence, $\text{head}(S_{i,q})$, are satisfied.
- **Rule 3.** For all tasks $\tau_i \in \tau$ and each sequence $S_{i,q} \in \mathcal{S}_i$, nodes in the sequence $S_{i,q}$ are executed in the order in which they appear in the sequence.
- **Rule 4.** For all tasks $\tau_i \in \tau$ and each sequence $S_{i,q} \in \mathcal{S}_i$, for each pair $(v_{i,a}, v_{i,b})$ of consecutive nodes in the sequence $S_{i,q}$, whenever node $v_{i,a}$ terminates its execution as part of $S_{i,q}$ and at least one of the precedence constraints incoming into $v_{i,b}$ is not satisfied, then the execution of the sequence $S_{i,q}$ is terminated (i.e., $v_{i,b}$ and the following nodes are not executed).
- **Rule 5.** For all tasks $\tau_i \in \tau$, for each job of τ_i, no more than one replica of each node $v_{i,b} \in V_i$ can start its execution across the sequences in $\mathcal{S}_i$. In case multiple replicas of a node $v_{i,b}$ could start executing in different sequences at the same time, one of those sequences executes $v_{i,b}$ and the other sequences are terminated early, according to an arbitrary tie-breaking rule.

In the following, we describe a possible implementation pattern for the runtime rules of replication-based scheduling in a real-time operating system.

Rules 1-3 can be obtained by extending the runtime support for a preemptive fixed-priority scheduler for uniprocessors to support delaying the release of a sequence with respect to its arrival and signaling the corresponding events. In particular, Rule 1 is implemented by executing sequences according to a uniprocessor preemptive fixed-priority scheduling policy on the processor on which they are assigned. Rule 2 is implemented by delaying the release of the sequence until all the precedence constraints incoming into the first node in the sequence are satisfied. Then, Rule 3 is obtained by executing nodes in a sequence in order, one after the other.

Rules 4 and 5 require implementing an efficient inter-processor synchronization mechanism. A simple and efficient way to implement this kind of synchronization is to leverage the availability of atomic instructions (e.g., store-exclusive instructions in Arm architectures) or higher-level operating system constructs emulating their behavior. These instructions can be used by the replication-based scheduler to control the contents of a small memory area dedicated to the scheduling of a specific task, which contains the completion state of each subtask for the current job of the task, assuming that each task releases at most one job at a time. In particular, consider a task $\tau_i \in \tau$. The scheduler reserves a memory area $\mathcal{B}_i$ for τ_i that is shared among all processors to which at least one sequence of τ_i is allocated. The bits in this shared memory area can be accessed and manipulated as a bitmap using atomic instructions. The bits in $\mathcal{B}_i$ are interpreted as a vector $[B_{i,1}, B_{i,2}, \ldots, B_{i,n_i}]$ of n_i consecutive data elements, where each element $B_{i,a}$ distinguishes the completion status of the corresponding subtask $v_{i,a}$ in τ_i for the job which is currently pending among three possible states, i.e., pending but not started ($B_{i,a} = S^P$), started but not completed ($B_{i,a} = S^S$), and completed ($B_{i,a} = S^C$). In particular, when a job of the task is released, the scheduler sets the state of all nodes in $\mathcal{B}_i$ to the state S^P. Then, the code that controls the execution of each sequence $S_{i,q}$ is instrumented such that the following rules are applied.

- Whenever a node $v_{i,a}$ starts executing as part of a sequence of τ_i, the corresponding state $B_{i,a} \in \mathcal{B}_i$ is set to S^S, meaning that the node has started its execution for the current job but did not complete yet.
- Whenever a node $v_{i,a}$ completes its execution as part of a sequence of τ_i, the corresponding state $B_{i,a} \in \mathcal{B}_i$ is set to S^C, meaning that the node has completed its execution for the current job.
- Whenever a node $v_{i,a}$ with precedence constrains incoming from other sequences is the next subtask to execute in a sequence $S_{i,q}$, the shared memory area is accessed as a bitmap by the sequence $S_{i,q}$ to simultaneously check the value of all bits corresponding to the completion state of the nodes from which the precedence constraints incoming into $v_{i,a}$ originate. If at least one of those states is not S^C, meaning that the corresponding node has not yet completed in the current job of τ_i, the sequence $S_{i,q}$ is terminated early (enforcing Rule 4).
- Whenever a node $v_{i,a}$ could start executing in a sequence $S_{i,q}$, the completion status of $B_{i,a}$ is accessed and, if $B_{i,a} \neq S^P$, meaning that the corresponding node has already started executing in the current job of τ_i, the sequence $S_{i,q}$ is terminated early (enforcing Rule 5).

Running example. In the schedule in Figure 3(a), node $v_{1,5}$ cannot execute as part of sequence $S_{1,3}$ at time 2 since the completion states of nodes $v_{1,2}$ and $v_{1,3}$ are $B_{1,2} = S^S$ and $B_{1,3} = S^S$ at this time. Therefore, the corresponding sequence $S_{1,3}$ is terminated at time 2. Instead, node $v_{1,5}$ is executed as part of the sequence $S_{1,2}$ since, at time 4, all of the elements $B_{1,2}$, $B_{1,3}$, and $B_{1,4}$ corresponding to the set of nodes with precedence constraints towards $v_{1,5}$ (i.e., nodes $v_{1,2}$, $v_{1,3}$ and $v_{1,4}$) signal a completion state S^C.

The most important advantage with respect to global scheduling is that replication-based scheduling does not require support for the migration of jobs and subtasks among processors. Instead, whenever data needs to be transferred from one DAG subtask to one of its successors, it only requires that such data can be accessed by the replicas of the successor node, which can be deployed in different sequences assigned to different processors. This can be achieved by using message passing or shared memory, like in any other partitioned or global scheduler. All the node replicas must have access to the shared memory or message queue, but only one will write into it and read from it at each job execution.

4.4 Properties

In the following, we derive a set of properties of replication-based scheduling, also proving that the requirements presented in Section 2.1 for a correct execution of parallel tasks are respected with replication-based scheduling.

▶ **Lemma 1.** *For each task $\tau_i \in \tau$ and each node $v_{i,a} \in V_i$, $v_{i,a}$ is present in at least one sequence in $\mathcal{S}_i$.*

Proof. By Algorithm 1, the source node of the DAG of τ_i is the first node of sequence $S_{i,1}$ (Line 3). Now, for any node that is in a sequence $S_{i,q} \in S_i$, all its immediate successors are either in the sequence $S_{i,q}$ (Lines 8-10 of Algorithm 1) or are the first node of another sequence in S_i (Lines 11-17 of Algorithm 1). Therefore, by induction starting from the source node of τ_i, all nodes of τ_i are at least in one sequence in S_i. ◀

▶ **Lemma 2.** *Replication-based scheduling satisfies Requirement 1; i.e., for each task $\tau_i \in \tau$, each node in G_i does not start executing before all of its predecessors have completed.*

Proof. Consider a task $\tau_i \in \tau$. Consider a replica of a node $v_{i,a} \in V_i$ in a sequence $S_{i,q}$. If that replica is located at the start of the sequence $S_{i,q}$, then, by Rule 2, the release of the sequence, and thus the start of the replica, is delayed until all the precedence constraints incoming into node $v_{i,a}$ are satisfied. Instead, if that replica is not the first node of the sequence $S_{i,q}$, then, by Rule 3, nodes in $S_{i,q}$ are executed in the order in which they appear in the sequence, and, by Rule 4, the sequence is terminated if at least one of the predecessors of the replica of $v_{i,a}$ did not yet execute when the replica of $v_{i,a}$ is reached in $S_{i,q}$. Therefore, a replica of $v_{i,a}$ may only execute if all precedence constraints are satisfied. ◀

▶ **Lemma 3.** *Replication-based scheduling satisfies Requirement 2; i.e., for each task $\tau_i \in \tau$, in all jobs of τ_i, each node in G_i executes exactly once.*

Proof. By Rule 5, each node in G_i does not execute more than once in all jobs of τ_i. It then remains to prove that each node in G_i executes at least once in all jobs of τ_i. We prove it by structural induction on the topology of the DAG G_i. The base case of the structural induction corresponds to proving that the source node $v_{i,S}$ executes at least once in each job of τ_i. This holds by considering that the source node $v_{i,S}$ has no incoming precedence constraints and a single replica of $v_{i,S}$ is present as the starting node of $S_{i,1}$ in $\mathcal{S}_i$; therefore, that replica can immediately start executing once the job of the task is released (Rule 2) and can never be prevented from executing as a result of Rules 4 or 5. For the inductive step, we prove that, in a generic job of τ_i, if all the predecessors of any node $v_{i,a} \in V_i$ execute at least once, then $v_{i,a}$ executes at least once. First, by Lemma 1, for each task τ_i, each node $v_{i,a} \in V_i$ is present in at least one sequence in $\mathcal{S}_i$. If at least one replica of $v_{i,a}$ appears as the first node of a sequence $S_{i,q} \in \mathcal{S}_i$, then that replica can never be terminated as a result of Rule 4, and can in fact only be terminated if another replica of $v_{i,a}$ is selected for execution according to the tie-breaking in Rule 5, therefore $v_{i,a}$ will be executed in $\mathcal{S}_i$ once all the precedence constraints incoming into $v_{i,a}$ are satisfied (Rule 2). In the following, consider the case where $v_{i,a}$ never appears as the first node of a sequence in $\mathcal{S}_i$. Consider the last sequence that executes an immediate predecessor of $v_{i,a}$. Call that sequence $S_{i,L}$ and the executed predecessor $v_{i,L}$. $S_{i,L}$ must exist since, by the induction assumption, all immediate predecessors of $v_{i,a}$ must execute at least once for each job of τ_i. By Algorithm 1 (Lines 7-18), all sequences that include an immediate predecessor $v_{i,L}$ of $v_{i,a}$ must have a replica of $v_{i,a}$ right after $v_{i,L}$. Thus, $S_{i,L}$ either executes $v_{i,a}$ after $v_{i,L}$, which would complete the proof, or $S_{i,L}$ is terminated early after executing $v_{i,L}$. In the latter case, it means that, by Rule 4, at least one of the predecessors of $v_{i,a}$ must not have completed its execution when $S_{i,L}$ completes the execution of $v_{i,L}$. However, this contradicts the assumptions that all predecessors of $v_{i,a}$ execute, and that $S_{i,L}$ is the last sequence executing a predecessor of $v_{i,a}$. Thus, $v_{i,a}$ certainly executes as part of $S_{i,L}$. ◀

5 Schedulability analysis

Unlike other scheduling algorithms (e.g., partitioned or global fixed-priority or Earliest Deadline First scheduling), replication-based scheduling was designed from the ground up so as to simplify its schedulability analysis and avoid analytical pessimism introduced by a limited understanding of which schedule may lead to the worst-case response time of each DAG task. In fact, a task set scheduled with replication-based scheduling may simply be analyzed as a set of sequential sporadic tasks with release jitter scheduled on single core platforms.

The proposed response-time analysis for replication-based scheduling derives a WCRT upper bound $\overline{R}_{i,a}$ for each node $v_{i,a}$ in V_i. Then, the WCRT upper bound $\overline{R}_i$ of a task τ_i is given by the maximum value of $\overline{R}_{i,a}$ for any $v_{i,a}$ in V_i, or, equivalently, for any $v_{i,a} \in \text{sink}(G_i)$. The task set τ is then deemed schedulable if $\overline{R}_i \leq D_i$ holds for each task τ_i. The main observation behind the analysis is that each sequence of a parallel task τ_i in τ behaves as a sporadic task with release jitter and execution time variation.

5.1 Response-time analysis with model transformation

Consider the following properties of replication-based scheduling to support our claim that each sequence can be modeled as an independent sequential sporadic task with release jitter and execution time variation executing on a single core platform.

▶ **Property 1** (From Rule 3). *A sequence $S_{i,q}$ starts by executing its first node $\text{head}(S_{i,q})$, and all the following nodes will execute sequentially.*

▶ **Property 2** (From Rule 2). *The first node of a sequence $S_{i,q}$, $v_{i,s} = \text{head}(S_{i,q})$, arrives at the same time as the job of the task τ_i, but is released and becomes eligible for execution only once all the precedence constraints incoming into $v_{i,s}$ have been fulfilled.*

▶ **Property 3.** *A sequence $S_{i,q}$ never self-suspends as part of its execution.*

Proof. None of the Rules 1-5 allows a sequence $S_{i,q}$ to perform a self-suspension. ◀

▶ **Property 4.** *A sequence $S_{i,q}$ never migrates.*

Proof. By Rule 1, $S_{i,q}$ can only execute on the processor $P(S_{i,q})$ on which it is assigned. ◀

From the above properties, it is evident that the behavior of a sequence $S_{i,q}$ is equivalent to executing a sequential sporadic task τ'_i on a single-core platform, subject to a release jitter J'_i, where the jitter is given by the largest amount of time by which the precedence constraints of the first nodes of the sequence are fulfilled, i.e., by the maximum response time among the immediate predecessors of the first node of $S_{i,q}$, while the WCET C'_i is simply given by the sum of the WCETs of the nodes in $S_{i,q}$.

Since the above observation holds for every sequence of every task in τ, the WCRT of a sequence $S_{i,q}$ can be obtained by means of a model transformation of the set of sequences allocated to the same processor $P(S_{i,q})$ as $S_{i,q}$ into a set of sporadic tasks with release jitter.

The WCRT R'_i of a task τ'_i, and thus of a sequence $S_{i,q}$, in a set τ' of sporadic tasks with release jitter can then be computed with the response-time analysis by Audsley et al. [3]. That is, $R'_i = r'_i + J'_i$, where r'_i is the smallest positive solution of the recurrent equation[1]

$$r'_i = C'_i + \sum_{\tau'_k \in \{\tau' \setminus \tau'_i\} | \pi_k = \pi_i} C'_k + \sum_{\tau'_j \in \text{hp}(\tau'_i)} \left\lceil \frac{r'_i + J'_j}{T'_j} \right\rceil C'_j, \tag{1}$$

where $\text{hp}(\tau'_i)$ denotes the set of tasks with priority higher than that of τ'_i.

Applying the above transformation requires deriving an upper bound on the release jitter J'_i of a sequence $S_{i,q}$, which is a function of the WCRT of the predecessors of the first node of $S_{i,q}$. In fact, a sequence $S_{i,q}$ is released only when all nodes with a precedence constraint towards $S_{i,q}$ have completed their execution. Therefore, for every DAG task τ_i, the proposed analysis computes WCRT upper bounds for each node of τ_i in their topological order in G_i.

[1] Note that, since jobs with identical priorities are executed in FIFO order, at most one job of a task with identical priority to τ'_i can interfere with a job of τ'_i, hence the second term of Equation (1).

Algorithm 3 Derivation of WCRT upper bounds $\overline{R}_i$ for each task τ_i in τ.

```
1:  procedure ComputeWCRTUpperBounds(τ)
2:      for all τ_i ∈ τ in decreasing priority order do
3:          for all v_{i,a} ∈ V_i in topological order do
4:              for all S_{i,q} ∈ 𝒮_i | v_{i,a} ∈ S_{i,q} do
5:                  τ′ ← Transformation(τ, (i, a, q))
6:                  R̄_{i,a,q} ← RTA(τ′, (i, a))
7:              end for
8:              R̄_{i,a} ← max {R̄_{i,a,q} | v_{i,a} ∈ S_{i,q} ∧ S_{i,q} ∈ 𝒮_i}
9:          end for
10:         R̄_i ← max {R̄_{i,a} | v_{i,a} ∈ V_i}
11:     end for
12: end procedure
```

More specifically, as detailed in Algorithm 3, tasks in τ are analyzed in decreasing priority order, and the subtasks of each task τ_i are analyzed in topological order. A WCRT upper bound $\overline{R}_{i,a}$ is derived for each subtask $v_{i,a}$, by taking the maximum value of the WCRT bounds of all replicas of that node across all sequences of τ_i (Lines 3-10). The WCRT upper bound $\overline{R}_i$ of τ_i is then given by the response time of the node of τ_i with the largest response time (Line 10). At Line 5, the WCRT bound $\overline{R}_{i,a,q}$ for the replica of a node $v_{i,a}$ in a sequence $S_{i,q}$ is calculated by transforming τ_i and all higher-priority and equal-priority tasks into a set of equivalent sporadic tasks τ' using Algorithm 4, detailed later in this section. The WCRT upper bound of $v_{i,a}$ in a sequence $S_{i,q}$ is then obtained by applying the response-time analysis by Audsley et al. [3] presented above to the equivalent sporadic task $\tau'_{i,a} \in \tau'$ (RTA at Line 6).

The model transformation procedure (Transformation at Line 5) is detailed in Algorithm 4. The procedure constructs a set τ' of sporadic tasks with release jitter. For the analysis of a replica of node $v_{i,a}$ of τ_i in sequence $S_{i,q}$, Algorithm 4 creates one sporadic task for each node of every task with priority higher than or equal to that of τ_i that has a replica assigned to the same processor as $S_{i,q}$. The procedure is based on the following three lemmas.

▶ **Lemma 4.** *The interference generated by a sequence $S_{h,p}$ with release jitter $J_{h,p}$ and WCET $\sum_{v_{h,k} \in S_{h,p}} C_{h,k}$ in an interval of length Δ is upper bounded by the sum of the interference generated by each of its nodes $v_{h,k}$ modeled as sporadic tasks with release jitter $J_{h,p}$ and WCET $C_{h,k}$.*

Proof. Since $S_{h,p}$ behaves as a sporadic sequential task, the interference generated by $S_{h,p}$ during an interval Δ is upper bounded by $\left\lceil \frac{\Delta + J_{h,p}}{T_h} \right\rceil \times \sum_{v_{h,k} \in S_{h,p}} C_{h,k} = \sum_{v_{h,k} \in S_{h,p}} \left\lceil \frac{\Delta + J_{h,p}}{T_h} \right\rceil \times C_{h,k}$, which is equivalent to the interference generated by a set of sporadic tasks made of one task per node $v_{h,k} \in S_{h,p}$ with release jitter $J_{h,p}$ and execution time $C_{h,k}$. ◀

▶ **Lemma 5.** *Maximizing the release jitter of a node $v_{h,k}$ maximizes the interference it generates.*

Proof. Equation (1) is monotonically non-decreasing with respect to the release jitter of each task. ◀

▶ **Lemma 6.** *Let $v_{i,b}$ be a node of τ_i that is not in sequence $S_{i,q}$ and has a precedence constraint towards the first node of $S_{i,q}$. The node $v_{i,b}$ cannot interfere with $S_{i,q}$.*

Algorithm 4 Model transformation algorithm.

```
1:  procedure Transformation(τ, (i, a, q))
2:      τ' ← ∅                                                        ▷ Transformed task set
        ▷ For all sequences of higher or equal priority assigned to the same processor as S_{i,q}
3:      for all τ_h ∈ hep(τ_i) do
4:          for all S_{h,p} ∈ 𝒮_h | P(S_{h,p}) = P(S_{i,q}) do
5:              J_{h,p} ← max{0, max_{v_{h,b} ∈ ipred(head(S_{h,p}))} {R̄_{h,b}}}   ▷ Release jitter of S_{h,p}
6:          end for
                ▷ For all nodes of τ_h with a replica assigned to the same processor as S_{i,q}
7:          for all v_{h,k} ∈ V_h | ∃S_{h,p} ∈ 𝒮_h, v_{h,k} ∈ S_{h,p} ∧ P(S_{h,p}) = P(S_{i,q}) do
8:              J'_{h,k} ← max_{S_{h,p} | v_{h,k} ∈ S_{h,p} ∧ P(S_{h,p}) = P(S_{i,q})} {J_{h,p}}   ▷ Max. release jitter of v_{h,k}
9:              τ'_{h,k} ← Create a sporadic task with release jitter J'_{h,k} and WCET C_{h,k}
10:             τ' ← τ' ∪ τ'_{h,k}
11:         end for
12:     end for
                ▷ For all nodes of τ_i assigned to the same processor as S_{i,q}
13:     for all v_{i,b} ∈ {V_i \ v_{i,a}} | ∃S_{i,p} ∈ 𝒮_i, v_{i,b} ∈ S_{i,p} ∧ P(S_{i,p}) = P(S_{i,q}) do
14:         if v_{i,b} is independent from head(S_{i,q}) in G_i then
15:             τ'_{i,b} ← Create a sporadic task with WCET C_{i,b} and release jitter J'_{i,b} = 0
16:             τ' ← τ' ∪ τ'_{i,b}
17:         end if
18:     end for
19:     S*_{i,q} ← The sequence obtained by removing all nodes after v_{i,a} in S_{i,q}
20:     J'_{i,q} ← max{0, max_{v_{i,b} ∈ ipred(head(S_{i,q}))} {R̄_{i,b}}}   ▷ Max. release jitter of S_{i,q}
21:     τ'_{i,a} ← Create a sporadic task with WCET Σ_{v_{i,j} ∈ S*_{i,q}} C_{i,j} and release jitter J'_{i,q}
22:     τ' ← τ' ∪ τ'_{i,a}
23:     return τ'
24: end procedure
```

Proof. Let head$(S_{i,q})$ be the first node of $S_{i,q}$. The node $v_{i,b}$ is either a predecessor or a successor of head$(S_{i,q})$. In case $v_{i,b}$ is a predecessor of head$(S_{i,q})$, then $v_{i,b}$ must be completed when $S_{i,q}$ is released. Therefore, $v_{i,b}$ does not interfere with $S_{i,q}$. In case $v_{i,b}$ is a successor of head$(S_{i,q})$, then a job of $v_{i,b}$ can only be released after $S_{i,q}$. Since jobs with equal priority execute in FIFO order, $v_{i,b}$ executes after $S_{i,q}$, and thus does not interfere with $S_{i,q}$. ◀

Following the result in Lemma 4, Algorithm 4 creates one sporadic task $\tau'_{h,k}$ per node $v_{h,k}$ of each sequence of every higher-priority or equal-priority task different from τ_i (i.e., of every task in the set hep(τ_i)) assigned to the same core as the sequence $S_{i,q}$ under analysis (Lines 3-12), in order to upper bound the interference generated by those sequences. According to Lemma 3, for each job released by a task, each of its nodes executes at most once, irrespective of its number of replicas. Therefore, Algorithm 4 only generates one sporadic task per node instead of one sporadic task per replica. The WCET of the generated task $\tau'_{h,k}$ is then equal to the WCET of the node $v_{h,k}$, and its release jitter is the maximum release jitter of all the sequences in which $v_{h,k}$ appears (Line 8), so as to maximize the interference it generates (see Lemma 5). The minimum inter-arrival time and the priority of the generated task $\tau'_{h,k}$ are inherited from the corresponding task τ_h.

After generating equivalent sporadic tasks for all the tasks in hep(τ_i), Algorithm 4 generates sporadic tasks to model the self-interference of nodes of τ_i on the sequence $S_{i,q}$ under analysis (Lines 13-18). One such task is generated for each node of τ_i, except $v_{i,a}$

itself, that is independent from the first node in $S_{i,q}$ and has a replica assigned to $P(S_{i,q})$ (in accordance with Lemma 6). Note that, according to Equation (1), since the sporadic tasks modeling the self-interference of nodes of τ_i have the same priority as $S_{i,q}$, their release jitter does not influence the WCRT of the sequence under analysis. Therefore, Algorithm 4 arbitrarily sets their release jitter to 0.

Finally, since we aim at computing the WCRT of node $v_{i,a}$ in sequence $S_{i,q}$, Algorithm 4 models the partial sequence $S^{\star}_{i,q} \subseteq S_{i,q}$ ending at $v_{i,a}$ as a sporadic task $\tau'_{i,a}$, with WCET equal to the sum of the execution time of its nodes and release jitter equal to the maximum WCRT upper bound of the predecessors of the first node of $S_{i,q}$ (Lines 19-22).

5.2 Analysis improvements

Although the analysis presented in Section 5.1 is an efficient approach to test the schedulability of a set of parallel tasks executing under replication-based scheduling, the analysis might yield pessimistic WCRT upper bounds in some cases.

In order to identify a potential source of such analytical pessimism, consider a replica of the node under analysis $v_{i,a}$ in sequence $S_{i,q}$ and a replica of another node $v_{i,b}$ in $S_{i,p}$ that triggers the release of $S_{i,q}$ (i.e., it is an immediate predecessor of head($S_{i,q}$) that causes the release jitter on $S_{i,q}$). According to the analysis in Section 5.1, the WCRT of $v_{i,a}$ in $S_{i,q}$ is upper bounded by the WCRT upper bound of $v_{i,b}$ added to the solution to Equation (1) for $v_{i,a}$. Assume that there is a node v_h of a higher-priority task with replicas assigned to the processors where $S_{i,q}$ and $S_{i,p}$ execute. Then, v_h interferes with both $v_{i,a}$ and $v_{i,b}$. Since the analysis in Section 5.1 analyses the WCRT of $v_{i,a}$ and $v_{i,b}$ independently from each other, it may account for the same jobs of v_h as interfering with both $v_{i,a}$ and $v_{i,b}$, thus overestimating the overall interference those jobs may generate.

The following lemma provides a lower bound on the redundant interference caused by the higher-priority node v_h in the computation of $\overline{R}_{i,a,q}$, with reference to the replicas of $v_{i,a}$ in $S_{i,q}$ and of $v_{i,b}$ in $S_{i,p}$.

▶ **Lemma 7.** *Let $r_{i,a,q}$ and $r_{i,b,p}$ represent the solutions to Equation (1) for, respectively, $v_{i,a}$ in $S_{i,q}$ and $v_{i,b}$ in $S_{i,p}$. Assume that the release jitter of $S_{i,q}$ is equal to the WCRT upper bound $\overline{R}_{i,b,p}$ of the replica of $v_{i,b}$ in $S_{i,p}$. The redundant interference caused by v_h on both the replica of v_a in $S_{i,q}$ and the replica of v_b in $S_{i,p}$, i.e., the amount of interference caused by v_h included in the computation of both $r_{i,a,q}$ and $r_{i,b,p}$, is lower bounded by*

$$\left(\left\lceil \frac{r_{i,b,p} + J'_h}{T_h} \right\rceil + \left\lceil \frac{r_{i,a,q} + J'_h}{T_h} \right\rceil - \left\lceil \frac{r_{i,b,p} + r_{i,a,q} + J'_h}{T_h} \right\rceil\right) \cdot C'_h. \tag{2}$$

Proof. Let J'_h and T_h be the release jitter and minimum inter-arrival time of node v_h. The number of jobs of v_h considered as causing direct interference on the replica of $v_{i,a}$ in $S_{i,q}$ as part of $r_{i,a,q}$ is given by $\left\lceil \frac{r_{i,a,q}+J'_h}{T_h} \right\rceil$ (from Equation (1)). Similarly, the number of jobs of v_h considered as causing interference on the replica of $v_{i,b}$ in $S_{i,p}$ is given by $\left\lceil \frac{r_{i,b,p}+J'_h}{T_h} \right\rceil$. Since the analysis in Section 5.1 adds the WCRT upper bound of $v_{i,b}$ (as part of the release jitter of $v_{i,a}$) to $r_{i,a,q}$ to calculate the WCRT upper bound of $v_{i,a}$, it considers that, in total, $\left\lceil \frac{r_{i,a,q}+J'_h}{T_h} \right\rceil + \left\lceil \frac{r_{i,b,p}+J'_h}{T_h} \right\rceil$ jobs of v_h participate to the WCRT upper bound of $v_{i,a}$. However, since $v_{i,b}$ in $S_{i,p}$ triggers the release of $S_{i,q}$, the time between the release of $v_{i,b}$ in $S_{i,p}$ and the completion of $v_{i,a}$ in $S_{i,q}$ is upper bounded by $r_{i,b,p} + r_{i,a,q}$. Therefore, the number of jobs of v_h released between the release time of $v_{i,b}$ and the completion of $v_{i,a}$ cannot be larger than $\left\lceil \frac{r_{i,b,p}+r_{i,a,q}+J'_h}{T_h} \right\rceil$. Thus, the analysis in Section 5.1 considers at least

Algorithm 5 Analysis improvements for the derivation of the WCRT of a node $v_{i,a}$ in sequence $S_{i,q}$ of a task $\tau_i \in \tau$.

1: **procedure** ReduceJitter($\tau, (i, a, q)$)
2: $\quad r_{i,a,q} \leftarrow$ The solution to Equation (1) for $v_{i,a}$ in $S_{i,q}$
3: $\quad J_{i,a,q} \leftarrow 0$ $\qquad \triangleright$ Jitter of $v_{i,a}$ in $S_{i,q}$
4: $\quad v_{i,s} \leftarrow \text{head}(S_{i,q})$
5: $\quad$ **for all** $v_{i,b} \in \text{ipred}(v_{i,s})$ **do**
6: $\quad\quad$ **for all** $S_{i,p} \in \mathcal{S}_i \mid v_{i,b} \in S_{i,p}$ **do**
7: $\quad\quad\quad r_{i,b,p} \leftarrow$ The solution to Equation (1) for $v_{i,b}$ in $S_{i,p}$
8: $\quad\quad\quad \mathcal{V}_h \leftarrow$ The set of all nodes with higher priority than τ_i with replicas assigned to both $P(S_{i,q})$ and $P(S_{i,p})$
9: $\quad\quad\quad$ **for all** $v_h \in \mathcal{V}_h$ **do**
10: $\quad\quad\quad\quad J'_h \leftarrow$ The maximum release jitter of v_h as an interfering sequential task
11: $\quad\quad\quad\quad C'_h \leftarrow$ The WCET of node v_h
12: $\quad\quad\quad\quad I_h^{\text{redundant}} \leftarrow \left(\left\lceil \frac{r_{i,b,p}+J'_h}{T_h} \right\rceil + \left\lceil \frac{r_{i,a,q}+J'_h}{T_h} \right\rceil - \left\lceil \frac{r_{i,b,p}+r_{i,a,q}+J'_h}{T_h} \right\rceil \right) \cdot C'_h$
13: $\quad\quad\quad$ **end for**
14: $\quad\quad\quad J^\star_{i,a,q} \leftarrow \overline{R}_{i,b,p} - \sum_{v_h \in \mathcal{V}_h} I_h^{\text{redundant}}$
15: $\quad\quad\quad J_{i,a,q} \leftarrow \max\{J_{i,a,q}, J^\star_{i,a,q}\}$
16: $\quad\quad$ **end for**
17: $\quad$ **end for**
18: $\quad \overline{R}_{i,a,q} \leftarrow J_{i,a,q} + r_{i,a,q}$
19: $\quad$ **return** $\overline{R}_{i,a,q}$
20: **end procedure**

$\left\lceil \frac{r_{i,b,p}+J'_h}{T_h} \right\rceil + \left\lceil \frac{r_{i,a,q}+J'_h}{T_h} \right\rceil - \left\lceil \frac{r_{i,b,p}+r_{i,a,q}+J'_h}{T_h} \right\rceil$ too many jobs of v_h as contributing to the WCRT of $v_{i,a}$ in $S_{i,q}$. Every such job of v_h has a WCET of C'_h. Therefore, Equation (2) is a lower bound on the redundant interference caused by v_h on both the replica of $v_{i,a}$ in $S_{i,q}$ and the replica of $v_{i,b}$ in $S_{i,p}$. ◀

We use Lemma 7 to improve the analysis in Section 5.1. We introduce an additional step in Algorithm 3 right after the WCRT upper bound $\overline{R}_{i,a,q}$ for a node $v_{i,a}$ within a sequence $S_{i,q}$ is obtained (i.e., right after Line 6). The additional analysis step computes a reduced value for the release jitter of the sporadic task modeling the sequence $S_{i,q}$ in the analysis of $v_{i,a}$ by discounting redundant interference caused by higher-priority nodes that interfere both with immediate predecessors of head($S_{i,q}$), whose WCRT upper bounds determine the release jitter of $S_{i,q}$, and with $S_{i,q}$ itself.

Algorithm 5 details how the WCRT upper bound $\overline{R}_{i,a,q}$ is updated for the node $v_{i,a}$ within the sequence $S_{i,q}$, by computing a reduced release jitter $J_{i,a,q}$ for $S_{i,q}$. In the algorithm, the jitter $J_{i,a,q}$ is initially set to 0 (Line 3). Then, at Lines 4-17, the procedure examines each immediate predecessor of head($S_{i,q}$) to determine which predecessors may generate the largest release jitter $J_{i,a,q}$ for $S_{i,q}$. For every predecessor $v_{i,b}$ of head($S_{i,q}$), and for every sequence $S_{i,p}$ in $\mathcal{S}_i$ containing a replica of $v_{i,b}$, a candidate value $J^\star_{i,a,q}$ for the release jitter is obtained by subtracting redundant interference from the WCRT upper bound $\overline{R}_{i,b,p}$ (Lines 5-16). Specifically, the total redundant interference on $v_{i,a}$ in $S_{i,q}$ and $v_{i,b}$ in $S_{i,p}$ to be subtracted from $\overline{R}_{i,b,p}$ is derived by first identifying all nodes with higher priority than τ_i that have at least one replica assigned to processor $P(S_{i,q})$ and one replica assigned to processor $P(S_{i,p})$, meaning that they contribute interference in the computation of both $r_{i,a,q}$ and $r_{i,b,p}$. For every such higher-priority node v_h, we use Equation (2) to compute

the redundant interference and discount it from $\overline{R}_{i,b,p}$ to obtain $J^{\star}_{i,a,q}$. The value of $J_{i,a,q}$ is then set to the maximum between $J^{\star}_{i,a,q}$ and the current value of $J_{i,a,q}$. Thus, the final value of $J_{i,a,q}$ is given by the maximum release jitter candidate among those computed for all replicas of every immediate predecessor of $\text{head}(S_{i,q})$. As a result, the replicas of the immediate predecessor of $\text{head}(S_{i,q})$ that produced a candidate release jitter value equal to the final value of $J_{i,a,q}$ satisfy the assumption in Lemma 7. Finally, given the resulting value of $J_{i,a,q}$, the WCRT upper bound of $v_{i,a}$ within the sequence $S_{i,q}$ is recomputed as $\overline{R}_{i,a,q} = J_{i,a,q} + r_{i,a,q}$ (Line 18).

6 Experimental results

This section presents the results of an experimental evaluation of the proposed replication-based scheduling approach, including a comparison with state-of-the-art variants of federated scheduling [23] and partitioned scheduling [2].

6.1 Experimental setup

The experimental campaign is based on the analysis of randomly generated task sets. The task set generation procedure works as follows. The number of tasks n composing each task set τ is a generation parameter which is fixed for each experiment. For each parallel task τ_i, the topology of the DAG G_i is generated according to the technique by Melani et al. [24]. This approach generates a series-parallel graph with multiple levels of nested parallel branches in a recursive approach which starts from an initial graph composed of two nodes and then recursively expands non-terminal nodes to either terminal nodes or additional parallel subgraphs, until a maximum recursion depth is reached. The maximum recursion depth is modeled as a generation parameter n_{rec}, and another generation parameter p_{par} is used to represent the probability with which a non-terminal node is expanded to a parallel subgraph within the recursion. The level of parallelism of the parallel subgraph is controlled with an additional parameter, n_{par}. In particular, the number of branches to which a node is expanded is selected from the discrete uniform distribution $[2, n_{par}]$.

Given the value of the system utilization U, the UUniFast algorithm by Bini and Buttazzo [10] was used to generate the utilization U_i for each task $\tau_i \in \tau$. In particular, UUniFast is used to uniformly select n real values $\hat{U}_i \in [0, 1]$ such that $\sum_{i=1}^{n} \hat{U}_i = 1$; then, the utilization U_i of each task τ_i is set to $U_i = U \cdot \hat{U}_i$. Once the DAG topology G_i of a task τ_i is generated, the minimum inter-arrival time T_i of τ_i is selected from a discrete uniform distribution with range $[T_{min}, T_{max}]$, where T_{min} and T_{max} are generation parameters. The deadline of each task τ_i is set to $D_i = T_i$ (implicit deadlines). The cumulative WCET C_i of τ_i is set to $C_i = U_i \cdot T_i$; then, the WCET $C_{i,a}$ of each node $v_{i,a} \in V_i$ is generated using the UUniFast algorithm by distributing the WCET C_i among the nodes of G_i in such a way that $\sum_{v_{i,a} \in V_i} C_{i,a} = C_i$. In particular, UUniFast is used to uniformly select n_i real values $\hat{C}_{i,a} \in [0, 1]$ such that $\sum_{v_{i,a} \in V_i} \hat{C}_{i,a} = 1$; then, the WCET $C_{i,a}$ of each node $v_{i,a} \in V_i$ is set to $C_{i,a} = C_i \cdot \hat{C}_{i,a}$. Finally, the priority level π_i of each task τ_i is assigned according to the Rate Monotonic algorithm, which assigns higher priority levels to tasks with smaller minimum inter-arrival time T_i.

In order to limit the amount of non-feasible task sets generated for the experiments, the generation procedure for each task τ_i is repeated (up to 5000 times) in case either **(i)** $C_{i,a} > D_i$ holds for some node $v_{i,a} \in V_i$; or **(ii)** $\sum_{v_{i,a} \in V(\lambda)} C_{i,a} > D_i$ holds for some path $\lambda \in \text{path}(G_i)$.

In each experiment, the number of processors m in the platform was fixed to a specific value, and the system utilization U was varied between 0 and m in increments of 0.5. For each value of U, 100 task sets were generated and analyzed using the replication-based, federated, and partitioned scheduling approaches. The performance metric considered in the experiments is the schedulability ratio with respect to the system utilization U, computed as the ratio between the number of task sets deemed schedulable by a given analysis approach and the total number of task sets evaluated for the utilization point U.

The following scheduling approaches and respective analyses were tested: **(RBS-WBF)** replication-based scheduling, testing all the available heuristics (Worst Fit, Best Fit, First Fit) and applying the analysis in Section 5.1; **(RBS-WBF-I)** like RBS-WBF, but leveraging the improved analysis in Section 5.2; **(RBS-DUAL)** replication-based scheduling using the variant allocation approach which treats high-utilization and low-utilization tasks differently, testing the Worst Fit heuristic and applying the analysis in Section 5.1; **(RBS-DUAL-I)** like RBS-DUAL, but leveraging the improved analysis in Section 5.2; **(RBS-OR)** logic OR combination of RBS-WBF and RBS-DUAL, which deems a task set schedulable if it is deemed schedulable by at least one of RBS-WBF and RBS-DUAL; **(RBS-OR-I)** logic OR combination of RBS-WBF-I and RBS-DUAL-I; **(FED-WBF)** federated scheduling [23], allocating low-utilization tasks by decreasing utilization order and testing all the standard Worst Fit, Best Fit, and First Fit heuristics; and **(PART-EDD)** partitioned scheduling, analyzed using an approach leveraging mixed-integer linear programming following a transformation to the event-driven delay-induced task model, and with allocation determined according to the best performing variant of the pseudo-federated approach, which treats high-utilization and low-utilization tasks similarly to federated scheduling and distributes nodes of low-utilization tasks on underutilized dedicated processors of high-utilization tasks [2].

6.2 Experimental results

Figure 4 reports the results of the experiments. For all system configurations, the values of n_{rec}, p_{par}, T_{min}, and T_{max} were set to $n_{rec} = 2$, $p_{par} = 0.8$, $T_{min} = 100$, and $T_{max} = 1000$, while the other parameters (m, n, n_{par}) were varied among the experiments, and their value for each experiment is reported above the corresponding graph. The PAR-FEAS curve represents the ratio of task sets which satisfy both feasibility conditions in the generation, i.e., $C_{i,a} \leq D_i$ for all nodes $v_{i,a} \in V_i$, and $\sum_{v_{i,a} \in V(\lambda)} C_{i,a} \leq D_i$ for all paths $\lambda \in \text{path}(G_i)$. This curve represents an upper bound on the attainable performance of the evaluated scheduling and analysis approaches.

The results for $n_{par} = 3$ (Figures 4(a-c)) share a common overall trend, with replication-based scheduling outperforming both federated and partitioned scheduling by a significant margin. For what concerns replication-based scheduling, the most significant performance loss occurs at utilization values U that are above 50% of the available system utilization m across all processors, with the overall performance decline starting at around 37.5% of m. Partitioned scheduling follows with an intermediate level of performance, while federated scheduling exhibits the worst performance among the evaluated approaches. The same general pattern is observed in the experiments in which a larger number of nodes is generated for each task, i.e., when $n_{par} = 5$ (Figures 4(d-f)). In this case, the drop-off for replication-based scheduling with respect to U occurs again at around 37.5% of m, but with a sharper performance loss after that point. Partitioned scheduling suffers a similar loss in performance, whereas federated scheduling exhibits robust performance with respect to the previous case. Across all experiments, the two tested allocation approaches for replication-based scheduling, RBS-WBF and RBS-DUAL, show comparable performance, with the combined approach

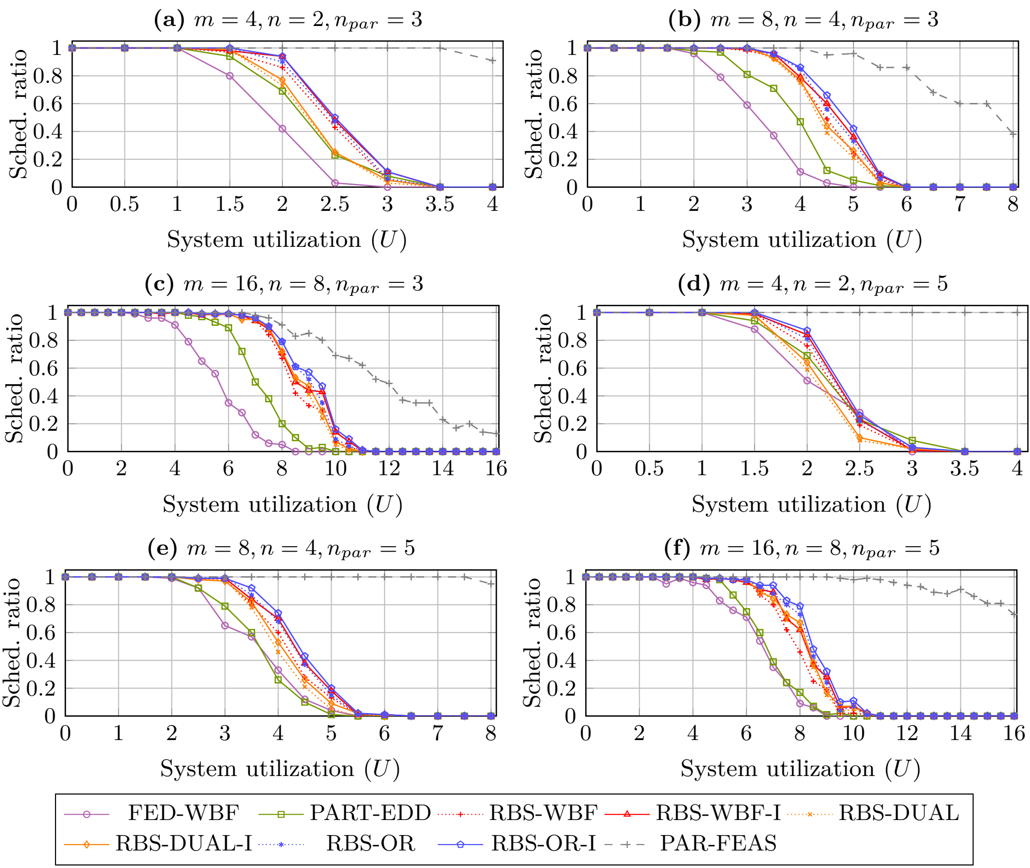


Figure 4 Schedulability ratio with respect to the system utilization U obtained for different system configurations.

RBS-OR granting an additional edge in performance, meaning that neither of the methods dominates the other. Finally, in all the tested scenarios, the RBS-WBF-I, RBS-DUAL-I, and RBS-OR-I approaches utilizing the improved analysis in Section 5.2 provided slightly improved performance with respect to the corresponding RBS-WBF, RBS-DUAL, and RBS-OR approaches adopting the analysis in Section 5.1.

Overall, the experiments show that replication-based scheduling can outperform both partitioned and federated scheduling by a large margin across several system configurations.

7 Conclusions and future work

This paper presented replication-based scheduling, a specialized scheduling approach for parallel real-time tasks executing on a multiprocessor platform which leverages the internal topology of the DAG of each task to provide enhanced schedulability performance with limited expected runtime overhead and analysis complexity. In addition to the overall scheduling paradigm, design-time allocation strategies were discussed, and a response-time analysis for the case of fixed-priority preemptive scheduling was provided. Experimental results showed that replication-based scheduling significantly outperforms state-of-the-art variations of both federated and partitioned scheduling. Future work includes implementing replication-based

scheduling in a real-time operating system, investigating further improvements to the provided analysis, and exploring variants of replication-based scheduling supporting Earliest Deadline First scheduling and non-preemptive execution of nodes. Finally, given the flexibility of the proposed scheduling framework, future work should also evaluate further variations to the design-time allocation algorithms.

References

1. Ishfaq Ahmad and Yu-Kwong Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 9(9):872–892, 1998.
2. Federico Aromolo, Alessandro Biondi, Geoffrey Nelissen, and Giorgio Buttazzo. Event-driven delay-induced tasks: Model, analysis, and applications. In *Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2021)*, pages 53–65. IEEE, 2021.
3. Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering*, 8(5):284–292, 1993.
4. Philip Axer, Sophie Quinton, Moritz Neukirchner, Rolf Ernst, Björn Döbel, and Hermann Härtig. Response-time analysis of parallel fork-join workloads with real-time constraints. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS 2013)*, pages 215–224. IEEE, 2013.
5. Sanjoy Baruah. Improved multiprocessor global schedulability analysis of sporadic DAG task systems. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS 2014)*, pages 97–105. IEEE, 2014.
6. Sanjoy Baruah. The federated scheduling of constrained-deadline sporadic DAG task systems. In *Proceedings of the 18th IEEE Design, Automation and Test in Europe Conference and Exhibition (DATE 2015)*, pages 1323–1328. IEEE, 2015.
7. Sanjoy Baruah. Federated scheduling of sporadic DAG task systems. In *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2015)*, pages 179–186. IEEE, 2015.
8. Andrea Bastoni, Björn Brandenburg, and James Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2010)*, pages 33–44, 2010.
9. Andrea Bastoni, Björn B. Brandenburg, and James H. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers. In *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS 2010)*, pages 14–24. IEEE, 2010.
10. Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.
11. Alessandro Biondi and Youcheng Sun. On the ineffectiveness of 1/m-based interference bounds in the analysis of global EDF and FIFO scheduling. *Real-Time Systems*, 54:515–536, 2018.
12. Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Sebastian Stiller, and Andreas Wiese. Feasibility analysis in the sporadic DAG task model. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS 2013)*, pages 225–233. IEEE, 2013.
13. Daniel Casini, Alessandro Biondi, Geoffrey Nelissen, and Giorgio Buttazzo. Partitioned fixed-priority scheduling of parallel tasks without preemptions. In *Proceedings of the 39th IEEE Real-Time Systems Symposium (RTSS 2018)*, pages 421–433. IEEE, 2018.
14. Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil Audsley, Raj Rajkumar, and Georg von der Brüggen. Many suspensions, many problems: A review of self-suspending tasks in real-time systems. *Real-Time Systems*, 55(1):144–207, 2019.

15 Son Dinh, Christopher Gill, and Kunal Agrawal. Efficient deterministic federated scheduling for parallel real-time tasks. In *Proceedings of the 26th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2020)*, pages 1–10. IEEE, 2020.

16 José Fonseca. *Multiprocessor Scheduling and Mapping Techniques for Real-Time Parallel Applications*. PhD thesis, Faculdade de Engenharia da Universidade do Porto, 2019.

17 José Fonseca, Geoffrey Nelissen, and Vincent Nélis. Improved response time analysis of sporadic DAG tasks for global FP scheduling. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems (RTNS 2017)*, pages 28–37. ACM, 2017.

18 José Fonseca, Geoffrey Nelissen, and Vincent Nélis. Schedulability analysis of DAG tasks with arbitrary deadlines under global fixed-priority scheduling. *Real-Time Systems*, 55(2):387–432, 2019.

19 José Fonseca, Geoffrey Nelissen, Vincent Nélis, and Luís Miguel Pinho. Response time analysis of sporadic DAG tasks under partitioned scheduling. In *Proceedings of the 11th IEEE Symposium on Industrial Embedded Systems (SIES 2016)*, pages 1–10. IEEE, 2016.

20 Xu Jiang, Nan Guan, Haochun Liang, Yue Tang, Lei Qiao, and Wang Yi. Virtually-federated scheduling of parallel real-time tasks. In *Proceedings of the 42nd IEEE Real-Time Systems Symposium (RTSS 2021)*, pages 482–494. IEEE, 2021.

21 Xu Jiang, Nan Guan, Xiang Long, and Wang Yi. Semi-federated scheduling of parallel real-time tasks on multiprocessors. In *Proceedings of the 38th IEEE Real-Time Systems Symposium (RTSS 2017)*, pages 80–91. IEEE, 2017.

22 Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS 2010)*, pages 259–268. IEEE, 2010.

23 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 *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS 2014)*, pages 85–96. IEEE, 2014.

24 Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio C. Buttazzo. Response-time analysis of conditional DAG tasks in multiprocessor systems. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS 2015)*, pages 211–221. IEEE, 2015.

25 Alessandra Melani, Maria A Serrano, Marko Bertogna, Isabella Cerutti, Eduardo Quinones, and Giorgio Buttazzo. A static scheduling approach to enable safety-critical OpenMP applications. In *Procedings of the 22nd Asia and South Pacific Design Automation Conference (ASP-DAC 2017)*, pages 659–665. IEEE, 2017.

26 Amir Nahir, Ariel Orda, and Danny Raz. Replication-based load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 27(2):494–507, 2015.

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

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

29 Niklas Ueter, Georg Von Der Brüggen, Jian-Jia Chen, Jing Li, and Kunal Agrawal. Reservation-based federated scheduling for parallel real-time tasks. In *Proceedings of the 39th IEEE Real-Time Systems Symposium (RTSS 2018)*, pages 482–494. IEEE, 2018.

30 Roberto Vargas, Eduardo Quinones, and Andrea Marongiu. OpenMP and timing predictability: A possible union? In *Proceedings of the 18th IEEE Design, Automation and Test in Europe Conference and Exhibition (DATE 2015)*, pages 617–620. IEEE, 2015.

From FMTV to WATERS: Lessons Learned from the First Verification Challenge at ECRTS

Sebastian Altmeyer
Universität Augsburg, Germany

Étienne André[1]
Université Sorbonne Paris Nord, LIPN, CNRS UMR 7030, F-93430 Villetaneuse, France

Silvano Dal Zilio
Univ. de Toulouse, INSA, LAAS, F-31400 Toulouse, France

Loïc Fejoz
RealTime-at-Work (RTaW), 615, Rue du Jardin Botanique, F-54600 Villers-lès-Nancy, France

Michael González Harbour
Universidad de Cantabria, Santander, Spain

Susanne Graf
Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, F-38000 Grenoble, France

J. Javier Gutiérrez
Universidad de Cantabria, Santander, Spain

Rafik Henia[1]
Thales Research & Technology, F-91767 Palaiseau, France

Didier Le Botlan
Univ. de Toulouse, INSA, LAAS, F-31400 Toulouse, France

Giuseppe Lipari
Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Julio Medina
Universidad de Cantabria, Santander, Spain

Nicolas Navet
University of Luxembourg, Luxembourg

Sophie Quinton[1]
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, F-38000 Grenoble, France

Juan M. Rivas
Universidad de Cantabria, Santander, Spain

Youcheng Sun
The University of Manchester, UK

Abstract

We present here the main features and lessons learned from the first edition of what has now become the ECRTS industrial challenge, together with the final description of the challenge and a comparative overview of the proposed solutions. This verification challenge, proposed by Thales, was first discussed in 2014 as part of a dedicated workshop (FMTV, a satellite event of the FM 2014 conference), and solutions were discussed for the first time at the WATERS 2015 workshop. The use case for the verification challenge is an aerial video tracking system. A specificity of this system lies in the fact that periods are constant but known with a limited precision only. The first part of the challenge focuses on the video frame processing system. It consists in computing maximum values of the end-to-end latency of the frames sent by the camera to the display, for two different buffer sizes, and then the minimum duration between two consecutive frame losses. The second challenge is about computing end-to-end latencies on the tracking and camera control for two different values of jitter. Solutions based on five different tools – Fiacre/Tina, CPAL (simulation and analysis), IMITATOR, UPPAAL and MAST – were submitted for discussion at WATERS 2015. While none of these solutions provided a full answer to the challenge, a combination of several of them did allow to draw some conclusions.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Embedded systems; General and reference → Verification; Software and its engineering → Software verification and validation

Keywords and phrases Verification challenge, industrial use case, end-to-end latency

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.19

[1] Corresponding author.

© Sebastian Altmeyer, Étienne André, Silvano Dal Zilio, Loïc Fejoz, Michael González Harbour, Susanne Graf, J. Javier Gutiérrez, Rafik Henia, Didier Le Botlan, Giuseppe Lipari, Julio Medina, Nicolas Navet, Sophie Quinton, Juan M. Rivas, and Youcheng Sun;
licensed under Creative Commons License CC-BY 4.0

Category Invited Paper

Supplementary Material *Software (ECRTS 2023 Artifact Evaluation approved artifact)*: https://doi.org/10.4230/DARTS.9.1.4

Funding This work was partially supported by MCIN/ AEI /10.13039/501100011033/ FEDER "Una manera de hacer Europa" under grant PID2021-124502OB-C42 (PRESECREL) and by the AIDOaRt project, an ECSEL Joint Under-taking (JU) under grant agreement No. 101007350.
Étienne André: Partially supported by the ANR-NRF French-Singaporean research program ProMiS (ANR-19-CE25-0015 / 2019 ANR NRF 0092) and ANR BisoUS (ANR-22-CE48-0012).

1 Introduction

Many model-based techniques and tools have been developped for the timing estimation and verification of critical real-time systems. One can classify these approaches into three categories: simulation, model checking and response-time analysis. The many existing tools apply to different but sometimes similar models and may provide different types of guarantees. This makes it difficult for researchers and practitioners to understand the advantages and drawbacks of one approach compared to another, or even simply to figure out which tools can perform a given type of analysis on a given system.

The WATERS industrial challenge was introduced in 2015 to address this issue by providing an opportunity for researchers to try their favorite tool on a practical problem. For Thales, who proposed the first challenge, this was an opportunity to better understand how various analysis methods and tools proposed by the research community can be applied to the large variety of real-time requirements of the Thales products (ranging from hard to soft real-time requirements). For the research community, the main motivation for participating to the challenge was to address concrete timing analysis problems issued from real industrial case studies. Such a challenge thus promotes discussions and closer interactions between research and industry.

A preliminary version of the challenge was presented and discussed at the FMTV 2014 workshop (FMTV standing for "Formal Methods for Timing Verification"), a satellite event of FM 2014 (the 19th International Symposium on Formal Methods) [11]. An improved version was then proposed for WATERS 2015 (the 6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems) [22], a satellite event of ECRTS 2015. Five solutions, representing all model-based timing analysis techniques (model checking, response-time analysis and simulation), were submitted that year. Interestingly, no tool was able to solve all subchallenges, which shows that there is currently no unique solution that fits every timing verification problem. The combined use of several tools, however, led to better results than those provided by each individual tool, while increasing the confidence in the produced results.

The first WATERS industrial challenge provided Thales, the solution providers as well as all WATERS 2015 attendees with a better understanding of the various techniques and tools, and in particular of their strengths and weaknesses with respect to several aspects, such as: ease of modeling, level of automation of the verification process, verification time, reliability of the results, etc. Based on the feedback given by the solution providers with respect to the description of the challenge, Thales were also able to provide a consolidated version of the challenge including a corresponding model in Papyrus [13], for which solutions could subsequently be submitted (see e.g., [21]).

Although this challenge is quite ancient now, we believe that the lessons learned from this experience and the research perspectives that it opened are still relevant today. The purpose of this paper is therefore to share that knowledge and to make available to the community

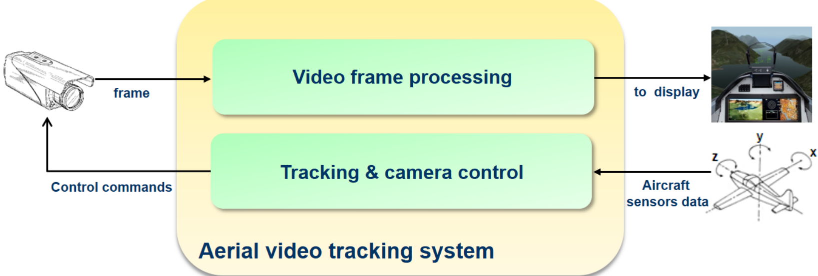


Figure 1 Subsystems of the aerial video tracking system.

the final description of the challenge and a comparative overview of the proposed solutions. The authoritative model of the challenge as well as the code for several of the solutions is available as additional material submitted together with this paper.

2 Presentation of the verification challenge

The use-case provided by Thales consists of an aerial video system to detect and track moving objects, e.g., vehicles on a roadway. Aerial video tracking systems are mission critical real-time systems since they embed intelligence, surveillance, reconnaissance, tactical and security applications characterized by strict constraints on timing. The main system tasks consist in:

- displaying high quality video images to the user;
- following the tracked object even when it is temporarily hidden from view (e.g., the vehicle proceeds in and out of a tree obstructed area) through motion prediction;
- detecting patches of the image that may be moving differently from the background by combining image registration and motion prediction.

For simplicity, the use-case is limited to the timing related aspects of two subsystems of the aerial video tracking system, as represented in Figure 1: a video frame processing system and a tracking and camera control system. As suggested by its name, the first subsystem processes the video frames sent by the camera. This includes embedding tracking data into the video, converting the frames to the required format and displaying a high quality video running at 25 frames per second on the monitor. The second subsystem performs motion prediction for the tracked object. Based on this prediction and the aircraft sensors data (position, direction, speed, etc.) it calculates new camera angles and sends instructions to control the camera.

We propose timing verification challenges related to each subsystem of the aerial video tracking use-case.

2.1 Challenge 1: Video Frame Processing

The functional view of the video frame processing subsystem is illustrated in Figure 2. It consists of a sequence of 4 functions processing the video frames from the camera to the display. The `Pre-processing` function removes reflections from the frames and normalizes the intensity of the individual pixels. The `Processing` function embeds tracking information into

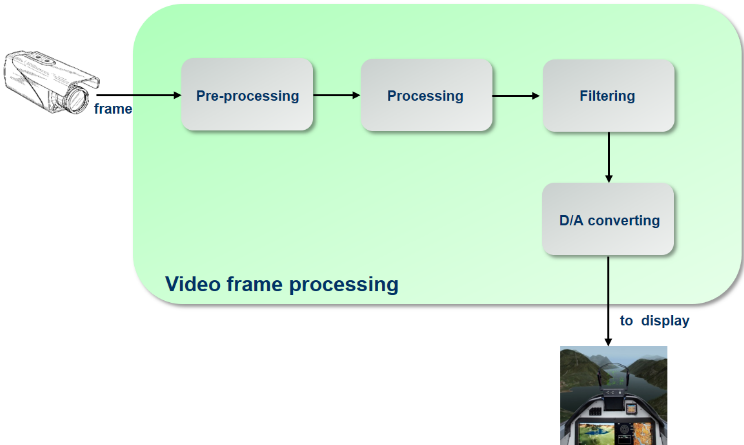


Figure 2 Functional view of the video frame processing subsystem.

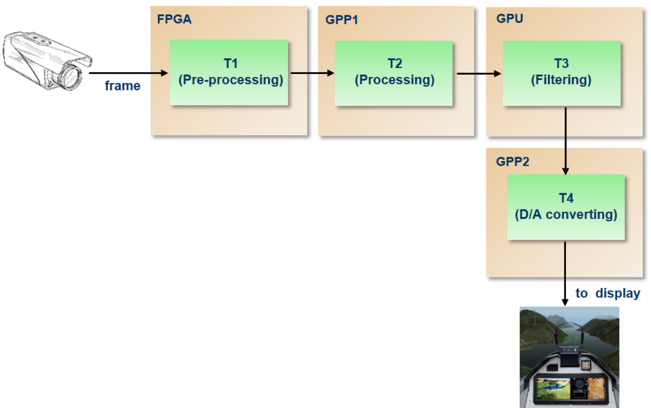


Figure 3 Architectural view of the video frame processing subsystem.

the pre-processed frames and executes zoom-in and zoom-out instructions. The `Filtering` function resizes the processed frames and removes noise. Finally, the `D/A converting` function converts the frames from digital to analog and sends them to the monitor.

For simplicity, we assume that each function is executed by a single task $\mathtt{T}_i$, as illustrated in Figure 3. All tasks are assumed to be mapped onto a different processor. Table 1 shows the execution time for tasks $\mathtt{T}_1$, $\mathtt{T}_3$ and $\mathtt{T}_4$ and the response time for task $\mathtt{T}_2$.[1]

Each frame sent by the camera activates task $\mathtt{T}_1$. The frames are sent strictly periodically with period $\mathtt{P}_1$, i.e., the time distance between two consecutive frames sent by the camera is *constant*. The exact value of $\mathtt{P}_1$ is however unknown since it may slightly vary from camera to camera. We know, however, that it ranges between $40\,ms - 0.01\%$ and $40\,ms + 0.01\%$. This *constant but unknown* value is a key aspect of the challenge.

[1] We need a response time rather than an execution time for $\mathtt{T}_2$ because it is running concurrently with other tasks that will be described in Challenge 2.

Table 1 Execution/response times of tasks from the video frame processing subsystem.

Task	Execution time
T_1	$bcet_1 = wcet_1 = 28\,ms$
T_3	$bcet_3 = wcet_3 = 8\,ms$
T_4	$bcet_4 = 1\,ms$; $wcet_4 = 10\,ms$
Task	**Response time**
T_2	$bcrt_2 = 17\,ms$; $wcrt_2 = 19\,ms$

After each execution, T_1 sends a frame through its output that activates task T_2. A register R is used for the communication between T_2 and T_3. At the end of each execution, T_2 overwrites the register content with the new frame.

When activated, task T_3 reads the current frame stored in the register R. For simplicity, we assume that there are no conflicts between read and write accesses to R. The activation of T_3 is strictly periodic, i.e., the value of period P_3 is *constant*. However, due to minor uncertainties in the clock implementation, the exact value of P_3 is unknown: it ranges between $\frac{40}{3}\,ms - 0.05\%$ and $\frac{40}{3}\,ms + 0.05\%$. Note that, since task T_3 is activated more frequently than task T_2, it will process the same register content more than once.

At the end of each execution, task T_3 produces a frame. Frames originating from the same register content are identical copies and are therefore assigned identical indices. The frames are inserted into a buffer Buf read by T_4. Buffer Buf has size n. For each frame, the following conditions must be met to get actually stored in Buf:

1. It is not full.
2. No other frame having the same index (i.e., identical copy) has already been stored in the buffer.

Otherwise, the frame is discarded. We call this a *smart insert* function. The time required to discard a frame or to store it in Buf can be ignored.

Task T_4 is activated strictly periodically, i.e., the value of period P_4 is *constant*. As for T_1 and T_3, the exact value of P_4 is again unknown, but we know that it is in the range $40\,ms \pm 0.01\%$. Each activation of T_4 leads to an execution. If buffer Buf is empty, the execution of T_4 takes $1\,ms$. Otherwise, T_4 consumes a single frame from the buffer, and in this case, its execution takes exactly $10\,ms$. Once a frame has been processed, task T_4 sends it (at the end of its execution) to be displayed on the monitor.

Communication between processors, access to register R between T_2 and T_3 and access to buffer Buf between T_3 and T_4 are considered to not consume any time.

Challenge 1A

The first part of the video frame processing timing verification challenge is to analyze the latency $E2E^{1A}$ (standing for end-to-end delay) of the frames sent by the camera that successfully reach the display.

1. Compute $E2E^{1A}_{max}$, the maximum value of this latency, for a buffer size $n = 1$.
2. Compute $E2E^{1A}_{max}$ for a buffer size $n = 3$.

Upper bounds for $E2E^{1A}_{max}$ are also of interest.

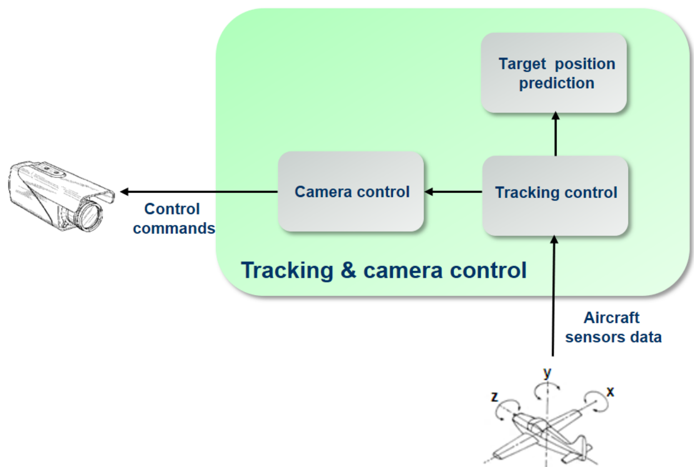


Figure 4 Functional view of the tracking and camera control subsystem.

Challenge 1B

Due to the small size of the buffer read by $\mathtt{T}_4$, it may happen that all frames with identical indices (i.e., all copies originating from the same register content between $\mathtt{T}_2$ and $\mathtt{T}_3$) are discarded at its entrance, e.g., when the period of $\mathtt{T}_4$ is smaller than the period of $\mathtt{T}_1$ (remember that the periods of $\mathtt{T}_1$, $\mathtt{T}_3$ and $\mathtt{T}_4$ are fixed, but their exact values are unknown). That is, no copy of the corresponding frame produced by the camera will ever reach the display. Losing frames is not very critical. However above a certain limit this may have an impact on the video quality and may be detected by the human eye.

The second part of the video frame processing timing verification challenge is therefore to analyze the distance between two frames produced by the camera that will be discarded at the buffer entrance, called `dist`. This distance `dist` may be expressed as a time distance or as a number of frames produced between two successive *losses*:

3. Compute the minimum loss distance $\mathtt{dist}_{min}$ for a buffer size $\mathtt{n} = 1$.
4. Compute $\mathtt{dist}_{min}$ for a buffer size $\mathtt{n} = 3$.

2.2 Challenge 2: Tracking and Camera Control

The functional view of the tracking and camera control subsystem is illustrated in Figure 4. It consists of 3 functions. The `Tracking control` function processes the aircraft sensors data (position, direction, speed, etc.), controls the whole tracking process and generates alerts and various tracking data. The `Target position` prediction function receives data about the aircraft speed, position and direction from the `Tracking control` function and performs motion prediction for the tracked object. The `Camera control` function receives data about the position of the tracked object from the `Tracking control` function and calculates a new angle for the camera based on the aircraft position, speed and direction and the tracked object motion prediction.

For simplicity, we assume that each function is executed by a single task, as illustrated in Figure 5. All tasks are mapped to a same processor GPP1 to which task $\mathtt{T}_2$ (which belongs to the video frame processing subsystem) is also mapped. All tasks are triggered by the arrival of data at their inputs. We assume fixed priority preemptive scheduling on the GPP1 with the following priority order: $\mathtt{T}_2 > \mathtt{T}_6 > \mathtt{T}_5 > \mathtt{T}_7$.

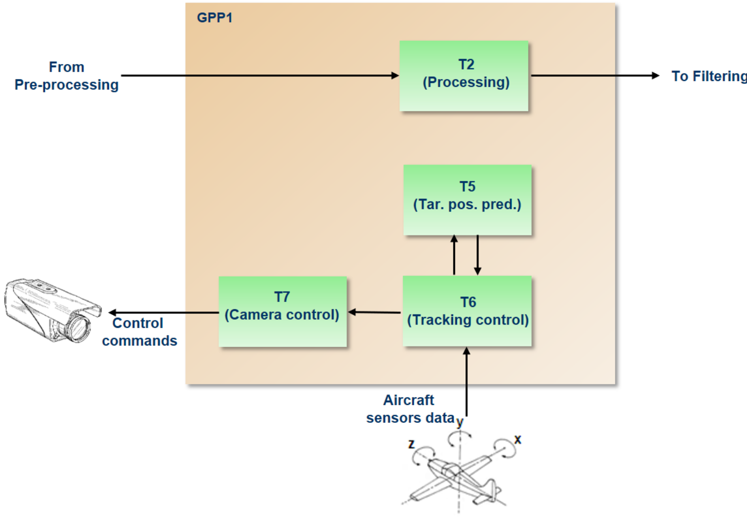


Figure 5 Architectural view of the tracking and camera control subsystem.

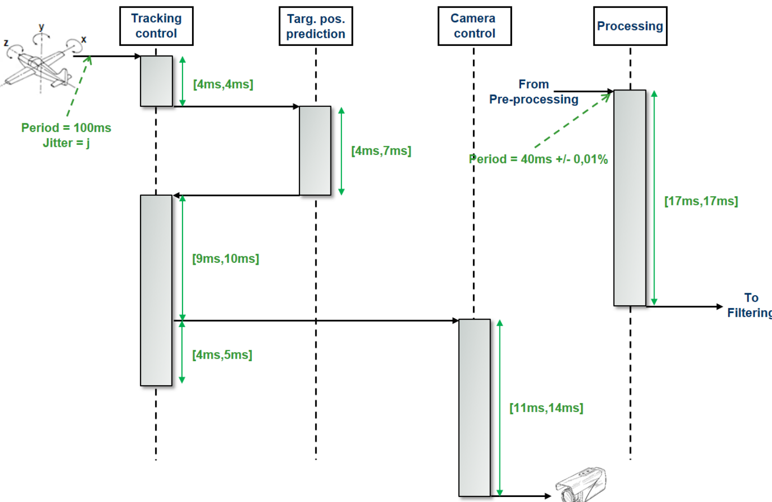


Figure 6 Sequence diagram of the functions on GPP1.

Table 2 Execution times of the individual functions and function segments by the tasks.

Function		Corresponding Execution Time [bcet, wcet] in *ms*
Tracking control	Segment 1	$[4, 4]$
	Segment 2	$[9, 10]$
	Segment 3	$[4, 5]$
Target position prediction		$[4, 7]$
Camera control		$[11, 14]$
Processing		$[17, 17]$

Figure 6 represents the sequence diagram of the functions on GPP1. The `Tracking control` function is activated periodically every 100 *ms*. Its periodic activation can however deviate by a jitter value `jitter`. As in Challenge 1, the `Process` function is activated strictly periodically, i.e., the time distance between two consecutive frames is *constant*. The exact value of the period is however unknown since it may slightly vary from camera to camera. However, we know that it ranges between $40\,ms - 0.01\%$ and $40\,ms + 0.01\%$.

A first segment of the `Tracking control` function is executed by task $\mathtt{T}_6$. Then the `Tracking control` function performs a synchronous call to the `Target position` prediction function and is suspended waiting for the answer. At the end of the `Target position` prediction function, task $\mathtt{T}_6$ resumes executing a second segment of the `Tracking control` function. An asynchronous call is then performed to the `Camera control` function executed by $\mathtt{T}_7$ while the last segment of the `Tracking control` function is executed by $\mathtt{T}_6$. All execution times by the tasks of the individual functions and function segments are given in Table 2.

Challenge 2A

The first part of the tracking and camera control timing verification challenge is to compute the best-case and worst-case end-to-end latencies from the activation of $\mathtt{T}_6$ to the termination of $\mathtt{T}_7$, $\mathtt{E2E}^{2A}_{min}$ and $\mathtt{E2E}^{2A}_{max}$ for different values of `jitter`:

1. Compute $\mathtt{E2E}^{2A}_{min}$ and $\mathtt{E2E}^{2A}_{max}$ for a jitter value $\mathtt{jitter} = 0\,ms$.
2. Compute $\mathtt{E2E}^{2A}_{min}$ and $\mathtt{E2E}^{2A}_{max}$ for a jitter value $\mathtt{jitter} = 20\,ms$.

Challenge 2B

Let us now assume that $\mathtt{T}_2$ and $\mathtt{T}_5$ have access to a shared resource (because the prediction requires information from the image). The resource is mutually exclusive and is protected by a priority ceiling protocol. The access to the shared resource takes 2 *ms* for both tasks. The second part of the tracking and camera control timing verification challenge is again to:

1. Compute $\mathtt{E2E}^{2B}_{min}$ and $\mathtt{E2E}^{2B}_{max}$ for a jitter value $\mathtt{jitter} = 0\,ms$.
2. Compute $\mathtt{E2E}^{2B}_{min}$ and $\mathtt{E2E}^{2B}_{max}$ for a jitter value $\mathtt{jitter} = 20\,ms$.
3. Compute the optimum priority assignment minimizing the worst-case latency `wcrt` for jitter values $\mathtt{jitter} = 0\,ms$ and $\mathtt{jitter} = 20\,ms$.

2.3 Discussion about the challenge

The challenge proposed by Thales is taken from a real industrial application. While the behavior described in the first part of the challenge conforms to a real application (only some execution times were modified), the second part of the challenge was synthesized based on

real timing behaviors encountered in several Thales real-time applications. A model of the application and its real-time behavior based on the Papyrus Modeling Environment and the MARTE profile was made available as a result of the WATERS 2015 workshop [13].

The second part of the challenge represents a classical scheduling verification problem that can be solved even manually by timing verification experts. This eases the evaluation of the quality of proposed solutions. The first part of the challenge, however, is rather untypical and more difficult to solve, thus requiring a tooled solution. In particular, the fact that the exact period for the arrival/sending of the video frames along the processing chain is constant but unknown represents a challenge for existing verification techniques. In addition, while the loss of data is usually not considered in classical scheduling problems, video frames may be discarded in the first part of the challenge depending on the buffer status.

3 Overview of the solutions provided

The proposed solutions fall into three categories: model checking, simulation and scheduling analysis. More specifically, the proposed solutions were based on the following tools:

- the timed model-checker Uppaal (Section 3.1);
- the parametric timed model-checker IMITATOR (Section 3.2);
- the timed model-checking framework Fiacre/Tina (Section 3.3);
- the tool for real-time systems schedulability analysis MAST (Section 3.4); and
- the simulation environment of CPAL (Section 3.5).

None of the proposed solutions could fully address the challenge, and it turned out that there were some misinterpretations of the model description, leading to very different answers for some of the solutions. In the rest of this section, we provide an overview of the various approaches. The objective is not so much to underline each individual contribution (some of which are outdated by now) than to provide a summary of the discussions that this challenge raised, and of the conclusions that were reached.

3.1 Solution using UPPAAL

The Uppaal solution [19] uses timed model checking as the main technique for answering challenge 1. The input formalism is timed automata (Ta) [2] – a dense-time extension of finite-state automata with a set of clocks measuring time – and the software used is Uppaal [14], a tool for the analysis of real-time systems described by a system of communicating timed automata.

A solution to challenge 1 was proposed[2] by a straightforward representation of the task system that almost directly maps to an implementation: each task is represented as a Ta executing a simple time- or event-triggered loop. Frames are represented by indexes counted modulo a sufficient[3] window size $\mathtt{N}_w$. For answering the challenge questions, for each frame `id`, an observer Ta `Thread(id)` is defined, which starts counting time when the frame is created and follows its progress through the tasks until the frame is lost or successfully processed. The variable execution time of task $\mathtt{T}_2$ is expressed by a time interval that is handled symbolically by the verification tool.

[2] Uppaal can also express Challenge 2, but the use of a schedulability analysis tool seemed to us a more obvious choice for that.

[3] The window size defines the number of frames which may be “in the system” simultaneously. In practice, we fix a size $\mathtt{N}_w$, and prove that it is sufficient by verifying that no frame lives longer than the corresponding time window $\mathtt{P}_1 \times \mathtt{N}_w$ (a new frame is created every $\mathtt{P}_1$ time units).

In order to model the assumed uncertainty about the periods of the different tasks, for each period, an integer constant (a "parameter"[4]) is defined, which is instantiated with a set of relevant values. This may be considered as a limitation, as the challenge specification specifies that the periods can take *any* value within some interval. Still, explicitly distinguishing different cases provides some insight: that only cases where $\mathtt{P}_1 < \mathtt{P}_4$ will lead to losses was not really a surprise, but the fact that even large deviations of $\mathtt{P}_3$ (of $\frac{1}{3}$ or more) have only very little influence on the results obtained, was probably less evident. The regularity of the results obtained for decreasing deviations (Table 3 shows results for $\frac{1}{3}$, $\frac{1}{6}$, $\frac{1}{12}$) allows us to extrapolate the results for smaller deviations.

The basic verification with Uppaal can be considered in the present case as a sort of compromise between simulation and parametric verification as proposed by Section 3.2. It provides exact results for given parameter instances, and there was no issue with scaling to compute minimal or maximal `E2E` or `dist` when excluding the first few hundred frames (600 and 1000 frames respectively in the results reported in Table 3). The solution uses the basic model-checking algorithms of Uppaal based on a symbolic clock representation. This allows us to achieve exact verification results for the model under study. Using relevant properties, a set of timings can thus be derived. The simulation capacities of Uppaal were also used but only to get some initial understanding or for debugging. Results were then confirmed by model checking.

3.2 Solution using IMITATOR

The IMITATOR solution [20] uses parametric timed model checking as the main technique to derive the end-to-end timings for answering Challenge 1. The underlying formalism is parametric timed automata [3], an extension of timed automata [2] with unknown timing parameters, in addition to the clocks used in timed automata invariants and guards. The software used is IMITATOR [5], a parametric timed model checker taking as input networks of parametric timed automata augmented with useful features such as rational-valued variables, stopwatches, etc.

The key solution to solve Challenge 1A is to consider a single *arbitrary* frame processing. Thanks to the *symbolic* representation of the state space offered by IMITATOR, the system can start from an arbitrary state, and perform a finite number of discrete actions simulating this arbitrary frame. Measuring the time from its input to the output, IMITATOR therefore derives a (parametric) best and worst case time. Parameters (i.e., unknown constants) are used to model the uncertain periods; an additional parameter $\mathtt{E2E} \geq 0$ is also used to represent the end-to-end latency of the target frame. And, for a given run, this value is *unique* as a single frame is considered.

The key aspect of this solution is the use of rational-valued *timing parameters* to model perfectly the unknown (but constant) periods. This solution is correct, as opposed to intervals – in which case the actual period can vary at each cycle, which is not the intended specification in the challenge. The solutions to Challenge 1A (see Table 3) for both buffer sizes are exact, while the proposed solution to Challenge 1B is only approximated, due to the increased system complexity.

In addition, a solution is derived for Challenge 2A by the same authors [20] using analytical methods, and then confirmed by IMITATOR. The extension of the solution to Challenge 2B was not modeled in [20] due to lack of time but seems straightforward to the authors.

[4] Note that this is different from the timing parameters (unknown constants) that will be described in Section 3.2, as they are handled manually here.

Finally, this way of modeling the solution was a source of inspiration for subsequent work on addressing scheduling problems under uncertainty using (extensions of) parametric timed automata [6].

3.3 Solution using Fiacre and Tina

The Tina approach [8] relies on several models to answer questions from Challenge 1, defined using either Time Petri Nets (TPN) [16] or Fiacre [7], a component-based specification language that extends TPN with data and priorities. All our models derive from a common specification, but are each specifically instrumented in order to check a given property: minimal and maximal traversal time; influence of different clock rates; possibility to lose frames; etc.

Our results are computed using the model-checking tool `sift`, part of the Tina toolbox [9], that provides state-space exploration and reachability algorithms for both TPN and Fiacre models. Like with IMITATOR and Uppaal, our approach is based on a dense-time hypothesis. We are not totally faithful to the specification though. In particular, we decided to use time intervals to model the uncertainty on the period instead of a fixed value inside an interval. This means that, inside the same execution trace, the periods of a task may vary. Because of this, each experiment that we perform only returns approximate results. Nonetheless, by using several iterations, and by using together methods that over- or under-approximate the possible behaviors, we were able to compute results within a given accuracy threshold (we use $10\mu s$ in Table 3). It would have been possible to derive an "exact" model using an extension of TPN with stopwatches, but this is way more computationally expensive and not usable in practice.

The frame processing challenge turned out to be a very interesting case study for our model-checking toolbox. First, since the description is highly modular, it is well-suited for component-based modeling languages. Also, it provides a good motivation for the use of high-level data structures in a specification language. In our Fiacre models, for instance, we use a queue of identifiers with a dedicated insertion function to model an unbounded number of frames. As a result, in the case $n = 3$, we can prove that there can be at most 5 different frames traveling at a given time in the pipeline, without the need to provide a bound *a priori*. Finally, many requirements can be reduced to safety properties, that is, checking that some "bad state" cannot occur. In this case, we often do not need to explore the whole state space of the system to return a meaningful result. We can also use more aggressive abstractions, that are able to speed up our computations. We give an example of such optimization in [8] that was able to reduce some model-checking tasks by a factor of $\times 250$ (from about $100s$ to less than $0.4s$).

3.4 Solution using MAST

This approach [15] is based on the Modeling and Analysis Suite for Real-Time Applications (MAST) [12], which is an open source set of tools for developing real-time applications to perform various kinds of schedulability analysis.

For Challenge 1, the authors focused on a different research question than the one posed by Thales, namely finding the worst-case conditions under which the system is still schedulable (i.e., it loses no frame). Under these assumptions, 3 out of the 4 questions can be answered by combining the information directly provided by MAST and ad-hoc external methods. In particular when $n = 3$, results for the required latencies and distances can be calculated as the maximum number of frames enqueued is 2. Furthermore, distance between two frames can be calculated for $n = 1$ based on the MAST method for calculating the buffer size.

The relevant characteristics for modeling Challenge 2 are: Each function is mapped to one task. All tasks are in the same processor using FP preemptive scheduling with priority order: $\mathtt{T}_2 > \mathtt{T}_6 > \mathtt{T}_5 > \mathtt{T}_7$. All tasks are triggered by the arrival of data at their inputs. $\mathtt{T}_7$ is invoked from an action in the middle of $\mathtt{T}_6$. Here the authors need to define the models to explore timing responses and priority assignments for tasks in GPP1 that encompass a fork, activation jitter for one of the two concurrent flows of execution, and shared resources. The approach in this case is to adapt the models used to the sets of equivalent cases for the relative values of priorities and the capacities of the analysis techniques available.

The presence of a fork leads to a multipath model for which only the holistic technique was available [15]. If the fork is decomposed using the values of the priorities to linearize it, then, this linear model is analyzed using offset-based techniques. A newer technique enabling the offset-based analysis of the multipath model has been more recently published in [4].

3.5 Solution using CPAL

This approach [1] is based on CPAL (Cyber Physical Action Language) [10], a language meant to model, simulate, verify and program Cyber-Physical Systems (CPS). CPAL does not provide any fully automatic analysis to compute a solution to the FMTV challenge. However, it helps to identify and validate best and worst-case scenarios. Thus, the language features of CPAL used for solving the challenges are the formal description, the edition, graphical representation and simulation of CPS models. The two challenges as specified in the original description of the FMTV challenge could come out as a little ambiguous: the CPAL model in contrast must be unambiguous and adhere to the well-defined semantics of the execution and simulation environment. Furthermore, the models can be written directly in the graphical CPAL editor, providing an immediate feedback on where and in which aspects the informal problem descriptions were underspecified. Since CPAL is a domain-specific language with native support for real-time scheduling, the modeling effort is small, and the risk of translation errors is limited.

4 Discussion

Let us now discuss briefly the results obtained using the different tools, and provide some overall conclusions. Table 3 shows an overview of the results for Challenge 1, and Table 4 those for Challenge 2. A first outcome is that no tool solved all challenges, even with an approximate solution. Challenge 1A is the one with the highest number of answers, with 4 out of 5 tools being able to make some answers – this challenge also has the highest diversity rate in terms of results (see below). Then, Challenge 1B had 4 out of 5 tools offering a solution, while the other challenges (2A and 2B) had 3 out of 5 tools being able to provide a (partial) solution. A second outcome is that the range of solutions is highly diverse: that is, different tools obtained different answers. This is not entirely surprising, as several tools knowningly analyzed slightly different problems (as detailed in Section 3), leading to completely different answers. Still, a certain (partial) consensus can be obtained by looking closely at the results, as will be discussed later.

[5] $\mathtt{P}_1 = \mathtt{P}_4 = 3 \times \mathtt{P}_3$.
[6] All measured results were below or equal to 146, the true worst-case is thus at least $146ms$.

Table 3 Overview of the results for Challenge 1.

Tool	1A: E2E or $E2E_{max}$ *ms*		1B: dist or $dist_{max}$ *ms* or frames	
	n = 1	n = 3	n = 1	n = 3
UPPAAL				
no deviation[5]	$[63, 118.33]$ *ms*	$[63, 118.33]$ *ms*	none	none
$P_4 = P_1 - \frac{1}{3}$	$[63, 118]$ *ms*	$[63, 118]$ *ms*	none	none
$P_4 = P_1 + \frac{1}{3}$	$[63, 145.33]$ *ms*	$[63, 226]$ *ms*	$[42, 240]$ frames	$[79, 480]$ frames
after $\geq$ 600 fr	$[90, 145.33]$ *ms*	$[170.66, 226]$ *ms*	$[79, 161]$ frames	$[79, 161]$ frames
$P_4 = P_1 + \frac{1}{6}$	$[63, 145.16]$ *ms*	$[63, 225.66]$ *ms*	$[84, 480]$ frames	$[159, 958]$ frames
after $\geq$ 1000 fr	$[89.84, 145.16]$ *ms*	$[162.66, 225.66]$ *ms*	$[159, 321]$ frames	$[159, 321]$ frames
$P_4 = P_1 + \frac{1}{12}$	$[63, 145.08]$ *ms*	–	–	$[319, -]$ frames
IMITATOR	$[63, 145.008]$ *ms*	$[63, 225.016]$ *ms*	–	$< 5,000$ frames
Fiacre/Tina	$[89.66, 145.33]$ *ms*	$[89.66, 225.33]$ *ms*	2 frames	between 35 and 4085 frames
MAST	–	If $P_1 \geq 39.996$ *ms* and $n \geq 2$ $[63, 118.344]$ *ms*	55.344 *ms*	28 *ms* for the highest possible frame production rate of $P_1 = 28$ *ms*
CPAL simulation	≥ 146 *ms* [6]	≥ 220 *ms*	2 frames	4 400 frames
CPAL analysis	$[89.6656, 146]$ *ms* ($E2E_{min}$ is 63 *ms* for the 1st frame)	$[89.6656, 226]$ *ms* ($E2E_{min}$ is 63 *ms* for the 1st frame)	–	–

Table 4 Overview of the results for Challenge 2.

Tool	2A: [bcrt,wcrt]		2B: [bcrt,wcrt]		
	jitter = 0 *ms*	jitter = 20 *ms*	jitter = 0 *ms*	jitter = 20 *ms*	Optimization
UPPAAL	–	–	–	–	–
IMITATOR	$[49, 74]$ *ms*	$[49, 94]$ *ms*	–	–	–
Fiacre/Tina	–	–	–	–	–
MAST	$[32, 74]$ *ms*	$[32, 94]$ *ms*	$[32, 78]$ *ms*	$[32, 98]$ *ms*	$T_5 > T_7 > T_6 > T_2$; if jitter = 0 *ms* then wcrt = 39 *ms*; if jitter = 20 *ms* then wcrt = 59 *ms*
CPAL simulation	–	–	–	–	–
CPAL analysis	$[33, 75]$ *ms*	$[33, 112]$ *ms*	$[33, 75]$ *ms*	$[33, 112]$ *ms*	$T_7 > T_6 > T_5 > T_2$ (37 *ms*)

4.1 Modeling

One important lesson from this challenge is related to the fact that there were ambiguities in the description, leading to misinterpretations (the same problem was understood and interpreted differently by different participants to the challenge). This shows the importance of providing unambiguous models to avoid misunderstandings. Based on this observation, Thales published a consolidated version of the challenge [13] including a Papyrus model annotated with MARTE.

A second observation related to modeling is that the ease of modeling differed quite a lot between the different tools. Some were quite well adapted to the nature of the challenge while others required quite heavy work to address the problem at hand. This made it difficult to validate the correctness of the proposed models.

4.2 Different solutions

Focusing on the solutions that did model the constant, yet unkown period parameter, a certain (partial) consensus can be obtained on some results.

For Challenge 1A, the lower bound for both $n = 1$ and $n = 3$ is around 89 ms, which is obtained by most tools (with some differences due to rounding approximations). The value 63 ms found by IMITATOR and UPPAAL corresponds to the first frame, which has a specific behavior.

In addition, 4 out of 5 tools agree that the upper bound is between 145 and 146 ms for $n = 1$, and between 225 and 226 ms for $n = 3$. Most importantly, the solution obtained by simulation for $n = 1$ (CPAL: 146 ms), and that acts as a *lower bound* on the desired maximum, matches the result obtained by parametric model checking (IMITATOR: 145.008 ms), which acts as an *upper bound* on the desired maximum. (The fact that the "lower" upper bound found with simulation by CPAL is lower than the upper bound found by model-checking comes from the discrete-time setting of CPAL simulation tool.) That is, the desired maximum is necessarily in $(145, 145.008]$ ms.

Concerning Challenge 1B, results are completely different. This challenge was the most difficult according to the participants. The actual value is probably around the result of CPAL, i.e., 4 400 frames, according to the participants, but no conclusion has been reached.

Although it was considered easy according to the participants, results for Challenges 2A and 2B vary quite a lot and more research would be needed to converge towards a consensual answer.

4.3 No perfect tool

The developers of the challenge were hoping to have a single tool being able to solve all sub-challenges immediately. Instead, it turned out that different tools solved parts of the challenge, and that had different strengths and weaknesses: ease of modeling, level of automation of the verification process, computational complexity of verification, reliability of the results, etc.

An additional lesson learned from the challenge is that the use of different techniques to solve the same timing verification problem may increase the confidence in the produced results. For example by applying simulation to an execution scenario identified by scheduling analysis as worst-case, we may determine if the analysis results are too pessimistic or not and thus evaluate the analysis quality. This point is very important in the industrial development of real-time systems since overestimation margins are usually small (over approximation up to 20 % are in general accepted in industry, larger over-approximations are refused due to the cost of the corresponding over dimensioned solutions).

For example, in Challenge 1A, combining the formal methods based result of IMITATOR (a possibly overapproximate result) with the simulation based result of CPAL (a possibly underapproximate result, due to the weakness of simulation) allowed us to confirm the exactness of both methods. Similarly, in [17], two different formalisms (parametric timed Petri nets and parametric timed automata) were used in academic software developed by two different teams in order to solve a problem close to the problem described in the current paper. The fact that both solutions led to the same result increases significantly the confidence we can have in these results. Finally, some tools could potentially be used together in a sequential manner: for example, the results of IMITATOR could be fed into non-parametric tools such as Uppaal or CPAL, so as to first infer a set of possible solutions, and then verify them using non-parametric methods. It remains an open question whether we can easily derive a holistic tool that solves the complete challenge.

5 Conclusion and perspectives

In this paper, we have presented the main features and lessons learned from the first edition of what has now become the ECRTS industrial challenge. This verification challenge, proposed by Thales, was first discussed in 2014 as part of a dedicated workshop (FMTV, a satellite event of the FM 2014 conference), and solutions were discussed for the first time at the WATERS 2015 workshop.

The use case for the verification challenge is an aerial video tracking system. The first part of the challenge focuses on the video frame processing system. It consists in computing maximum values of the end-to-end latency of the frames sent by the camera to the display, for two different buffer sizes, and then the minimum duration between two consecutive frame losses. The second challenge is about computing end-to-end latencies on the tracking and camera control for two different values of jitter. Solutions based on five different tools – Fiacre/Tina, CPAL (simulation and analysis), IMITATOR, Uppaal and MAST – were submitted for discussion at WATERS 2015. While none of these solutions provided a full answer to the challenge, a combination of several of them did allow to draw some conclusions.

From Thales' point of view, the timing verification challenge was a success. The submitted solutions to the challenge represent all three model-based timing verification techniques: timing simulation, scheduling analysis and model checking using timed automata. This gave Thales the opportunity to evaluate these techniques and better understand their strengths and weaknesses. As a direct result of the challenge, point to point collaborations were set-up with several participants. Thales and RTaW set-up a research project (FUI WARUNA) focusing on timing verification and its integration into the industrial design process. The solution using IMITATOR was further evaluated by a trainee at Thales and the results were published in a joint paper [17]. Another joint paper between Thales and UNICAN about the integration of response-time analysis and optimization in the industrial design process was also published [18].

From an academic perspective, we also consider this first edition as a success. This event provided a unique opportunity for researchers to try out their pet tool on a problem of industrial relevance, and to discuss and compare its performance and limitations with colleagues in a collaborative way. Although the challenge was fairly simple in its formulation, it was sufficient to underline the gap that exists between academic tools and industrial real-time systems. Still, the proposed verification problem did strike a difficult balance between practical relevance and feasibility: it was challenging, yet within reach of academic tools. Furthermore, the fact that a combination of solutions could provide a much better

answer to the challenge than any single tool opens up a scientific line of research that has not been explored since: when is it useful to combine different formalisms and analysis techniques to solve a verification problem, and how can this be done in an efficient manner?

This first positive experience triggered a series of subsequent industrial challenges at WATERS, most notably two iterations by Bosch GmbH. The WATERS challenge has now grown to become a full-fledged event of the ECRTS conference.

Let us conclude this paper by a few general comments. We believe that the noncompetitive way in which the challenge was organized contributed a lot to its success. The review process focused on clarifying assumptions and limitations of the proposed solutions and did not intend to declare a winner. In fact, participants to the challenge were invited to review other submissions, considering that they were the best suited for this. This process ensured that everyone could focus on the scientific discussions without the additional burden of organizing or participating in a competition.

More generally, events like this help bringing closer researchers in academia and industrial practitioners from different application domains. For real-time systems research, which tends to measure its success by its practical relevance and transfer to industry, this is obviously very useful. Now, this raises an interesting question, which is rarely discussed by the community: how much research should focus on addressing the needs of industry? A useful expansion of the challenge in future years could experiment with other types of collaborative efforts, e.g., research approaches that would investigate alternative trajectories to those of immediate interest to industry – for example, approaches that would strike a different balance between predictability and complexity – or even focusing on society's needs via another proxy than industry – for example by collaborating with nonprofit organizations. In any case, one can only hope that more use cases will be made available to the community in the future, and that venues for discussing them and potential solutions will spread.

References

1 Sebastian Altmeyer, Loïc Fejoz, and Nicolas Navet. Using CPAL to model and validate the timing behaviour of embedded systems. Solutions to the FMTV Challenge, Informal proceedings of the 6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2015), July 2015. URL: `https://nicolas.navet.eu/publi/challenge.pdf`.

2 Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994. `doi:10.1016/0304-3975(94)90010-8`.

3 Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. Parametric real-time reasoning. In S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal, editors, *STOC*, pages 592–601, New York, NY, USA, 1993. ACM. `doi:10.1145/167088.167242`.

4 Andoni Amurrio, Ekain Azketa, J. Javier Gutiérrez, Mario Aldea Rivas, and Michael González Harbour. Response-time analysis of multipath flows in hierarchically-scheduled time-partitioned distributed real-time systems. *IEEE Access*, 8:196700–196711, 2020. `doi:10.1007/s10009-017-0467-0`.

5 Étienne André. IMITATOR 3: Synthesis of timing parameters beyond decidability. In Rustan Leino and Alexandra Silva, editors, *CAV*, volume 12759 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2021. `doi:10.1007/978-3-030-81685-8_26`.

6 Étienne André, Emmanuel Coquard, Laurent Fribourg, Jawher Jerray, and David Lesens. Parametric schedulability analysis of a launcher flight control system under reactivity constraints. *Fundamenta Informaticae*, 182(1):31–67, September 2021. `doi:10.3233/FI-2021-2065`.

7 Bernard Berthomieu, Jean-Paul Bodeveix, Patrick Farail, Mamoun Filali, Hubert Garavel, Pierre Gauffilet, Frédéric Lang, and François Vernadat. Fiacre: an intermediate language for model verification in the topcased environment. In *Embedded Real Time Software (ERTS)*, 2008.

8 Bernard Berthomieu, Silvano Dal Zilio, and Didier Le Botlan. Latency analysis of an aerial video tracking system using Fiacre and Tina. Solutions to the FMTV Challenge, Informal proceedings of the 6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2015), July 2015. URL: `https://www.ecrts.org/forum/download/FMTV15_Solution_Fiacre_Tina.pdf`.

9 Bernard Berthomieu and François Vernadat. Time Petri nets analysis with TINA. In *QEST*, pages 123–124. IEEE Computer Society, 2006. `doi:10.1109/QEST.2006.56`.

10 CPAL : A Cyber-Physical Action Language. URL: `https://www.designcps.com/model-based-design-with-cpal/`.

11 Formal Methods for Timing Verification (FMTV) Workshop, a satellite event of FM'14, the 19th International Symposium on Formal Methods. URL: `https://www.comp.nus.edu.sg/~pat/FM2014/`.

12 Michael González Harbour, J. Javier Gutiérrez García, José Carlos Palencia Gutiérrez, and J. M. Drake Moyano. MAST: Modeling and analysis suite for real time applications. In *ECRTS*, pages 125–134. IEEE Computer Society, 2001. `doi:10.1109/EMRTS.2001.934015`.

13 Rafik Henia. Consolidated version of the WATERS industrial challenge by Thales including a corresponding model in Papyrus. URL: `https://www.ecrts.org/forum/viewtopic26ef.html?f=34&t=86&sid=d74079af129d5480a5ac4fd1778eecc1`.

14 Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997. `doi:10.1007/s100090050010`.

15 Julio L. Medina, Juan M. Rivas, J. Javier Gutiérrez, and Michael González Harbour. Solving the 2015 FMTV challenge using response time analysis with MAST. Solutions to the FMTV Challenge, Informal proceedings of the 6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2015), July 2015. URL: `https://www.ecrts.org/forum/download/FMTV15_Solution_MAST.pdf`.

16 Philip Meir Merlin. *A study of the recoverability of computing systems.* PhD thesis, University of California, Irvine, CA, USA, 1974.

17 Baptiste Parquier, Laurent Rioux, Rafik Henia, Romain Soulat, Olivier H. Roux, Didier Lime, and Étienne André. Applying parametric model-checking techniques for reusing real-time critical systems. In Cyrille Artho and Peter Csaba Ölveczky, editors, *FTSCS*, volume 694 of *Communications in Computer and Information Science*, pages 129–144. Springer, November 2016. `doi:10.1007/978-3-319-53946-1_8`.

18 Juan Maria Rivas, J. Javier Gutiérrez, Mario Aldea Rivas, César Cuevas, Michael González Harbour, José María Drake, Julio L. Medina, Laurent Rioux, Rafik Henia, and Nicolas Sordon. An experience integrating response-time analysis and optimization with an MDE strategy. In Paolo Milazzo, Dániel Varró, and Manuel Wimmer, editors, *MELO @ STAF*, volume 9946 of *Lecture Notes in Computer Science*, pages 303–316. Springer, 2016. `doi:10.1007/978-3-319-50230-4_23`.

19 Lijun Shan and Susanne Graf. Timing verification of an aerial video tracking system using UPPAAL. Solutions to the FMTV Challenge, Informal proceedings of the 6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2015), July 2015. URL: `https://www.ecrts.org/forum/download/FMTV15_Solution_UPPAAL.pdf`.

20 Youcheng Sun, Étienne André, and Giuseppe Lipari. Verification of two real-time systems using parametric timed automata. Solutions to the FMTV Challenge, Informal proceedings of the 6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2015), July 2015. `https://lipn.univ-paris13.fr/ andre/documents/verification-of-two-real-time-systems-using-parametric-timed-automata.pdf`.

21 Youcheng Sun, Étienne André, and Giuseppe Lipari. Verification of an aerial video system. Solutions to the consolidated 2015 industrial Challenge, Informal proceedings of the 8th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2017), July 2017. URL: `https://www.ecrts.org/forum/download/FMTV17_submitted.pdf`.

22 Website of the WATERS'15 workshop. URL: `http://waters2015.inria.fr/`.