

18th International Workshop on Worst-Case Execution Time Analysis

WCET 2018, July 3, 2018, Barcelona, Spain

Edited by

Florian Brandner



Editors

Florian Brandner
Télécom ParisTech
Université Paris-Saclay
Paris, France
florian.brandner@telecom-paristech.fr

ACM Classification 2012

Computer systems organization → Real-time systems, Theory of computation → Program analysis, Software and its engineering → Software performance, Software and its engineering → Software verification and validation

ISBN 978-3-95977-073-6

Published online and open access by

Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at <http://www.dagstuhl.de/dagpub/978-3-95977-073-6>.

Publication date

September, 2018

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 <http://dnb.d-nb.de>.

License

This work is licensed under a Creative Commons Attribution 3.0 Unported license (CC-BY 3.0): <http://creativecommons.org/licenses/by/3.0/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/OASlcs.WCET.2018.0

ISBN 978-3-95977-073-6

ISSN 2190-6807

<http://www.dagstuhl.de/oasics>

OASlcs – OpenAccess Series in Informatics

OASlcs aims at a suitable publication venue to publish peer-reviewed collections of papers emerging from a scientific event. OASlcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

Editorial Board

- Daniel Cremers (TU München, Germany)
- Barbara Hammer (Universität Bielefeld, Germany)
- Marc Langheinrich (Università della Svizzera Italiana – Lugano, Switzerland)
- Dorothea Wagner (*Editor-in-Chief*, Karlsruher Institut für Technologie, Germany)

ISSN 2190-6807

<http://www.dagstuhl.de/oasics>

■ Contents

Preface	
<i>Florian Brandner</i>	0:vii
Committees	
.....	0:ix
Invited Papers	
Mixed Feelings About Mixed Criticality	
<i>Reinhard Wilhelm</i>	1:1–1:9
Regular Papers	
Formal Executable Models for Automatic Detection of Timing Anomalies	
<i>Mihail Asavoaie, Belgacem Ben Hedia, and Mathieu Jan</i>	2:1–2:13
Reducing Timing Interferences in Real-Time Applications Running on Multicore Architectures	
<i>Thomas Carle and Hugues Cassé</i>	3:1–3:12
Toward Contention Analysis for Parallel Executing Real-Time Tasks	
<i>Fabrice Guet, Luca Santinelli, Jérôme Morio, Guillaume Phavorin, and Eric Jenn</i>	4:1–4:13
A New Hybrid Approach on WCET Analysis for Real-Time Systems Using Machine Learning	
<i>Thomas Huybrechts, Siegfried Mercelis, and Peter Hellinckx</i>	5:1–5:12
TASKers: A Whole-System Generator for Benchmarking Real-Time-System Analyses	
<i>Christian Eichler, Tobias Distler, Peter Ulbrich, Peter Wägemann, and Wolfgang Schröder-Preikschat</i>	6:1–6:12
Experimental Evaluation of Cache-Related Preemption Delay Aware Timing Analysis	
<i>Darshit Shah, Sebastian Hahn, and Jan Reineke</i>	7:1–7:11
Embedded Program Annotations for WCET Analysis	
<i>Bernhard Schommer, Christoph Cullmann, Gernot Gebhard, Xavier Leroy, Michael Schmidt, and Simon Wegener</i>	8:1–8:13
Fine-Grain Iterative Compilation for WCET Estimation	
<i>Isabelle Puaut, Mickaël Dardaillon, Christoph Cullmann, Gernot Gebhard, and Steven Derrien</i>	9:1–9:12



■ Preface

It is my great pleasure to present you the proceedings of the *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*, which was collocated with the Euromicro Conference on Real-Time Systems (ECRTS 2018) in Barcelona, Spain. The workshop was held prior to the main conference on July 3, 2018, and consisted of 8 presentations of regular papers and an invited keynote talk by Reinhard Wilhelm. Each paper received 5 reviews from members of the program committee and the final selection was based on a lively online discussion. The proceedings that you have in front of you here were published only after the workshop in order to allow authors to integrate feedback from discussions at the workshop. Workshop participants still had access to preliminary versions of the presented papers roughly two weeks before the event in order to facilitate these discussions.

Despite the tight schedule – reviewers had less than 3 weeks to complete their work – all reviews were delivered on-time, which allowed for a lively and insightful online discussion. I thus would like to say a big thank you to the members of the program committee and the external reviewers for their great work and effort. I would also like to thank the members of the steering committee. Most notably, I thank Jan Reineke, who chaired the workshop in 2017, for sharing his past experience and giving me advise. In a similar vein, I would like to thank Michael Wagner from OASICs, who helped with the preparations for the proceedings here. A special thank you goes to Reinhard Wilhelm, who not only gave an excellent keynote that sparked a vivid debate, but also covered his travel expenses out of his own pocket. Last, but not least, I would like to thank the authors for their excellent contributions and the workshop participants for their stimulating questions and comments.

I am also grateful to the organizers of the ECRTS 2018 conference – notably Francisco J. Cazorla and Gerhard Fohler – for offering me their ideas and help, and quickly responding to questions whenever I needed information regarding administrative issues.

It was a great pleasure to organize and chair WCET 2018 and I am looking forward to upcoming editions of the workshop.

Palaiseau, August 7, 2018
Florian Brandner



■ Committees

Program Chair

- Florian Brandner – Télécom ParisTech, Université Paris-Saclay

Program Committee

- Armelle Bonenfant – Université Toulouse III – Paul Sabatier, France
- Björn Lisper – Mälardalen University, Sweden
- Claire Maiza – Grenoble INP/Verimag, France
- Clément Ballabriga – Lille 1 University, France
- Jan Reineke – Saarland University, Germany
- Jakob Zwirchmayr – TTTech Computertechnik AG – Automotive, Austria
- Jaume Abella – Barcelona Supercomputing Center, Spain
- Jörg Mische – Augsburg University, Germany
- Kartik Nagar – Purdue University, United States
- Luca Santinelli – ONERA, France
- Martin Schoeberl – Technical University of Denmark, Denmark
- Peter Ulbrich – Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
- Simon Wegener – AbsInt Angewandte Informatik GmbH, Germany
- Tullio Vardanega – University of Padua, Italy
- Xenofon Koutsoukos – Vanderbilt University, USA

External Reviewers

- Hugues Cassé – Université Toulouse III – Paul Sabatier, France
- Luca Pezzarossa – Technical University of Denmark, Denmark
- Eleftherios Kyriakakis – Technical University of Denmark, Denmark
- Pascal Sotin – Université Toulouse Jean Jaurès, France
- Marianne De Michiel – Université Toulouse III – Paul Sabatier, France
- Linus Källberg – Mälardalen University, Sweden
- Oktay Baris – Technical University of Denmark, Denmark
- Tórrur Biskopstø Strøm – Technical University of Denmark, Denmark
- Peter Wägemann – Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

Steering Committee

- Björn Lisper – Mälardalen University, Sweden
- Isabelle Puaut – University of Rennes I/IRISA, France
- Jan Reineke – Saarland University, Germany




Mixed Feelings About Mixed Criticality

Reinhard Wilhelm

Informatik, Universitaet des Saarlandes, Saarland Informatics Campus

Saarbruecken, Germany

wilhelm@cs.uni-saarland.de

 <https://orcid.org/0000-0002-1825-0097>

Abstract

I point to some challenges for WCET analysis offered in the transition to integrated mixed-criticality systems (MCSs) and to multi-core platforms, claim that proposed certification standards are inadequate, show that the MCS model heavily used by the scheduling community is fraught, and clarify why the traditional abstract interface between WCET analysis and schedulability analysis is obsolete.

A central point is the insistence on sound approaches. I give a detailed account of how the most rigid certification procedures, those of the avionics domain, are satisfied, to defend the validity of my claims.

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

Keywords and phrases WCET analysis, mixed criticality systems, multi-core platforms, scheduling, schedulability

Digital Object Identifier 10.4230/OASIScs.WCET.2018.1

Category Invited Paper

Funding Funding for our research was obtained from Deutsche Forschungsgemeinschaft under Collaborative Research Centers 124 and AVACS and the Project Daedalus in the European 5th Framework Programme.

Acknowledgements I would like to thank Rob Davis, Markus Pister, Gernot Gebhard, and Jan Reineke for support in writing this paper. Rob Davis tried to convince me of the value of the Vestal model of MCS. Markus Pister enlightened me about certification processes relating to safety-critical systems. Gernot Gebhard let me use an illuminating figure from his PhD thesis. Jan Reineke provided valuable comments on earlier versions of this article.

1 Introduction

The general setting for WCET analysis is that a set of hard real-time tasks is to be executed on a given hardware platform. Being hard real-time tasks means having associated deadlines within which they have to finish their execution. *Timing Verification* has to verify that these timing constraints are satisfied. Traditionally Timing Verification is split into a *WCET analysis*, which determines upper bounds on the execution times, and a *schedulability analysis*, which takes these upper bounds and attempts to verify that the given set of tasks when executed on the given platform will all respect their deadlines.

There are two strong trends in the embedded-systems industry, the transition from single-core to multi-core execution platforms and the transition from federated systems to integrated systems comprising components with different levels of criticality.



© Reinhard Wilhelm;

licensed under Creative Commons License CC-BY

18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018).

Editor: Florian Brandner; Article No. 1; pp. 1:1–1:9

OpenAccess Series in Informatics



OASIS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The criticality level (aka Safety integrity level (SIL)) of a component is derived from the impact of a failure of the component on the functioning of the system. It determines the size of the effort to deliver assurance of the correct functioning of the component. A mixed criticality system (MCS) is one that has two or more distinct criticality levels. Up to five levels exist in the standards, e.g. the IEC 61508, DO-178B and DO-178C, DO-254 and ISO 26262.

For architectures with instructions that had constant execution times, WCET analysis methods using *Timing Schemata* [14] were the method of choice. Timing schemata describe how (bounds on) the execution times of a programming-language construct were composed from the (bounds on) the execution times of its components. These methods would thus do structural induction over the structure of a program and determine bounds for ever bigger parts of the program.

Performance-enhancing architectural components such as caches, pipelines, and speculation made previous methods for WCET analysis using *Timing Schemata* [14] obsolete. Execution times do not compose any longer because instruction execution-times are now dependent on the execution state in which they were executed. In the composition $A;B$ the execution time of statement B depends on the execution state produced by statement A . The variability of execution times grew with several architectural parameters, e.g. the cache-miss penalty and the costs for pipeline stalls and for control-flow mis-predictions.

1.1 The Central Idea – Proving Safety Properties

We started off to solve the WCET problem for architectures with state-dependent execution times. Let me describe the central idea behind the *microarchitectural-analysis* phase in our WCET-analysis method [18], first in a conceptual way, i.e. not quite like it is implemented, later closer to how it is implemented:

- Define any architectural effect that causes an instruction to execute longer than its fastest execution time as a *Timing Accident*. Typical such timing accidents are cache misses, pipeline stalls, bus-access conflicts, or branch mis-predictions. Each timing accident is associated with a *Timing Penalty*. Timing penalties may be constant, but may also be execution-state dependent.

The property that an instruction will not cause a particular timing accident is then a safety property. The occurrence of a timing accident thus violates a corresponding safety property.

- Use an appropriate method for the verification of safety properties to prove that for the instructions in the program some of the potential timing accidents will never happen. The goal is to prove as many of such safety properties as possible. Conceptually, the safety properties shown to hold could be used to reduce the worst-case execution-time bound for an instruction, which a naive, sound WCET analysis would have to assume, by the cost for the excluded timing accidents. In practice, pipeline analysis drives a cycle-wise transition, which considers the abstract execution state, e.g. makes no transition under a cache miss if a cache miss can be excluded.
- Prove these safety properties by abstract interpretation (AI) [7] in the following way: Use AI to compute invariants at each program point, in our case an upper approximation of the set of execution states that are possible when execution reaches this program point. Derive the above mentioned safety properties, that certain timing accidents will not happen, from these invariants. For example, AI computes abstract cache states at each program point, which represent the sets of concrete cache states that may reach this program point. The abstract cache states are used to classify memory accesses at each

program point as definite hits or misses. Predicted cache hits are then used to prove that the timing accident, this memory access will miss the cache, will never happen [10].

This method for the micro-architectural analysis was the main innovation that made our WCET analysis work for real-life architectures and scale to industrial-size software [9].

Now follows the description of the microarchitectural analysis that is closer to the implementation. Driver of this analysis is the pipeline analysis [15]. It goes through the instruction stream, instruction by instruction, and executes the current instruction on the current abstract execution state. This abstract execution state contains uncertainty, i.e. misses some components. Transitions to all potential successor states are performed whenever the transition to the next state depends on such a missing part of the state. The timing contributions of these transitions are accumulated until an instruction can be retired. In the end upper bounds on the execution times of basic blocks are obtained that are coefficients in an Integer Linear Program representing the control flow of the program [18].

We currently experience two significant developments in the safety-critical embedded-systems industry that are of concern to the WCET-analysis domain, the introduction of multi-core execution platforms and the integration of applications of different criticalities on such platforms. As we will later see, the clean interface between schedulability and timing analyses becomes obsolete as soon as multi-core architectures with shared resources are used for the implementation of hard real-time systems, and the (extremely productive) scheduling community has adopted a system model, ignoring fundamentals of WCET analysis.

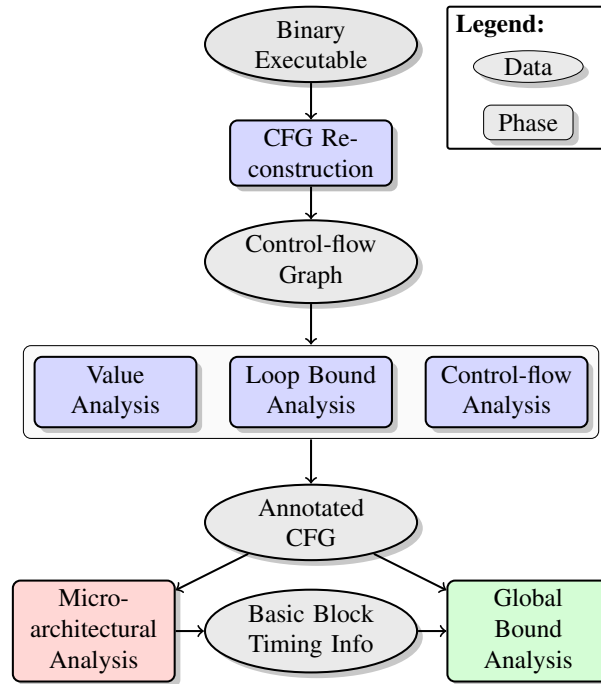
1.2 Terminology

We consider only sound WCET-analysis methods. *Soundness* means that a method and associated tool will always produce *conservative* WCET estimates, i.e. estimates that will never be exceeded in any execution. Being conservative is a Boolean property. Unfortunately, *conservative* is often used as a metric property, *more conservative* meaning *less precise*. However, calling results of an unsound method conservative is a misnomer. The really meant, other dimension, in addition to soundness, is *accuracy*. Accuracy of some WCET estimate, obtained by a sound method, expresses the degree of over-estimation, the difference between a WCET estimate and the real WCET. It does not make sense to talk about the accuracy of an unsafe estimate or an unsound method. In case of an unsound method it is not even clear whether a "more conservative" estimate moves towards the real WCET from below or is larger than the real WCET and moves further away from it.

WCET analysis can be seen as the search for a longest path in the state space spanned by the program under analysis and by the architectural platform. Most real-time software is written as to guarantee termination. Its state space can thus be easily abstracted to a finite abstract state space, which is still too large to be exhaustively explored. We can, therefore, not expect to identify the real WCET, but only safe upper bounds to all execution times, which we will call WCET estimates. (Safe) over-approximation is used in several places. In particular, an abstraction of the execution platform is employed by the WCET analysis. How to convince oneself (or the certification authorities) of the correctness of this architectural model is the subject of the next section.

2 Certification

The claim that our WCET-analysis tools produce safe results is a strong one and often disputed by some proponents of unsound WCET-analysis methods. Their argument is, to develop an error-free instantiation of the, in principle, sound WCET-analysis technology



■ **Figure 1** The architecture of the aiT tool.

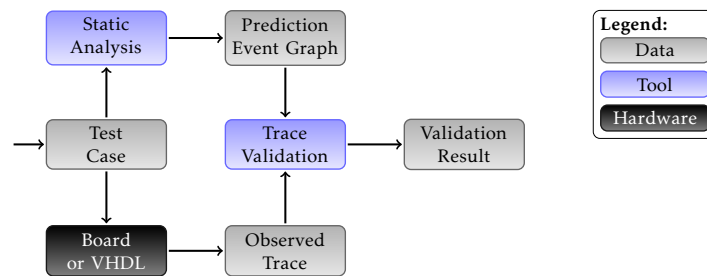
is so difficult, that one might use a simpler unsound method in the first place. The main complaint is the complexity of the abstract architectural models. So, what is the basis for our claims?

Tool Qualification according to DO178

Let us start with a description of the safety standards and the tool-qualification processes of the avionics industry, which are the most rigid of the safety-critical industries. Certification of avionics systems is regulated by the international standard DO-178C [1]. WCET-analysis tools fare under *verification tools*. Verification tools have no overly rigid certification requirements, unlike *development tools*. They require a specification of the tool functionality, from which several levels of requirements are derived. DO-178C exhales a test-based spirit. Most of the qualification is test based, requiring some coverage criteria to be observed. However, note that in case of a static verification tool, test coverage means something different from the usual interpretation as, e.g. coverage of the program control flow. At analysis time, a static-analysis tool analyses all paths and does not need coverage criteria for its analysis. It is the ISA and the set of paths through the execution platform that need to be covered. Huge sets of test traces in qualification suites are used at tool-qualification time to cover the sets of paths through the execution platform.

Certification becomes more challenging through DO-333, the Formal-Methods Supplement to DO-178C. It asks for a statement that a formal method including the underlying theory is *adequate* for solving the corresponding verification problem. This introduces and enforces *soundness* of the methods and tools.

Several component analyses in the tools are instances of abstract interpretation [7], a scientific method with a strong underlying theory, relating analysis results to semantic properties of analyzed programs. Value analysis and control-flow analysis, c.f. Figure 1



■ **Figure 2** Trace Validation according to [11]. The instruction sequences together with the generated prediction graphs, annotated by state and timing information are part of the Qualification Support Kit.

are more or less standard abstract interpretations, the difference is that these analyses are performed on the binary level and not on the source level. Still, adequacy of these analyses is easily accepted. The instantiation of the abstract-interpretation framework for the microarchitectural analysis, however, is far from trivial. In particular, each contains an abstraction of the execution platform. How does one make sure that such an abstraction is conservative? The European Aviation Safety Agency (EASA) is strict on this issue. It has accepted AbsInt’s aiT as a validated WCET-analysis tool for several time-critical subsystems in the Airbus A380 and A350 planes.

Trace Validation

EASA requires the tool user to perform a tool qualification. As written above, the most complex part of the WCET-analysis tool is the abstract architectural model. Therefore, the most complex task in tool qualification is the validation of this abstract architectural model. It is done by *Trace Validation*. The tool user may ask the tool provider to support them by providing a *Qualification Support Kit (QSK)* containing the abstract architectural model and sets of test traces, annotated with timing information. The abstract model is used as a generator of event traces. Typically, only events that can be externally observed are generated and thus contained in traces in the prediction graph. Several (hand-written) instruction sequences, *test cases* according to Figure 2, are run through this abstract model, each producing a graph of traces, the so-called *Prediction Graph* [11].

In trace validation, an instruction sequence is executed on the actual hardware. Interrupts are used to stop execution at each desired execution cycle. This way, the execution of instruction sequences are extended cycle by cycle to observe actual execution states and execution times. Whatever machine information can be read out is used. The observed trace, the reached execution state and the consumed time are checked for containment in the prediction graph. The predicted execution time may be larger than the observed execution time, but never smaller. Some interesting components of the architectural state, e.g. the cache state, are not directly observable. These need to be indirectly observed through executions that are forced to lead to cache hits and cache misses. A tremendous effort is invested to cover both all instructions and all architectural components, essentially by triggering many different initial architectural states.

In the case of the AbsInt static WCET tool, aiT, the validation suite may contain several thousand event traces, even for a simple DLX-like architecture like the ARM Cortex-M4.

Testing in the Operating Environment

In addition, DO-178 asks the user of a tool to be qualified to test the tool in their operating environment. This includes testing it on representative user code, besides testing it possibly on synthetic examples. In model-based design processes, which are quite common in the safety-critical embedded-systems domain, this is often done by exhaustively testing patterns used by the code generators.

3 Multi-Core Architectures

WCET analysis for single-core architectures is theoretically understood and practically solved. The significance of timing-predictability of execution platforms is recognized, but has left few traces in the architectural domain, a notable exception being the Kalray MPPA [8]. The transition of the embedded systems industry to multi-core platforms presented new challenges by increasing the complexity of WCET analysis considerably. In general, all possible interleavings of the concurrently executed tasks have to be analyzed, since different interleavings may lead to different execution times. The reason was is the interaction on shared resources of tasks executing on different cores [2].

The interference on shared resources of tasks running on different cores invalidates the traditional interface between WCET analysis and schedulability analysis, which is the following: WCET analysis determines an upper bound on the execution times of a task, and schedulability analysis uses this bound as input. However, different schedules on the different cores lead to different interactions on the shared resources, and in consequence, to different execution times of the tasks. So, the WCET estimate determine the schedules, and the schedules influence the execution times. This fact is ignored by quite a few people working on multi-core scheduling.

A position paper on the use of multi-core platforms in future avionics systems [6], written by an international consortium, recognizes that the interference on shared resources makes the traditional spatial and temporal partitioning methods required by ARINC 653 problematic. They require *robust partitioning* of the co-executing tasks to allow separate WCET determination. The relevant necessary condition for robust partitioning reads, *Software partitions cannot consume more than their allocations of shared resources*. This formulation, while applicable to bandwidth resources like buses, ignores the important differences between *storage resources* and *bandwidth resources*. Caches are typical storage resources. Analyzing shared caches is particularly challenging. It is clear that the cache state, and therefore also the cache-miss rate and the execution time, depend on the particular interleaving of the executions of different co-executed tasks. Buses are typical bandwidth resources. Competition for this resource is resolved by bus protocols, which then influence, for example, the memory-access time. Bus protocols become part of the WCET analysis.

In case the requirement for robust partitioning is violated the position paper asks for *mitigation* by the developer. The only problem is that it remains unclear how such mitigation could look like. [19] gives a survey of promising approaches for achieving robust partitioning.

4 Mixed Feelings about Mixed Criticality

Steve Vestal [17] has proposed a model of mixed-criticality systems for schedulability analysis. This model is based on a conjecture that *the higher the degree of assurance required that actual task execution times will never exceed the WCET parameters used for analysis, the larger and more conservative the latter values become in practice*. The survey [5] of mixed-criticality

systems by Burns and Davis adopts the same assumption, *A key aspect of MCS is that system parameters, such as tasks' worst-case execution times (WCETs), become dependent on the criticality level of the tasks.* These authors, however, seem to be skeptical about this assumption: *Although it is reasonable to assume confidence increases (i.e. uncertainty decreases) with larger estimates of worst-case execution time, this may not be universally true. It would certainly be hard to estimate what increase in confidence would result from, say, a 10% increase in all Cs.* It is illuminating that both articles do not mention soundness, as if it were of no concern in the safety-critical systems domain.

I see no inherent reason why higher criticality should entail higher WCET estimates. It seems that this assumption is intuitively based on the assumption that WCET estimates are determined by measurement; higher criticality levels require higher assurance, and higher assurance is achieved by performing increasing sets of tests. Since WCETs monotonically increase with increasing the set of tests – an observed WCET does not disappear, when more tests are added – more tests can, in fact, not produce lower WCET estimates.

4.1 Exploitation of Hardware Resources

Another motivation is an assumed higher exploitation of the hardware performance: Let us assume that by extensive measurement the Maximum Observed Execution Time (MOET) of the high-criticality task T_h is found to be substantially less than the WCET estimate, $C_h(HI)$ provided by a sound tool. This MOET may be considered as an intermediate (low-criticality) budget, $C_h(LO)$. Some low criticality software, T_l , with no strong guarantees, which will be run on the same hardware platform, might have a MOET of $C_l(LO)$. The scheduler may drop or degrade the low-criticality task in the event that either T_l exceeds its MOET of $C_l(LO)$ or T_h exceeds its MOET of $C_h(LO)$. Since the high-criticality task must execute and must meet its timing constraints, the overall performance required of the system is given by $\max(C_l(LO) + C_h(LO), C_h(HI))$, which may be substantially less than $C_l(LO) + C_h(HI)$. The strength of this motivation, of course, depends on the size of $C_h(HI) - C_h(LO)$, i.e. on the amount of over-estimation. There are a few publications documenting the amount of over-estimation, see [16]. Between 15 and 25 % over-estimation were observed on real Airbus code. Of course, the amount of over-estimation depends on many factors, in particular on the timing predictability of the execution platform [12, 13, 4].

4.2 Schedulability Analysis

Vestal thus starts with the assumption that different WCETs are associated with different criticality levels of a task, the higher the criticality level, the higher the WCET estimate, and then proposes to use two different versions of preemptive fixed-priority (PFP) scheduling with deadline-monotonic priority assignment; tasks with smaller deadlines get higher priority.

The first approach attempts to solve the problem that low-criticality tasks with shorter deadlines than higher-criticality tasks would receive higher priorities. By cutting the longer execution times of higher-criticality tasks into short time slices they receive higher priorities. The scheduling algorithm then is able to use time left over by higher-criticality tasks not exhausting their WCET estimate for lower-criticality tasks. So far so good! No treatment is dedicated to the case that the high-criticality task exceeds its WCET estimate and in consequence its deadline. This is possible since measurement-based analyses are not guaranteed to produce safe upper bounds.

Vestal's second approach uses Audsley's priority-assignment algorithm [3] in a setting with WCET bounds increasing with criticality level, for which it was not originally described. I assume that the algorithm can be adapted to work for a setting with different WCET estimates associated with different criticalities.

4.3 Consequences of Ignoring Sound Approaches

So, Vestal's schedulability test is only sound with respect to correct WCET estimates and will, if given incorrect estimates, accept task sets whose high-criticality tasks may at run time exceed their deadlines. The scheduling community on the one hand would claim that their algorithms are sound, but gladly accepts input produced by unsound methods, which invalidates the overall correctness claim. However, his model has been and still is the underlying model for most scheduling research on mixed-criticality systems. [5] lists almost 200 publications.

References

- 1 RTCA/DO-178C Software Considerations in Airborne Systems and Equipment Certification, 2013.
- 2 Andreas Abel, Florian Benz, Johannes Doerfert, Barbara Dörr, Sebastian Hahn, Florian Hauptenthal, Michael Jacobs, Amir H. Moin, Jan Reineke, Bernhard Schommer, and Reinhard Wilhelm. Impact of Resource Sharing on Performance and Performance Prediction: A Survey. In Pedro R. D'Argenio and Hernán C. Melgratti, editors, *CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, volume 8052 of *Lecture Notes in Computer Science*, pages 25–43. Springer, 2013. doi:10.1007/978-3-642-40184-8_3.
- 3 Neil Audsley. Optimal Priority Assignment and Feasibility of Static Priority Tasks with Arbitrary Start Times. Technical Report 164, Department of Computer Science, University of York, November 1991.
- 4 Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard von Hanxleden, Reinhard Wilhelm, and Wang Yi. Building timing predictable embedded systems. *ACM Trans. Embedded Comput. Syst.*, 13(4):82:1–82:37, 2014. doi:10.1145/2560033.
- 5 A. Burns and R. Davis. Mixed criticality systems-a review. Technical report, Department of Computer Science, University of York, 2013.
- 6 Certification Authorities Software Team. *Multi/core processors*, CAST-32A edition, November 2016. position paper.
- 7 Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. doi:10.1145/512950.512973.
- 8 Benoît Dupont de Dinechin, Duco van Amstel, Marc Poulhiès, and Guillaume Lager. Time-critical computing on a single-chip massively parallel processor. In Gerhard Fettweis and Wolfgang Nebel, editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, pages 1–6. European Design and Automation Association, 2014. doi:10.7873/DATE.2014.110.
- 9 C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *EMSOFT*, volume 2211 of *LNCS*, pages 469–485, 2001.
- 10 Christian Ferdinand and Reinhard Wilhelm. Efficient and Precise Cache Behavior Prediction for Real-Time Systems. *Real-Time Systems*, 17(2-3):131–181, 1999.
- 11 Gernot Gebhard. *Static timing analysis tool validation in the presence of timing anomalies*. PhD thesis, Saarland University, 2013. URL: <http://scidok.sulb.uni-saarland.de/volltexte/2013/5558/>.

- 12 Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *IEEE Proceedings on Real-Time Systems*, 91(7):1038–1054, 2003.
- 13 Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007. doi:10.1007/s11241-007-9032-3.
- 14 Alan C. Shaw. Deterministic timing schema for parallel programs. In V. K. Prasanna Kumar, editor, *The Fifth International Parallel Processing Symposium, Proceedings, Anaheim, California, USA, April 30 - May 2, 1991.*, pages 56–63. IEEE Computer Society, 1991. doi:10.1109/IPPS.1991.153757.
- 15 Stephan Thesing. *Safe and Precise WCET Determinations by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.
- 16 Stephan Thesing, Jean Souyris, Reinhold Heckmann, Famantanantsoa Randimbivololona, Marc Langenbach, Reinhard Wilhelm, and Christian Ferdinand. An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics Software. In *2003 International Conference on Dependable Systems and Networks (DSN 2003), 22-25 June 2003, San Francisco, CA, USA, Proceedings*, pages 625–632. IEEE Computer Society, 2003. doi:10.1109/DSN.2003.1209972.
- 17 Steve Vestal. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *Proceedings of the 28th IEEE Real-Time Systems Symposium (RTSS 2007), 3-6 December 2007, Tucson, Arizona, USA*, pages 239–243. IEEE Computer Society, 2007. doi:10.1109/RTSS.2007.47.
- 18 Reinhard Wilhelm, Sebastian Altmeyer, Claire Burguière, Daniel Grund, Jörg Herter, Jan Reineke, Björn Wachter, and Stephan Wilhelm. Static Timing Analysis for Hard Real-Time Systems. In Gilles Barthe and Manuel V. Hermenegildo, editors, *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings*, volume 5944 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2010. doi:10.1007/978-3-642-11319-2_3.
- 19 Reinhard Wilhelm, Jan Reineke, and Sven Wegener. Keeping Up with Real Time. In Umut Durak, Juergen Becker, Sven Hartmann, and Nikolaos S. Voros, editors, *Advances in Aeronautical Informatics: Technologies Towards Flight 4.0*. Springer, 2018.

Formal Executable Models for Automatic Detection of Timing Anomalies

Mihail Asavaoe

CEA LIST
Gif-sur-Yvette, France
mihail.asavaoe@cea.fr

Belgacem Ben Hedia

CEA LIST
Gif-sur-Yvette, France
belgacem.ben-hedia@cea.fr

Mathieu Jan

CEA LIST
Gif-sur-Yvette, France
mathieu.jan@cea.fr

Abstract

A timing anomaly is a counterintuitive timing behavior in the sense that a local fast execution slows down an overall global execution. The presence of such behaviors is inconvenient for the WCET analysis which requires, via abstractions, a certain monotony property to compute safe bounds. In this paper we explore how to systematically execute a previously proposed formal definition of timing anomalies. We ground our work on formal designs of architecture models upon which we employ guided model checking techniques. Our goal is towards the automatic detection of timing anomalies in given computer architecture designs.

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

Keywords and phrases timing anomalies, predictability, formal methods, model checking

Digital Object Identifier 10.4230/OASICS.WCET.2018.2

Acknowledgements The authors would like to thank Simon Wegener from AbsInt GmbH for providing valuable feedback on this work.

1 Introduction

Modern computer architectures are designed to alleviate the bottleneck between processors, and memory systems, leading to utilization of caches, pipelines and speculation mechanisms. Such architectures are often used in embedded system design and hence required to satisfy, *a posteriori*, stringent timing behavior. The alternative is to design predictable systems, which focus on building systems with *a priori* guarantees of timing requirements.

The quest for predictability is a complicated endeavor as all components to build and execute a system, such as processors, high/low-level languages, compilers, operating systems, communication systems, etc., can impact the definition and verification of timing requirements. Designs of predictable systems as well as associated timing analyses identify and circumvent the sources of non-predictability in various ways: disabling the problematic component(s) (e.g., a particular shared resource), proposing timewise restrictions (e.g., semantics based on temporal isolation), using predictable components (e.g., LRU caches) or even straightforwardly



© Mihail Asavaoe, Belgacem Ben Hedia, and Mathieu Jan;
licensed under Creative Commons License CC-BY

18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018).

Editor: Florian Brandner; Article No. 2; pp. 2:1–2:13

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

assuming the predictability. Timing analyses need to overcome various challenges: in single-cores, the worst-case execution time (WCET) analysis is complicated by the presence of timing anomalies [11] whereas in multi-cores, the worst-case response time (WCRT) analysis is hampered by timing compositionality issues [5].

The WCET analysis computes sound and (desirably) tight worst-case execution bounds, exploring, via convenient abstractions, all the execution paths of a program running on a computer architecture. Typically, the WCET analysis works on the control-flow graph of the binary code, augmented with semantic information from both the code (e.g., loop bounds) and the underlying architecture (e.g., cache hits/misses). The WCET analysis workflow integrates the results of cache analyses with accurate pipeline modeling to safely search for the longest execution path. This searching process is complicated by the presence of timing anomalies as these are non-monotonic behaviors. In essence, a timing anomaly is a counterintuitive behavior in the sense that the local worst-case timing behavior does not result in the global worst-case performance.

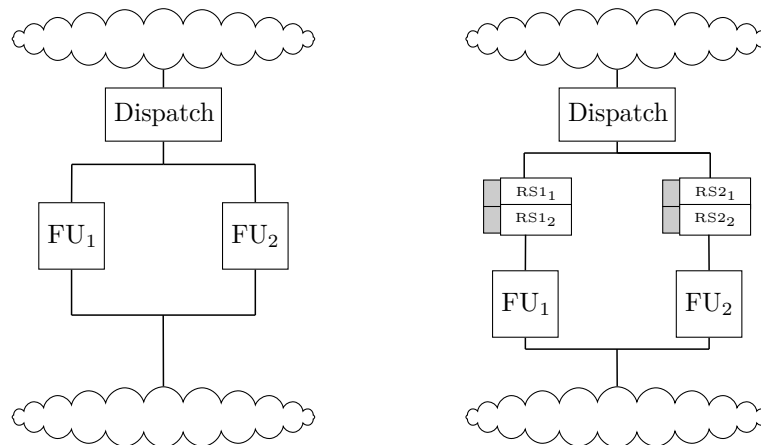
Formal-based methods, e.g., static analysis or model checking, soundly explore all system behaviors, while mitigating precision and performance arguments. Formal reasoning could either evaluate predictability issues of existing systems or guide the construction of predictable systems. Hence the formal, systematic study of timing anomalies becomes essential. In this direction, the first formal definition of timing anomaly is proposed in [14]. *The contribution of this paper is to execute this formal definition of a timing anomaly, based on model-checking, towards the automatic detection of timing anomalies.* Our method consists of three phases. First, we consider formal executable models of particular computer architectures, specified using the TLA+ language [8]. These models are deterministic and tested for conformance against actual system behaviors. Second, we systematically enable, directly over the concrete models, non-deterministic choices (i.e., abstract behaviors) as the necessary conditions to facilitate the study of timing anomalies. Finally, we employ model checking, using the TLC tool [18] for TLA+ models, to explore the execution paths of the abstract specification. While our method for automatic detection of timing anomalies is general, our current investigation is in its incipient stages. However, we evaluate our technique on standard examples of scheduling timing anomalies while using models of resource contention in superscalar processors.

We organize this paper as follows. In Section 5, we review some related work and in Section 2, the formal definition of timing anomalies. In Section 3 we briefly introduce the TLA+ language and present our formal architecture models. In Section 4 we describe the automatic detection of timing anomalies. We conclude and outline future work in Section 6.

2 Timing Anomalies – Definition and Examples

Essentially, the first formal definition of timing anomalies, in [14], encodes *an abstract state space*, constructed with respect to both an architecture and an input program and *a property pattern*, expressed with respect to a locality concept. Our proposed method *executes* this formal definition, towards an automatic technique for the detection of timing anomalies. Next, we introduce the ingredients: the running examples and the necessary steps to formalize the timing anomalies.

We consider as running examples the cases of scheduling timing anomalies introduced in [16]. The goal is to study policies of resource allocation (e.g., functional units) in superscalar architectures. Two snapshots of the execution stage of superscalar processors are shown in Figure 1. On the left side, the resources FU_1 and FU_2 execute instructions in the program order, while on the right side, the reservation stations $RS_{1,2}$ and $RS_{2,1,2}$ allow out-of-order



■ **Figure 1** Snapshot of superscalar processor with: (left) in-order resource allocation for both FUs and (right), respectively out-of-order resource allocation for both FUs.

execution on both FU_1 and FU_2 . On these platforms we execute program paths of size 4 (i.e., instructions A to D with the alphabetical order giving the program order), under certain allocation constraints, as in Figure 2.

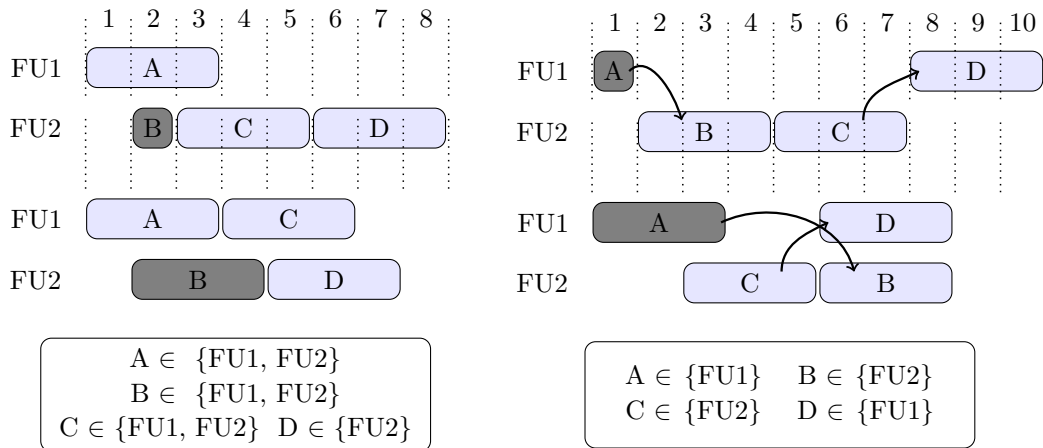
The architecture model in Figure 1 (left) is without reservation stations and the resource allocation is dynamically decided for instructions like A, based on resource availability. The resource FU_1 is the default execution unit for such instructions in the case when both FUs are available. Consequently, the program execution is guided by the program order. The architecture model in Figure 1 (right) imposes different constraints on the set of instructions with respect to resources. As supported by the constraints in Figure 2 (right), each instruction could be executed on a single type of resource. The resources FU_1 and FU_2 can also execute instructions in out-of-order fashion, based on the content of their respective reservation stations. Consequently, the program execution is guided by the data dependencies between instructions. In our example, the instructions B and C are independent and could be executed in any order, whereas instructions A and B are always executed in the program order.

Examples of scheduling timing anomalies for the architecture models with in-order and out-of-order resource allocation are shown in Figure 2 (left) and respectively (right). In both cases, a pivot instruction with variable latency causes a timing anomaly. For example, a faster execution of instruction B frees FU_2 for the execution of instruction C. It further delays the instruction D whose execution is conditioned by the availability of the same resource FU_2 , in Figure 2 (left).

The formal definition of timing anomalies requires the following concepts:

- (1) an (*abstract*) *architecture model* to provide the settings of the execution environment;
- (2) a notion of *locality* to express local worst-case behaviors;
- (3) a path mapping as a *labeling function* to correlate the program with the architecture.

Each of the three points requires specific assumptions. For example, the key ingredient towards the construction of a convenient architecture model - point (1) is to enable non-deterministic choices as a standard method to compactly encode system behaviors. The code and the related input data are also part of the system (abstract) state. Point (2) defines the locality as the sequence of abstract system states which satisfies particular constraints with respect to system behaviors. For example, a system execution path is studied locally – between two points of interest – (e.g., when instruction A is in a particular pipeline stage)



■ **Figure 2** Example of scheduling timing anomalies from [16], with out-of-order (left) and in-order (right) resource allocation, under given allocation constraints for instructions A to D.

with respect to locality constraints (e.g., the interaction between A and all other possible combinations of instructions). Lastly, point (3) is necessary to interpret the search for timing anomalies on the specified architecture system. It relates the instruction-level view given by the program paths with the cycle-level view of the architecture execution paths.

Our approach towards the automatic detection of timing anomalies encodes this formalization. Briefly, we address point (1) when we define, using the TLA+ specification language, a cycle-accurate computer architecture specification; we refer to it as the concrete architecture model. Furthermore, we abstract this concrete model as we encode the necessary non-deterministic choices; we refer to the new specification as the abstract architecture model. Then, we address point (2) when we consider the locality as defined by a particular pipeline stage, hence the locality is a priori encoded by our abstract/concrete architecture state. The locality constraints are either directly represented in the model (as constraints on the input data/program) or computed during the exploration of the state space. Finally, we directly insert the labeling function, i.e., point (3) in the abstract architecture model, more specifically in the code component of the abstract model state. We elaborate next on all these points.

3 Design of Formal Executable Models

Our modeling for automatic detection of timing anomalies fully adheres to the formalization steps (1) – (3), which are required by the definition of timing anomalies from [14]. Our concrete and abstract models are TLA+ specifications.

We choose the TLA+ modeling language because of several semantic considerations. TLA+ features an advanced module system based on interfaces, parameters, local declarations etc. which allow accurate construction of (concrete and abstract) architecture models from simpler components. The modeling language also features untyped set theory (and predicate logic) to specify rich state information. Abstraction in TLA+ is ensured by temporal existential quantification which hides unnecessary state elements. Refinement in TLA+ is ensured by supporting stuttering invariance (i.e., execution steps that do not change the values of state variables of interest) which allows reasoning about system paths on different levels of granularity. All the aforementioned concepts establish TLA+ as an unified logical

language designed to specify both systems and their properties, as well as verifying, using the same specification, both a system and its possible refinements. This latter characteristics of TLA+ is particularly attractive for our investigation towards automatic detection of timing anomalies as our framework is based on a single formal specification (i.e., of the concrete hardware model), which is then systematically refined. We recommend [12] for an in-depth and comprehensive survey of the TLA+ language semantics and its applications. Next, we introduce several elements of the TLA+ language and we exemplify their usage with snapshots of our formal models.

A TLA+ specification is two-tiered. The first level contains state and state transition formulas (i.e., system specification) and the second level contains temporal formulas evaluated on sequences of states (i.e., system properties). A particularity of the TLA+ language is the transition predicate (also called *action*) which establishes a relation between variable values in the current and next states. For example, if x is a state variable, the action $x' = x + 1$ means that the next value of x (the primed variant) is the current value of x (the unprimed variant) incremented by 1. Whenever state elements are unmodified by a transition, for example $x' = x$, the TLA+ notation is *UNCHANGED* x . If x is a record with two fields *fst* and *snd*, an individual field is accessed with “.”, for example $x.fst$. As such, the TLA+ action $x' = [x \text{ EXCEPT } !.fst = 1]$ means that only the value of *fst* of x is modified in the next state. When a TLA+ module X with an internal state variable x and a transition Act is used in another module, the operator “!” gives access to each, e.g., $X!x$ and respectively $X!Act$. Finally, we denote by $\langle S \rangle$ the state configuration of an TLA+ specification $Spec$ (i.e., S is the set of semantic entities that are necessary to define the behaviors of $Spec$).

(1) The hardware model – concrete

We define the two instances of superscalar architectures from [16] and for each instance we define a concrete model which is then systematically transformed into an abstract model. The formal computer architecture model is developed in a modular fashion, according to the principles described in [9], using the TLA+ module system.

The state configuration \mathcal{C} of our concrete architecture model consists of two state components: the architecture $Arch$ and the input program $Code$.

$$\mathcal{C} = \langle Arch, Code \rangle .$$

The concrete $Arch$ consists of several variables to represent the pipeline stages; these variables are updated cycle-wise based on the content of their inner states and the necessary signals from the memory system, as in [16]. Since we aim for the detection of scheduling timing anomalies, we implicitly represent the signals from the memory system, while fully specifying the execution pipeline stage and an instruction progress through the pipeline. The $Arch$ state configuration for the architecture model in Figure 1 (left) is that of a standard 5-stage pipeline:

$$Arch = \langle _IF, _ID, _EX, _MEM, _WB \rangle .$$

whereas for the architecture model in Figure 1 (right) is a 6-stage pipeline, with an extra instruction issue stage. The names for the pipeline stages stand for instruction fetch ($_IF$), instruction decode ($_ID$), execute ($_EX$), memory access ($_MEM$) and write-back ($_WB$).

Both pipeline models are dual-issue. Structurally, our architecture models are incrementally built from simple parameterized modules of buffers and functional units, which are instantiated into pipeline stages. Each functional unit and internal buffers of the pipeline

$$\begin{aligned}
& \mathbf{AcquireFU1} \triangleq \\
& \quad \wedge \mathit{condAcquireFU1} \\
(1) \quad & \boxed{
\begin{aligned}
& \wedge \text{IF } \mathit{isCurrIns} \\
& \quad \text{THEN } _IF' = [_IF \text{ EXCEPT } !.\mathit{buff} = \mathit{FBUFF!Set}(\mathit{code.currIns})] \\
& \quad \text{ELSE } _IF' = [_IF \text{ EXCEPT } !.\mathit{buff} = \mathit{FBUFF!Reset}] \\
& \wedge _ID' = \mathit{updateID}(_ID) \\
& \wedge _EX' = [_EX \text{ EXCEPT } !.\mathit{fu1} = \mathit{FU1!Acquire}(_ID.\mathit{buff.instr})] \\
& \wedge _MEM' = \mathit{updateMEM}(_MEM) \\
& \wedge _WB' = \mathit{updateWB}(_WB) \\
& \wedge _code' = \mathit{updateCode}(\mathit{code}) \\
& \wedge \mathit{cycle}' = \mathit{updateClk}(\mathit{cycle})
\end{aligned}
} \\
(2) \quad & \boxed{
\begin{aligned}
& \wedge \text{IF } \mathit{isCurrIns} \\
& \quad \text{THEN } \exists \mathbf{d} \in \mathbf{code.currInstr.tvar}: \\
& \quad \quad _IF' = [_IF \text{ EXCEPT } !.\mathit{buff} = \mathit{FBUFF!Set}(\mathit{code.currIns}, \mathbf{d})] \\
& \quad \text{ELSE } _IF' = [_IF \text{ EXCEPT } !.\mathit{buff} = \mathit{FBUFF!Reset}]
\end{aligned}
}
\end{aligned}$$

■ **Figure 3** The TLA+ rule for acquiring the functional unit FU1. With (1), the rule presents the concrete architecture behavior. When (1) is replaced by (2), the rule shows the abstract architecture behavior when exploiting timing variations for the current instruction.

stages provide an interface for their operations, accessible via the “!” operators. Semantically, our architecture model for the out-of-order resource allocation supports the Tomasulo algorithm, as in [1], whereas the in-order resource allocation is driven by the program order.

Let us briefly explain our concrete architecture model using an excerpt of the TLA+ formal model, in Figure 3. We recall that our objective is to study scheduling timing anomalies which manifest when instructions are deployed for functional units in the *execute* stage of the pipeline. This scheduling mechanism consists of operations of acquire and/or release of one or both functional units (i.e., in short *FUs*). Figure 3 presents the specification of acquiring the functional unit FU1, a rule named **AcquireFU1**. Other TLA+ rules specify pipeline stalls, flushes, simultaneous acquires of both FUs, etc. Each rule is guarded by a predicate (e.g., $\mathit{condAcquireFU1}$) and contains the actions to update the *Arch* and *Code* (i.e., variable $_code$) parts of the concrete configuration, as well as the clock variable (i.e., cycle). In our example, the guard $\mathit{condAcquireFU1}$ is a predicate which establishes the necessary conditions to activate the rule **AcquireFU1**:

$$\begin{aligned}
\mathbf{condAcquireFU1} \triangleq & \wedge \neg \mathit{emptyID} \wedge \mathit{isAvaiFU}(_ID.\mathit{buff.instr}, \mathit{FU1!fname}) \\
& \wedge (\mathit{emptyEX} \vee (\mathit{emptyFU1} \wedge \mathit{FU2!inExec}(_EX.\mathit{fu2})))
\end{aligned}$$

The first line ensures that there is an instruction in the decode stage which is ready and needs to be executed by FU1 as $\mathit{isAvaiFU}$ checks whether instruction instr from the decode stage can be executed over the functional unit FU1. The second line ensures that there is not another case of acquire or release of either FUs at the same time.

When $\mathit{condAcquireFU1}$ is true, the new content of the instruction stage (emphasized by (1)), $_IF'$, retrieves a new instruction, if it exists (variable $\mathit{isCurrIns}$), and sets the internal state of this stage (using the accessor “ buff ”) to this instruction. If a new instruction is not available, the new internal state of the instruction stage is reset, i.e. it is emptied, using the operation *Reset*. The new content of the execute stage, $_EX'$, is modified only for the first functional unit (using the accessor “ $\mathit{fu1}$ ”) with an instruction from the decode stage

(i.e., $_ID.buff.instr$). In a similar way, the other pipeline stages (decode, memory access and write-back) and the code update their next state, via the corresponding *update* functions. Finally, the clock cycle advances using the *updateClk* function.

Let us recall that the input program is represented in the concrete state configuration \mathcal{C} by the state component *Code*. In our TLA+ models, the program is represented by its set of program paths and each path is a sequence of instructions. An instruction is represented by several parameters: the program counter, the execution resources (as the set of necessary FUs), the latencies (the concrete representation considers exactly one latency per instruction), the dependencies with respect to other instructions and finally the temporal availability (in this latter case, it is borrowed from the task-oriented model of computation). Whereas our instruction representation does not model a particular instruction set architecture (ISA), it contains all the necessary semantic ingredients to capture existing ISAs semantics. Figure 2 shows examples of instructions respecting the properties of our instruction model.

The concrete architecture models are cycle-accurate and deterministic. The part *Code* of \mathcal{C} is instantiated with concrete program paths and executed using the TLC model checker. We rely on the TLC statistics on the state space to assess the determinism aspect of our architecture models and to drive, whenever necessary, model refinements. We extensively test both concrete models to gain confidence in their correct functionality and determinism. The abstract models are constructed directly over the concrete models, e.g., replacing predicate (1) with (2) in Figure 3. We detail this procedure in the next section.

(2) The locality concept

It is accepted [11, 14] that locality matches an instruction progress through the pipeline stages. The notion of locality is thus formalized as a path fragment of interest, for any execution path in the model. The particular example of scheduling timing anomalies, which appear in processors due to contention for functional units defines the locality level as the execution pipeline stage, i.e. the $_EX$ stage in our pipeline models.

The locality constraints are convex predicates which hold locally – on path fragments of interest. They could be (a) pre-determined and encoded in the program part of the state configuration, e.g., in our case in *Code*, or (b) dynamically calculated during the model execution. We experiment with both variants and henceforward and without the loss of generality, our locality constraints are given, i.e., we assume (a). Precisely, we work with convex predicates in the form of single linear inequalities where an instruction latency is bounded by a pre-computed value. For example, in Figure 2 (left), the execution time for B is bounded by 1 for the first execution and by 3 for the second execution.

(3) The labeling function

We use the *Arch* configuration to construct cycle-accurate architecture models. It is necessary to relate them to the instruction-level information presented in the *Code* configuration. We address this aspect directly in the concrete architecture model, as our method is centered around the program path. Hence, *Code* encodes all the program paths [10] which are evaluated path by path. In the general form, our code-related configuration is:

$$Code = \langle [Paths], CurrPath \rangle.$$

with the input program and data in $[Paths]$ and the current program path in *CurrPath*. Since the study of timing anomalies require input variations at the path level, we assume, without loss of generality, that a simplified *Code* contains only *CurrPath*. Structurally, a

program path is encoded as a list of instructions. Semantically, each instruction advances in the program order, given by the program counter, through the pipeline stages until *Execute*, where a corresponding resource allocation takes place. TLA+ facilitates a flexible encoding of path-related variations with its set-theoretic semantics. For example, the latency 3 of instruction B from Figure 2 is adequately represented in *CurrPath* by a singleton.

The design of concrete architecture models follows the principles of the formal definition of timing anomalies: the architecture is deterministic and cycle accurate, the code is part of the model; the program paths are evaluated one by one etc. Over such infrastructures, we systematically construct abstract models which are checked for timing anomalies. We present next how we perform abstractions over the concrete model and how we use model checking for the detection of timing anomalies.

4 Detection of Timing Anomalies

Generally, a TLA+ specification *Spec* consists of the definition of the initial state *Init* and a state transformer *Trans* applied over the state variables, e.g., in our case \mathcal{C} :

$$Spec == Init \wedge \Box Trans_{\mathcal{C}}.$$

where \Box is the temporal operator “always”. *Trans* contains guarded transitions for pipeline stalls, flushes, single acquire of FU₁ or FU₂, simultaneous acquires of both FUs, simultaneous acquire of FU₁ and release of FU₂ etc.

(1) The hardware model – abstract

We construct an abstract architecture model which augments the concrete model *Spec* with non-deterministic choices. The abstraction creates “diamonds” in the specification which are to be explored with the model checker. The abstract state configuration, \mathcal{A} refines \mathcal{C} in both architecture *AArch* and program *ACode* components:

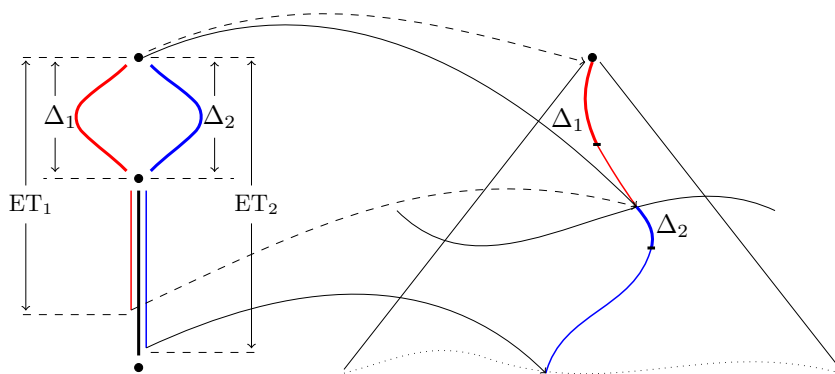
$$\mathcal{A} = \langle AArch, ACode \rangle.$$

For example, variable latency $\{1, 3\}$ of instruction B in Figure 2 (left) form a diamond in the abstract architecture model. Similar variations lead to have *CurrPath* of the *ACode* configuration encoding sets of concrete paths. The state transformer *Trans'* associated to *AArch* extends its concrete counterpart based on *Arch* to fully explore these sets of paths. For example, for the aforementioned instruction B, the latency is non-deterministically chosen between 1 and 3, when applicable (i.e., in certain states of interest). The model checking explores both possibilities of the new abstract architecture model – *Spec'*:

$$Spec' == Init \wedge \Box Trans'_{\mathcal{A}}.$$

The abstract architecture model includes the path-level variations, as presented in Figure 3 on rule **AcquireFU1** where predicate (2) replaces (1). This particular rule shows two important aspects of our abstract model: it is constructed directly over the concrete model and the abstraction points – the “diamonds” – are visible in the specification. This latter point opens the possibility of exploring the diamonds in a guided way, which establishes the third step of our systematic framework for automatic detection of timing anomalies.

A timing anomaly is characterized by a pair of execution paths because it “compares” local worst-case variations with respect to global worst-cases. For example, let us consider two execution paths, as in Figure 4 (left), where local variations Δ_1 and respectively Δ_2 ,



■ **Figure 4** Abstraction diamond (left) and timing anomaly on the search tree (right).

with $\Delta_1 > \Delta_2$ result in global execution times ET_1 and respectively ET_2 , with $ET_1 < ET_2$. Intuitively, automated detection of timing anomalies over the abstract model $Spec'$ means searching for such pairs of execution paths with counter-intuitive behavior. Now, it remains to encode this property in TLA+ and launch the TLC model checker to search for timing anomalies. A simple way to encode this property is as an invariant of the form:

$$Prop_{TA} = \square \neg (\Delta_1 > \Delta_2 \wedge ET_1 < ET_2)$$

and the property is checked on $Spec'$.

We accommodate such a formulation over a transformed search tree, as in Figure 4 (right), where, intuitively, each path consists of two different paths of the original encoding of the search space. More simply, a diamond is fully unfolded along a single path, while respecting the initial conditions of its cases. For example, the two execution paths in Figure 2 (left) form a single execution path in the new search tree, with $\Delta_1 = 1$, $\Delta_2 = 3$, $ET_1 = 8$ and $ET_2 = 7$. This new search tree is constructed on-the-fly and the detection procedure stops when the first “long” path which violates the property $Prop_{TA}$ is found. Intuitively, the diamond unfolding corresponds to a simple observer automaton which toggles between two states (e.g., with a set/reset-like semantics).

We guide the model checker to find “long” paths, implementing a mechanism to track the exploration of all diamonds. We opt to encode this mechanism directly in the abstract model (it can also be automatically generated for a given abstract model). As such, we extend the abstract configuration \mathcal{A} to accommodate the guiding mechanism:

$$\mathcal{A}_{state} = \langle AArch, ACode, Guide \rangle.$$

In its simplest form, *Guide* monitors the execution, records taken decisions and direct subsequent executions to the unexplored state space. Precisely, our *Guide* encodes how to construct long paths and then how to fully explore the new state space. We address the first point using a single TLA+ rule which is activated only when the first (red) execution in Figure 4 (right) terminates and under the same initial conditions, the second (blue) execution starts. With respect to the second point, our current implementation supports a rudimentary, though systematic, exploration of all diamonds in our abstract model. For example, if the set of timing variations of a particular instruction is $\{2, 4, 5\}$, *Guide* explores (in this order) the diamonds $\{2, 4\}$, $\{2, 5\}$ and $\{4, 5\}$. Variations on multiple instructions are handled in

a similar fashion with the order of diamonds also depending on the instruction program counters. More refined heuristics to speed-up the model checking (e.g., with interpolation techniques as in [6]) are left for future work.

We detect (scheduling) timing anomalies like those in Figure 2, from [16]. We experiment with small scale architecture models, in total around 2K lines of TLA+ specification, upon which we execute both concrete and abstract program paths. At the architecture level, we consider two dual-issue pipelines with 5 stages for the in-order functional unit allocation and respectively with 6 stages for the out-of-order variant, with precise modeling of the instruction advancement in the pipelines and with complete specification of resource contention in the execution stages. We also perform preliminary experimentation with variants of the in-order architecture models based on pipeline stalls, as indicated in [4]. At the program level, we consider program paths which activate the worst-case contention scenarios for the model in Figure 1 (right), when all reservation stations and functional units are full. Next, we elaborate on some experimentation, conducted on a quad-core Intel i7 at 2.8GHz with 16GB RAM and with the TLA+ Toolbox using the TLC model checker version 2.19.

Let us exemplify with the following test scenario, named \mathcal{T} , upon which we construct several test variants. At the architecture level, we consider the 5-stage pipeline with in-order functional unit allocation. At the code level, we use a program path of size 20, with multiple variations for instruction latencies and resource allocations (actual statistics on the size of the both feasible and infeasible search space are subsequently given). We investigated several aspects of our approach: (a) the concrete executions are deterministic, (b) the absence of timing anomalies in \mathcal{T} and finally (c) the detection of timing anomalies in methodically-constructed variants of \mathcal{T} , using the guide mechanism. The TLC model checker provides several statistics on the search space, notably the problem diameter, the number of existing states and the number of distinct states. Our extensive evaluation of (a), on concrete executions (i.e., the instruction latencies are given as singletons) of \mathcal{T} end, after 2-3 seconds, with identical numbers on all these parameters. The absence of timing anomalies (b) requires full exploration of the state space of \mathcal{T} . As such, we employ bounded model checking (with a bound value of 100) and prove that \mathcal{T} does not have timing anomalies in approximately 7 hours and with a maximum memory consumption of 39GB. The statistics on the entire state space of \mathcal{T} include 839M states found with 835M distinct states (i.e., around 0.5% duplicated states). Finally, we address (c) the detection of timing anomalies in \mathcal{T} , using *Guide*. We produce several variants of \mathcal{T} , “inserting” timing anomalies (as variations of instruction latencies) into the test scenario. For example, small timing variations (i.e., $|\Delta_1 - \Delta_2| \leq 2$ cycles), at various path locations (i.e., program counters of 5, 14 and 20) cause timing anomalies with *ETs* variations of up to 20 cycles. The timing anomalies are found as “long” paths in the search tree of Figure 4 (right) using bounded model checking with the bound value of 1000. The running time varies between 1-2 minutes, with around 10M states covered. We also experimented with variants of \mathcal{T} with well-concealed timing anomalies, yielding a running time time of around 1 hour and up to 200M explored states.

We address next some advantages and weaknesses of our detection algorithm. We present a general method, which it is not restricted to scheduling timing anomalies, as exemplified here. Because our approach is constructed over a concrete architecture model, it is possible to subject the detection of timing anomalies to guided, but non-exhaustive, heuristics (as they were firstly observed in [11]). Also, our detection procedure could be tuned to compute the local variations (deltas) of [13]. Finally, our formal architecture models could be adapted to experiment with newly proposed and/or predictability-driven architecture modifications [4]. On the other hand, our approach relies on two daunting tasks: the construction of the formal

infrastructure and the handling of the state space explosion. While we discussed possible approaches towards the latter, building a cycle accurate formal executable architecture remains complicated. A possible solution is to use already constructed formal models, another is to automatically extract them from existing HDL designs (as suggested in [1, 3]).

5 Related Works

The first assessment of timing anomalies in the context of the WCET analysis is presented in [11]. It reports timing anomalies caused by caches (and identified in [14] as speculation timing anomalies) in out-of-order architectures. The first formal definition of timing anomalies is proposed in [14]. We elaborate on its technical aspects in Section 2 as it establishes the foundation of our work. A class of timing anomalies (and identified in [14] as scheduling timing anomalies) is studied in [16] on two superscalar models with in-order and respectively out-of-order resource allocation. The work in [2] presents an actual computer architecture – the LEON2 processor – with timing anomalies (i.e., speculation timing anomalies).

Our approach shares similarities with two other approaches [1, 3] on the formal investigation of timing anomalies. Briefly, the work in [1] focuses on proving the absence of timing anomalies using bounded model checking, whereas the work in [3] combines static analysis with measurement-based techniques towards detection of timing anomalies. Both approaches, as well as ours, follow a similar pattern – the construction of a convenient representation of the abstract architecture state space and work, as in our case, under the assumption that the input program has a finite number of paths. With respect to [1], our framework targets the detection of timing anomalies, hence our *Guide* (through simple) advances on the straightforward model-checking algorithm of [1]. Moreover, our framework checks the presence of timing anomalies on a single model and with the property $Prop_{TA}$ given as an invariant, whereas the technique in [1] requires two models out of which one is assumed without timing anomalies and with a property expressed over the execution paths of both models. Also, the work in [1] focuses on identification of timing anomalies independently from a given program, but no complexity analysis or runtime performance results are reported and no specifics of the formal models are presented. We instead focus on the identification of code-specific scheduling timing anomalies and provide details on the formal models in Sections 3 and 4. Note that we could also add constraints in our work to upgrade to a code-independent problem. However, we believe that the code-specific problem is more interesting from an industrial point of view as most hardware architectures are subject to timing anomalies. Identifying where within a code such anomalous behavior can happen are useful to later insert mitigation mechanisms. With respect to [3] which checks timing variations of the worst-case path (computed with an WCET analyzer) using program runs on the actual architecture, our approach directly integrates the concrete architecture model in order to support such runs. The approach in [3] constructs a prediction graph which is a compact representation of instruction-level simulations of program paths. Whereas the work in [3], though extensive, relies on non-exhaustive investigation of the architecture, ours is able to provide formal guarantees with respect to it (though subjected to scalability issues).

The work [11] which introduces timing anomalies in context of the WCET analysis also proposes a simple code modification to eliminate their effects. An alternative approach, in [13] explores the abstract hardware state space using pre-computed local worst-cases called deltas. [7] proposes to speed-up WCET analyses by parallelizing their computations. However, this computation methodology generates timing anomalies that are not necessarily

present in the underlying architecture model. Lastly, the compositional timing analyses for multicores, from [4], address the problem of timing anomalies with sound techniques, ranging from pipeline stalls to overapproximation of local effects with integer linear programming.

Automatic detection of timing anomalies supports the design of predictable/compositional systems as, according to [15], the timing anomalies are “at the heart of unpredictability at processor level”. The timing anomalies could have bounded or unbounded effects (also known as domino effects), leading to a classification of computer architectures [17] into: fully timing compositional (i.e., without timing anomalies), compositional with constant bounded effects (i.e., only with bounded timing anomalies) and non-compositional (i.e., with domino effects).

6 Conclusions and Future Work

We presented a methodology to automatically detect timing anomalies based on formal models of computer architectures. Our proposal is systematic; it starts with a concrete architecture model, thoroughly tested to gain confidence in its concrete semantics. Then, we constructed abstractions, which are necessary to facilitate the study of timing anomalies, directly over the concrete architecture model. Finally, we described a detection procedure based on guided model checking. Our preliminary investigation considered a simple transformation of the search space to check for timing anomalies expressed as invariants.

New designs of either whole systems or specific components claim to be free of timing anomalies and it is important to rely on formal techniques to validate their behavior. Our preliminary study remains to be developed in several directions. We leave as our future work a similar investigation of timing anomalies due to prefetching, towards our goal for complete architecture models and the development of heuristic techniques to accelerate the model checking phase (e.g., using cuts, as in [6]).

References

- 1 J. Eisinger, I. Polian, B. Becker, A. Metzner, S. Thesing, and R. Wilhelm. Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In *DDECS*, pages 15–20, 2006.
- 2 G. Gebhard. Timing anomalies reloaded. In *WCET*, pages 1–10, 2010.
- 3 G. Gebhard. *Static timing analysis tool validation in the presence of timing anomalies*. PhD thesis, Saarland University, 2013.
- 4 S. Hahn, M. Jacobs, and J. Reineke. Enabling compositionality for multicore timing analysis. In *RTNS*, pages 299–308, 2016.
- 5 S. Hahn, J. Reineke, and R. Wilhelm. Towards compositionality in execution time analysis: Definition and challenges. *SIGBED Rev.*, 12(1):28–36, 2015.
- 6 J. Henry, M. Asavoae, D. Monniaux, and C. Maiza. How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics. In *LCTES*, pages 43–52, 2014.
- 7 R. Kirner, A. Kadlec, and P. Puschner. Precise worst-case execution time analysis for processors with timing anomalies. In *ECRTS*, pages 119–128, 2009.
- 8 L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.
- 9 M. Langenbach, S. Thesing, and R. Heckmann. Pipeline modeling for timing analysis. In *SAS*, pages 294–309, 2002.
- 10 J. Larus. Whole program paths. In *PLDI*, pages 259–269, 1999.
- 11 T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS*, pages 12–21, 1999.

- 12 S. Merz. On the logic of TLA+. *Comp. and Artificial Intelligence*, 22(3-4):351–379, 2003.
- 13 J. Reineke and R. Sen. Sound and efficient WCET analysis in the presence of timing anomalies. In *WCET*, 2009.
- 14 J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. In *WCET*, 2006.
- 15 L. Thiele and R. Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2-3):157–177, 2004.
- 16 I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in super-scalar processors. In *QSIC*, pages 295–306, 2005.
- 17 R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 28(7):966–978, 2009.
- 18 Y. Yu, P. Manolios, and L. Lamport. Model checking TLA+ specifications. In *CHARME*, pages 54–66, 1999.

Reducing Timing Interferences in Real-Time Applications Running on Multicore Architectures

Thomas Carle

Université Paul Sabatier, IRIT, CNRS
Toulouse, France
thomas.carle@irit.fr

Hugues Cassé

Université Paul Sabatier, IRIT, CNRS
Toulouse, France
casse@irit.fr

Abstract

We introduce a unified WCET analysis and scheduling framework for real-time applications deployed on multicore architectures. Our method does not follow a particular programming model, meaning that any piece of existing code (in particular legacy) can be re-used, and aims at reducing automatically the worst-case number of timing interferences between tasks. Our method is based on the notion of *Time Interest Points* (TIPs), which are instructions that can generate and/or suffer from timing interferences. We show how such points can be extracted from the binary code of applications and selected prior to performing the WCET analysis. We then represent real-time tasks as sequences of time intervals separated by TIPs, and schedule those tasks so that the overall makespan (including the potential timing penalties incurred by interferences) is minimized. This scheduling phase is performed using an Integer Linear Programming (ILP) solver. Preliminary results on state-of-the-art benchmarks show promising results and pave the way for future extensions of the model and optimizations.

2012 ACM Subject Classification Software and its engineering → Automated static analysis

Keywords and phrases Multicore architecture, WCET, Time Interest Points

Digital Object Identifier 10.4230/OASIS.WCET.2018.3

1 Introduction

The advent of multicore architectures in embedded real-time systems raises multiple challenges for the community. For single-task (single-threaded) applications running on single-core architectures, the computation of safe-yet-precise Worst-Case Execution Time (WCET) bounds is a mature research domain, in which the complexity of hardware acceleration mechanisms (e.g. branch predictors) and of programs semantical properties (e.g. infeasible execution paths) must be mitigated in the analysis in order for the problem to remain tractable. On single-core machines, using preemptions to implement multi-task applications additionally incurs Cache-Related Preemption Delays (CRPDs) [2]: since multiple tasks share the instruction and data caches, a preemptive task can invalidate cache lines still needed by preempted tasks. This leads to additional timing penalties that were not present in the analysis of single-task applications.

For applications running on multicore architectures, deriving WCET bounds for the tasks running on each core becomes even more complex. Indeed, logically independent tasks can cause or suffer from *timing interferences* induced by the execution of tasks running simultaneously on other cores. For architectures where multiple cores share caches, the same



© Thomas Carle and Hugues Cassé;

licensed under Creative Commons License CC-BY

18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018).

Editor: Florian Brandner; Article No. 3; pp. 3:1–3:12

OpenAccess Series in Informatics



OASIS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

effect as CRPD can be observed. However, caches are not the only source of contention in multicore architectures, and subtler timing interferences between tasks can be generated in other shared elements such as the interconnect.

We consider that closely integrating WCET analysis and Time-Triggered (TT) scheduling can be a pragmatic and efficient way of coping with this increasing complexity by reducing the temporal instability of the applications. Existing models [15, 6] have shown that this approach yielded good results, but they require the analyzed applications to be written in a particular fashion. On the other hand, we propose a unified, code-analysis centric approach targetting arbitrary applications, and thus suited for legacy applications. Our technique analyses each task's code in isolation, and pinpoints all instructions that can generate or suffer from timing interferences. We call these particular instructions Time Interest Points (TIPs). Our method abstracts each task of the system into a sequence of code segments delimited by two (not necessarily consecutive) TIPs. Each segment's execution duration is stabilized by injecting a busy-wait loop before the ending TIP, directly in the binary code. Each segment is represented by its duration and the worst number of TIPs executed on any control flow path contained in the code of the segment. The objective of our approach is to schedule the segment sequences according to the real-time (e.g. periods and deadlines) and functional (data dependencies) constraints of their respective tasks, while reducing the number of possible timing interferences. In this paper, we propose an ILP formulation of the scheduling constraints in order to formally expose the problem. Since this paper presents a preliminary investigation of this model we will only focus on applications composed of two tasks running at the same frequency, yet the proposed approach can be easily extended to more general task systems (e.g. multi-periodic dependent tasks). Our approach does not rely on a particular programming model, and can be used on existing code without re-writing it. It works at binary level, allowing the analysis and the automatic code injection in pre-compiled code, and freeing our analysis from any programming language constraint.

This paper is divided as follows: Section 2 gives a presentation of existing work in the domain, Section 3 formally presents the problem and Section 4 details our method. Finally, Section 5 provides a proof-of-concept and Section 6 concludes.

2 Related work

2.1 Multicore interference analysis frameworks

Several Worst Case Response Time analysis frameworks [1] for multicore architectures have been devised in the past years. Their goal is to provide a schedulability criterion for a multi-task real-time system prior to its deployment, in particular for task systems scheduled using a non TT policy (e.g. fixed priority or EDF). The objective is to derive an exact or conservative bound on the number of timing interferences that can occur on each task, and to apply timing penalties to their WCETs accordingly. In [3], the analysis framework is based on the analysis of all possible execution traces of the task system on a given architecture, and allows a very high level of precision in the modeling of the architecture components, raising the concerns of the authors about the complexity of their analysis. Alternatively, the authors of [14, 16] propose an analysis method based on real-time Calculus for applications running on multicore architectures: tasks are approximated as sequences of time intervals containing the minimum and maximum number of potential interferences that can occur for the task on these intervals. However, to the best of our knowledge, the authors do not provide methods to obtain such abstractions from actual code. Our method uses an intermediate representation that is very close to the one defined in [14] and refined in [16]. However our

model differs in several points. First, instead of verifying the schedulability of the system, we use this representation to derive a schedule of the tasks. Second, in our method each code portion corresponding to a segment in the representation is temporized using busy-wait loops so that it executes for exactly the segment duration. Finally, our method targets the general model of multicore architectures with starvation-free interconnects, instead of the more restricted model of TDMA interconnect based architectures of [16].

2.2 Multicore extensions of the PREM model

The PREM [13] model was designed to avoid timing interferences for applications running on single core architectures connected to peripherals. The main idea is to separate the application into phases of three types: Read phases perform reads in the memory to preload the application code and the needed data, Execution phases perform the task calculation using only the instructions and data present in the cache, and Write phases update the values of modified variables in the main memory. The phases of the application can then be statically scheduled so that no Read or Write phase occurs when a peripheral uses the bus¹. This model is extended to multicore architectures with scratchpad memories [15] and caches [6] by separating each task in three phases (Read/Exec/Write for the REW model or Acquisition/Execution/Restitution for the AER model) and by scheduling them statically so that memory phases from two or more cores never happen simultaneously. Each phase is time-triggered following the pre-computed starting dates. These methods work at the granularity of tasks, meaning that each task is composed of exactly one Read, one Execution and one Write phase. The Read (or Acquisition) phase prefetches all the data and instructions *potentially* required for the execution of the task in the local L1 cache or scratchpad, even though they may not be actually needed during the execution. To do so, it must be clear what data will *potentially* be read or written, as well as what code *may* be executed, by the task. This is defined by the programmer, using for example a system-level language such as PRELUDE [12] or *wrapper functions*. Tasks whose memory requirements exceed the capacity of the cache or scratchpad have to be manually divided into smaller subtasks. By contrast, our method works at a finer grain level and does not require any programmer's intervention.

3 Problem setting and formalism

In this section we define the formalism that will be used to describe our model and method throughout the paper.

3.1 Architecture

Our model focuses on multicore architectures composed of N cores, each of them connected to a private L1 cache. Each L1 cache is connected to the main memory through a starvation-free interconnect.

Each core has a programmable timer that can wake up a task sequencer (implementing a schedule computed off-line) using an interrupt through a direct link (not going through the shared interconnect). The core can program or rearm the timer through the shared interconnect. Moreover, each core also has a time stamp counter register *tsc_reg* (or an equivalent register) which counts CPU clock cycles with a fine granularity. These architecture traits are present in commercial off-the-shelf microprocessors such as the Aurix Tricore [10] or multicore ARMv8A [4].

¹ In the PREM model, peripherals such as sensors are allowed to write to the main memory.

3.2 Real-time tasks

We consider real-time applications modeled under the form of *non-preemptive* mono-periodic task systems. Formally, we denote $\mathcal{T} = \{\tau_i | 1 \leq i \leq n\}$ a task system composed of n tasks. Each task $\tau_i \in \mathcal{T}$ is characterized by:

- its period² $\tau_i.p \in \mathbb{N}$,
- its deadline $\tau_i.d \in \mathbb{N}$ (when $\tau_i.p = \tau_i.d$, the task is said to have an implicit deadline),

In the scope of this paper, we assume that each task runs on a separate core: this simplifies the scheduling ILP system, and at the same time allows us to apply our technique in situations where interferences are more likely to appear. This model is simple, yet complex enough to capture the traits of real-time applications with regard to multicore timing interferences.

3.3 WCET Computation

The identification of TIPS and the proposed scheduling method require not only WCET computation by static analysis but also intermediate results such as the analysis of the data cache. To this end, we use the Implicit Path Enumeration Technique (IPET) [11] approach which is made of three passes: (a) the path analysis, (b) the accelerator mechanism analysis and (c) the time analysis.

The path analysis consists in parsing all executions of the program. In order to increase the precision of the analysis, the IPET is performed on the binary form of the program and therefore, a compact and complete representation of a task is the *Control Flow Graph* (CFG). A CFG is a graph $G = \langle V, E, \nu, \omega \rangle$ where the nodes set V is composed of *Basic Blocks* (BB). A BB is a sequence of instructions in which only the first instruction can be targeted by a branch and only the last instruction can be a branch. $E \subseteq V \times V$ is the set of edges representing sequential execution or branches of the program. $\nu, \omega \in V$ are special empty BBs ensuring that G contains exactly one entry point (ν) and one exit point (ω).

The second analysis (b) aims at estimating the impact of accelerator mechanisms such as caches or branch predictors: these statistically improve the execution of the program (*hit*), but they do not work all the time (*miss*). A very common approach to support them is to statically compute abstract states (including all possible hardware states) and to assign a category representing their behavior. For example, for data caches [7], we distinguish four categories: *Always Hit* (AH), *Always Miss* (AM), *Persistent* (PE) or *Not-Classified* (NC). NC is the most imprecise case and a fall-back when the cache behavior is too complex. PE is a bit smarter and arises in loops: it means that the first access may cause a *miss* but the following accesses will cause *hits*. Notice that only memory instructions classified as AH are guaranteed to not generate interferences.

The last pass (c) computes the duration of BBs and weaves together (1) the WCET expression as the sum of all BBs durations multiplied by their occurrence counts on the WCET path, and (2) the constraints representing the execution paths and the effects of the accelerator mechanisms. The result gives an ILP system whose maximization provides the WCET.

3.4 TIPSGraph

We define TIPSGraphs as an intermediate representation in order to transform the CFG representing the control flow of a task into a sequence of time intervals representing the timing aspects of the task execution.

² In the scope of this paper we only target mono-periodic systems, so all tasks have the same period.

A TIPsGraph for task τ_i , $G_{TIPs}(\tau_i) = \{V_{TIPs}(\tau_i), E_{TIPs}(\tau_i)\}$ is composed of TIPs $t \in V_{TIPs}(\tau_i)$ and of edges $e \in E_{TIPs}(\tau_i)$.

TIPs $t \in V_{TIPs}(\tau_i)$ are instructions of task τ_i which can create or suffer from interferences in a multicore execution context, or *pivot instructions* which represent flow disjunctions (i.e. conditional branches) and junctions in the CFG. Pivot instructions allow our algorithm to encapsulate *if* and *loop* constructs into a single TIPsGraph edge, and thus to restrain the complexity of the subsequent ILP system.

Typically, TIPs can be:

- Memory instructions (stores and loads), when the static analysis cannot guarantee that they will always result in AH,
- Memory instructions addressing shared variables, or data residing in a cache block that can be written by another task,
- Instructions for which the static analysis cannot guarantee that they will always result in a hit in the instruction cache,
- Pivot instructions.

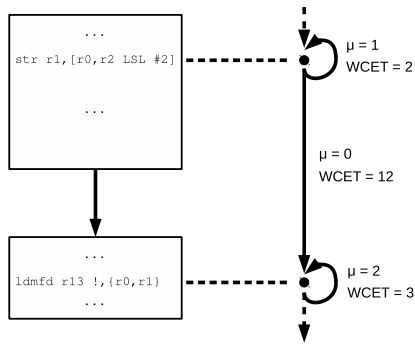
Instructions falling in the first and third categories can generate interferences for other tasks or suffer from interferences from other tasks on the interconnect (e.g. memory bus). Instructions falling in the second category are subject to interferences due to cache coherence maintenance. In the scope of this paper, we will focus on instructions falling in the first and last categories only, although the extraction of TIPsGraphs including TIPs falling in the other two categories is performed using the same algorithm. The reason for this restriction is that increasing the number of TIPs in the system dramatically complexifies the ILP system that we use for scheduling. Consequently, for the scope of this paper we consider that the tasks code is preloaded into the Instruction caches (or equivalently in private ScratchPad Memories) when the system is powered up.

An edge $e \in E_{TIPs}(\tau_i)$ is characterized by:

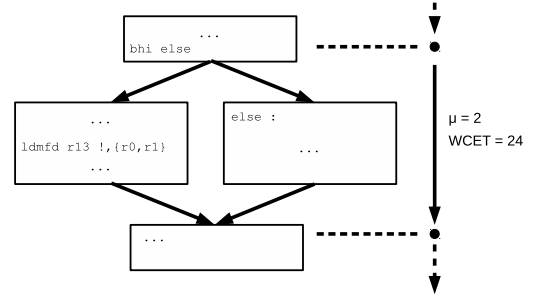
- its source TIP instruction $e.src \in V_{TIPs}(\tau_i)$,
- its destination instruction $e.dst \in V_{TIPs}(\tau_i)$,
- the worst-case number $e.\mu$ of TIPs encountered on any control-flow path linking $e.src$ to $e.dst$,
- $e.WCET$: the WCET of control-flow paths linking $e.src$ to $e.dst$.

3.5 Temporal segments sequence

Each task τ_i is represented as a sequence of time intervals (or segments) $\{(d_{i,j}, \mu_{i,j})_{0 \leq j < n_i}\}$, n_i being the number of segments that compose τ_i . These sequences are used to generate the ILP system which ultimately produces the tasks schedule. A time interval $ti_{i,j}$ is characterized by its duration $d_{i,j}$, as well as the worst case number of non-AH memory accesses $\mu_{i,j}$ performed during the execution of the segment. An important point is that a segment is characterized by an exact duration, and not by a WCET: in order to effectively reduce conflicts on the interconnect through careful scheduling of the tasks, we must know in advance when a task accesses memory. In order to suppress the temporal instability inherent to unbalanced control-flow paths and to the conservatism of our WCET estimation technique, stabilization loops are injected automatically in the binary code before the end of each segment. These loops poll the *tsc_reg* of their core until a pre-computed date is reached. Once it has been reached, the normal execution flow resumes. This technique has been introduced in the PREM model [13] to stabilize the duration of the whole Execution phase of each task.



■ **Figure 1** Example of graph extraction for a linear sequence of BBS.



■ **Figure 2** Example of graph extraction for a non-linear control structure.

The segments are straightforwardly obtained from a TIPsGraph by translating each edge in the graph into a segment.

4 Multicore WCET analysis using TIPs

In this section, we describe how TIPsGraphs are extracted from the CFG of tasks and then transformed into sequences of temporal segments. We also explain how the ILP scheduling system is generated from a set of temporal segments sequences.

4.1 Extracting a TIPsGraph from the CFG of a task

The extraction of the TIPsGraph of a task τ_i is performed by exploring the task's CFG from the entry point to the exit point. During this exploration, the extraction algorithm can be in one of two situations: either it is exploring a linear sequence of BBS without pivot instruction, or it has reached a pivot which marks a disjunction in the control flow. In this second case, a subprocedure looks for the matching junction in the graph and creates an edge between the disjuncting pivot and its matching join pivot (see 4.1.2).

4.1.1 Linear sequence of Basic Blocks

As long as the exploration procedure has not encountered a pivot instruction, it goes through the instructions of the program in sequence. If an instruction *inst* is a memory instruction (*str/ldr/stm/ldm* in ARM instruction set) which is not guaranteed to result in a hit (non-AH) in the data cache, the procedure creates a corresponding TIP *new_TIP* in $V_{TIPs}(\tau_i)$, and an edge *e* in $E_{TIPs}(\tau_i)$ from the last encountered TIP *last_TIP* to the current TIP, with $e.\mu = 0$ since no memory operation is performed between the two TIPs, and $e.WCET$ equal to the WCET of the code portion between *last_TIP* and *new_TIP*. In order to reduce the number of extracted TIPs, the procedure then regroups all non-AH memory instructions directly following *new_TIP* in the code as part of the same TIP (if such instructions are present). It computes the number μ' of all non-AH memory accesses performed by the instruction(s) grouped in the TIP, as well as the WCET of the corresponding instruction(s) $WCET'$, and creates an edge *e'*, in which $e'.src = e'.dst = new_TIP$, $e'.\mu = \mu'$ and $e'.WCET = WCET'$. This self-edge looping on the TIP accounts for the duration of the instruction(s) represented by the TIP, which generate traffic on the interconnect. The procedure then resumes the exploration of the instructions in sequence. Figure 1 illustrates

this process: the boxes on the left represent BBs in the CFG of a task, and the graph in the right is the part of the TIPsGraph corresponding to this part of the CFG. The *str* instruction in the top is analyzed as non-AH, so a TIP (a node) is created in the TIPsGraph. The self-edge on this TIP is labeled with $\mu = 1$ because the *str* instruction only performs one non-AH memory access. The WCET label for this edge corresponds to the WCET of this *str* instruction³. The next non-AH memory access found by the procedure is made by the *ldmfd* instruction at the bottom. A TIP is added to represent this instruction in the TIPsGraph, and an edge links it to the previous TIP.

If a pivot instruction p is reached, the procedure creates a corresponding TIP in $V_{TIPs}(\tau_i)$, as well as an edge e from the last encountered TIP to p , with $e.\mu = 0$ and $e.WCET = WCET(last_TIP, p)$. The procedure then follows the algorithm described in the next section. Finally, when the procedure reaches the end of the CFG, it returns $G_{TIPs}(\tau_i)$.

4.1.2 Non-linear control structures

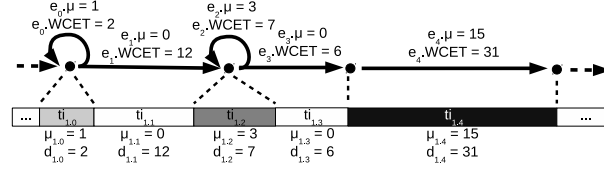
When a pivot instruction p is reached, it necessarily marks a disjunction in the control flow (an *if* branch or the start of a loop). In order to analyze the disjoint part of the CFG as a whole, the procedure first looks for the unique pivot instruction p' that marks the corresponding junction of the control flow paths, and puts it in $V_{TIPs}(\tau_i)$. This instruction is the first instruction of the first BB that (a) is a (direct or transitive successor of the BB containing p and (b) dominates the exit point (ω) of the CFG. Then the procedure explores all control flow paths between p and p' , in order to find the maximum number of non-AH memory instructions μ_{max} present on any path linking p to p' . Finally, it creates an edge e in $E_{TIPs}(\tau_i)$, with $e.src = p$, $e.dst = p'$, $e.\mu = \mu_{max}$ and $e.WCET = WCET(p, p')$. The procedure then resumes the linear exploration of the CFG described in Section 4.1.1.

The exploration of non-linear control structures is illustrated by Figure 2. The *bhi* instruction is a pivot which opens a disjoint section of the CFG. The procedure adds a TIP corresponding to this pivot in the TIPsGraph. After this, it looks for the first instruction after the disjoint portion of the CFG (the first instruction of the bottom BB) and creates a corresponding TIP. Then an analysis is performed on the two paths: the maximum number of non-AH memory accesses on either paths is 2: the left path executes a *ldmfd* instruction performing two memory accesses, both of which were labeled non-AH by the cache analysis. On the other hand, the path on the right makes no non-AH memory access. The WCET of the section between the *bhi* instruction and the first instruction in the BB at the bottom was found to be 24 time units. This WCET does not necessarily correspond to the left path.

4.2 From a TIPsGraph to a temporal segments sequence

Once a TIPsGraph containing all TIPs of a task has been extracted, its translation into a sequence of temporal segments is straightforward: the graph is traversed from its starting node to its end node, passing by each edge exactly once, with a priority given to self-edges. When traversing an edge e , it is translated into a segment s with $s.d = e.WCET$ and $s.\mu = e.\mu$. Figure 3 shows how a segment sequence is obtained from the TIPsGraph of a task τ_1 : the TIPsGraph section considered in this example starts by a TIP on the left. The first edge e_0 to be translated into a segment is a self-edge: a segment $ti_{1,0}$ is created with

³ In the figures, the WCETs are given in arbitrary time units.



■ **Figure 3** Example of temporal segment sequence extraction.

$d_{1,0} = e_0.WCET = 2$ and $\mu_{1,0} = e_0.\mu = 1$. Then a second segment $t_{i,1}$ with $\mu_{1,1} = 0$ is extracted from edge e_1 , and so on. In the figure, the density of the color of the segments reflects the number of TIPs they contain: the higher the μ , the darker the segment.

We will now present how such sets of sequences are translated into ILP variables and constraints in order to schedule the task system.

4.3 Multicore scheduling using ILP

In this section, we present the variables and constraints that are used to model our scheduling problem in ILP. Multiple objective functions can be used, optimizing different aspects, but overall the constraints presented here remain the same regardless of the optimization criterion. Finally, some constraints make use of ∞ : these constraints are encoded using a sufficiently large integer number (i.e. at least one order of magnitude larger than the variables of the system)⁴.

For each task τ_i in our system, we first introduce two sets of variables: $\{s_{i,j} | 0 \leq j < n_i\}$ and $\{\gamma_{i,j} | 0 \leq j < n_i\}$, which represent respectively the start time and the number of interferences for each segment $t_{i,j}$. In addition to these variables, we define s_{i,n_i} as the end date of the last temporal segment of τ_i (i.e. the end date of t_{i,n_i-1}). Using these variables, the following constraints impose the sequential execution of τ_i and the application of deadline $\tau_i.d$ (c_{inter} represents the cost of an interference):

$$s_{i,0} \geq 0 \quad (1)$$

$$s_{i,n_i} \leq \tau_i.d \quad (2)$$

$$\forall j : 0 \leq j < n_i, s_{i,j+1} = s_{i,j} + d_{i,j} + c_{inter} \times \gamma_{i,j} \quad (3)$$

The tricky part concerns the evaluation of $\gamma_{i,j}$ which depends on the segments of tasks running on other cores, $k.l$ (segment l of task k), that overlap the execution of segment $i.j$. Variable $\chi_{i,j-k,l} \in \{0, 1\}$ asserts whether $i.j$ and $k.l$ overlap. In this case, $i.j$ undergoes at most $\min(\mu_{i,j}, \mu_{k,l})$ interferences from $k.l$. In fact, considering all segments of τ_k overlapping $i.j$, our conservative approximation is that $i.j$ suffers in the worst case from the sum of interferences generated by each overlapping segment of core k , with at most $\mu_{i,j}$ interferences in total. The interferences with τ_k are recorded in $\gamma_{i,j-k}$ and, as exposed below, $\gamma_{i,j}$ is the sum of interferences of τ_i with all other tasks:

$$\gamma_{i,j} = \sum_{0 \leq k < n \wedge k \neq i} \gamma_{i,j-k}, \text{ with: } \gamma_{i,j-k} = \min \left(\mu_{i,j}, \sum_{0 \leq l < n_k} \mu_{k,l} \times \chi_{i,j-k,l} \right)$$

The formulation of $\gamma_{i,j-k}$ cannot be translated as is in the ILP system because of the \min

⁴ As a result, $\infty \times 0 = 0$

but we can rewrite it as:

$$\gamma_{i,j-k} \leq \mu_{i,j} \quad (4)$$

$$\gamma_{i,j-k} \leq \left(\sum_{0 \leq l < n_k} \mu_{k,l} \times \chi_{i,j-k,l} \right) \quad (5)$$

$$\gamma_{i,j-k} \geq \mu_{i,j} - \infty \times (1 - \alpha_{i,j-k}) \quad (6)$$

$$\gamma_{i,j-k} \geq \left(\sum_{0 \leq l < n_k} \mu_{k,l} \times \chi_{i,j-k,l} \right) - \infty \times \alpha_{i,j-k} \quad (7)$$

$$0 \leq \alpha_{i,j-k} \leq 1 \quad (8)$$

Eq. (4) and (5) enforce the selection of the minimum but, according to the trend of the objective function, a possible value for $\gamma_{i,j-k}$ could be 0. This is prevented by the variable $\alpha_{i,j-k}$ and Eq. (6) and (7) which ensure that either $\mu_{i,j}$, or the sum of $\mu_{k,l}$ is selected.

To detect overlapping and define $\chi_{i,j-k,l}$, we have to compare start and end dates of segments of tasks running on different cores, i,j and k,l :

$$\theta_{i,j-k,l} \iff s_{k,l} \leq s_{i,j} < s_{k,l+1}, \text{ and} \quad \theta_{k,l-i,j} \iff s_{i,j} \leq s_{k,l} < s_{i,j+1}$$

Considering the trend to minimize $\gamma_{i,j}$, $\theta_{i,j-k,l}$ (and symmetrically $\theta_{k,l-i,j}$) can be viewed as the selection of exactly one of the following constraints:

$$s_{k,l} \leq s_{i,j} < s_{k,l+1} (\theta_{i,j-k,l} = 1); \quad s_{i,j} < s_{k,l} (\theta_{i,j-k,l} = 0); \quad s_{k,l+1} \leq s_{i,j} (\theta_{i,j-k,l} = 0)$$

Introducing the cancellation variable $\beta_{i,j-k,l}$, the ILP formulation becomes:

$$s_{k,l} \leq s_{i,j} + \infty \times (1 - \theta_{i,j-k,l}) \quad (9)$$

$$s_{i,j} < s_{k,l+1} + \infty \times (1 - \theta_{i,j-k,l}) \quad (10)$$

$$s_{i,j} < s_{k,l} + \infty \times (1 - \beta_{i,j-k,l}) \quad (11)$$

$$s_{k,l+1} \leq s_{i,j} + \infty \times (\beta_{i,j-k,l} + \theta_{i,j-k,l}) \quad (12)$$

$$0 \leq \beta_{i,j-k,l} + \theta_{i,j-k,l} \leq 1 \quad (13)$$

Eq. (9) and (10) apply only if $\theta_{i,j-k,l} = 1$ (overlapping of segments i,j and k,l). When $\theta_{i,j-k,l} = 0$, only one constraint between Eq. (11) and (12) holds, depending on the value of $\beta_{i,j-k,l} \in \{0,1\}$. The last constraint ensures that $\beta_{i,j-k,l}$ and $\theta_{i,j-k,l}$ are not both set to 1 at the same time.

Notice that $\theta_{i,j-k,l}$ and $\theta_{k,l-i,j}$ can be set to 1 together when the segments start at the same date ($s_{i,j} = s_{k,l}$). Finally, $\chi_{i,j-k,l}$ is defined as:

$$0 \leq \chi_{i,j-k,l} \leq 1 \quad (14)$$

$$\chi_{i,j-k,l} \geq \theta_{i,j-k,l} \quad (15)$$

$$\chi_{i,j-k,l} \geq \theta_{k,l-i,j} \quad (16)$$

At this point, we have presented all the models and algorithms required to apply our method. In the next section, we present our preliminary results on realistic applications.

■ **Table 1** Summary of applications profiles.

bench	WCET in isolation (in clock cycles)	# segments	# TIPs	longest segment (in clock cycles)	max TIPs in a segment
edn	416221	70	5882	208056	3400
insertsort	2968	30	13	2796	1
fibcall	942	18	69	761	60

5 Proof-of-concept

We developed a prototype application⁵ based on the OTAWA [5] WCET analyzer and applied it on three benchmarks from the *Mälardalen* [8] suite: *edn*, *fibcall* and *insertsort*. These benchmarks exhibit common traits of embedded applications, and as we will see, they show very different profiles in terms of WCET and of number of memory accesses.

The first result of our analysis method is that we are able to exhibit and analyze a safe and refined timing profile of memory accesses of these applications. These profiles can also be used to extract precise arrival curves suited for methods such as [14]. We summarize key points in Table 1.

These three applications show varied profiles in number of segments, overall size and number of TIPs. Yet, one common trait is that each of them has one segment that lasts around half of its total WCET or more (WCETs and segment lengths are given in number of processor cycles). This is the result of aggregating *ifs* and *loops* inside one segment. However, we are currently working on adding more precision to the analysis of such constructs, and in particular on allowing the extraction of segments delimited by a chosen number of loop iterations.

Once this profiling is done, our prototype calls CPLEX [9] to schedule tasks two-by-two on separate cores, minimizing the makespan of the task system. We chose to fix the interference cost c_{inter} to 10 processor cycles, because it is approximately the cost of accessing the shared data scratchpad in the Aurix Tricore architecture. The result for *insertsort* and *fibcall* with this objective function is an interference-free schedule in which *insertsort* begins its execution at date 0 and finishes at date 2968. *fibcall* starts at date 170 and finishes at date 1112. Without our method the worst-case of 13 interferences should have been assumed, incurring a total additional duration of 130 cycles, which is more than a 10% overhead for *fibcall*. This preliminary experiment on real applications illustrates the possibility to reduce the number of timing interferences without having to re-write existing code, as well as the necessity to define precise analysis models in order to do so. We also tried to apply our method on application pairs featuring *edn*, but CPLEX failed to provide a solution. We believe this is linked to this application having a too long overall WCET, which increases dramatically the feasible region to be explored. These experiments convince us that our method should rely on efficient scheduling heuristics rather than on ILP solvers if we are to successfully deal with large tasks and/or large task systems.

⁵ Following this proof-of-concept, a complete analysis application is now under development.

6 Conclusion and future work

In this paper we proposed a novel approach for the WCET analysis of applications running on multicore architectures. This method is particularly well-suited for legacy applications, since it can be fully automated, requires no re-writing of existing code, and works directly at the binary level. It is based on the notion of Time Interest Points, which are instructions in the binary code that potentially cause or suffer from timing interferences on the interconnect. Our method extracts such TIPS and abstracts the application tasks as sequences of TIPS separated by temporal segments. In order to increase the timing stability of this representation, waiting loops are automatically injected at the end of each segment. These sequences of temporal segments are then scheduled in order to minimize the application makespan. In order to illustrate how this approach works, we implemented a prototype application and applied it on benchmarks from the *Mälardalen* suite. Our preliminary results (application profiling and scheduling) lead to the following conclusions:

- This method is technically feasible and promising, especially for the analysis of legacy code,
- Our next efforts should target the definition of fast-yet-efficient scheduling heuristics, to free our method from the limitations inherent to ILP and allow the resolution of larger systems as well as the introduction of new kinds of TIPS in our problems (e.g. Instruction Cache TIPS),
- In order to aggressively reduce the number of interferences, we must break down large temporal segments that represent *ifs* and *loops*. For example, we want to make it possible to extract temporal segments as specified chunks of loop iterations.

References

- 1 A. Abel, F. Benz, J. Doerfert, B. Dörr, S. Hahn, F. Hauptenthal, M. Jacobs, A. H. Moin, J. Reineke, B. Schommer, and R. Wilhelm. Impact of resource sharing on performance and performance prediction: A survey. In *CONCUR*, 2013.
- 2 S. Altmeyer and C. Maiza Burguière. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *Journal of Systems Architecture*, 2011.
- 3 S. Altmeyer, R. I. Davis, L. Soares Indrusiak, C. Maiza, V. Nélis, and J. Reineke. A generic and compositional framework for multicore response time analysis. In *RTNS*, 2015.
- 4 ARM. *ARM Cortex-A Series – Programmer’s Guide for ARMv8 - A*, v1.0 edition, 2015.
- 5 C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. Ottawa: An open toolbox for adaptive wcet analysis. In *Software Technologies for Embedded and Ubiquitous Systems*, 2010.
- 6 G. Durieu, M. Faugère, S. Girbal, D. Gracia Pérez, C. Pagetti, and W. Puffitsch. Predictable flight management system implementation on a multicore processor. In *ERTS²*, 2014.
- 7 C. Ferdinand and R. Wilhelm. On predicting data cache behavior for real-time systems. *Lecture notes in computer science*, 1998.
- 8 J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The mälardalen WCET benchmarks: Past, present and future. In *10th International Workshop on Worst-Case Execution Time Analysis, WCET*, 2010.
- 9 IBM. Cplex user’s manual. https://www.ibm.com/support/knowledgecenter/SSSA5P_12.7.0/ilog.odms.studio.help/pdf/usrcplex.pdf, 2016.
- 10 Infineon. *AURIX TC27x D-Step (32-Bit Single-Chip Microcontroller) User’s Manual, v2.2*, 2014.
- 11 Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Workshop on Languages, Compilers, and Tools for Real-Time Systems*, 1995.

- 12 C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems*, 21(3), 2011. doi:10.1007/s10626-011-0107-x.
- 13 R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for cots-based embedded systems. *RTAS*, 2011.
- 14 R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. *DATE*, 2010.
- 15 B. Rouxel, S. Derrien, and I. Puaut. Tightening contention delays while scheduling parallel applications on multi-core architectures. *ACM Trans. Embed. Comput. Syst.*, 2017.
- 16 A. Schranzhofer, J.-J. Chen, and L. Thiele. Timing analysis for tdma arbitration in resource sharing systems. *RTAS*, 2010.

Toward Contention Analysis for Parallel Executing Real-Time Tasks

Fabrice Guet

ONERA - The French Aerospace Lab, Toulouse, France

Luca Santinelli

ONERA - The French Aerospace Lab, Toulouse, France

Jérôme Morio

ONERA - The French Aerospace Lab, Toulouse, France

Guillaume Phavorin

IRT Saint Exupery, Toulouse, France

Eric Jenn

IRT Saint Exupery, Toulouse, France

Abstract

In measurement-based probabilistic timing analysis, the execution conditions imposed to tasks as measurement scenarios, have a strong impact to the worst-case execution time estimates. The scenarios and their effects on the task execution behavior have to be deeply investigated. The aim has to be to identify and to guarantee the scenarios that lead to the maximum measurements, i.e. the worst-case scenarios, and use them to assure the worst-case execution time estimates.

We propose a contention analysis in order to identify the worst contentions that a task can suffer from concurrent executions. The work focuses on the interferences on shared resources (cache memories and memory buses) from parallel executions in multi-core real-time systems. Our approach consists of searching for possible task contenders for parallel executions, modeling their contentiousness, and classifying the measurement scenarios accordingly. We identify the most contentious ones and their worst-case effects on task execution times. The measurement-based probabilistic timing analysis is then used to verify the analysis proposed, qualify the scenarios with contentiousness, and compare them. A parallel execution simulator for multi-core real-time system is developed and used for validating our framework.

The framework applies heuristics and assumptions that simplify the system behavior. It represents a first step for developing a complete approach which would be able to guarantee the worst-case behavior.

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

Keywords and phrases Contention analysis, parallel executions, measurement-based probabilistic timing analysis, probabilistic worst-case execution time

Digital Object Identifier 10.4230/OASICS.WCET.2018.4

1 Introduction

Today's multi- and many-core platforms provide an amount of computational resource unconceivable a decade ago for real-time systems. While performance increases due to the availability of multiple cores and the possibility for parallel execution, the determinism is heavily challenged by the use of optimization features like cache memories or pipelines.



© Fabrice Guet, Luca Santinelli, Jérôme Morio, Guillaume Phavorin, and Eric Jenn; licensed under Creative Commons License CC-BY

18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018).

Editor: Florian Brandner; Article No. 4; pp. 4:1–4:13

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

With multiple cores being accessible, task concurrently running (co-running) on different cores suffer from interferences while racing for shared resources like cache memories and buses. Due to concurrent accesses and bottlenecks, shared resource interferences have a prominent impact on tasks executions. Timing anomalies and worst-case conditions appear more often and the worst-case task execution time increases.

Isolation techniques and deterministic policies like Round Robin bus scheduling may be applied for reducing interferences. They would come at the cost of expensive implementations and decreased average performances. A valid alternative would consist in effectively modeling the interferences that each task suffers under different execution conditions. This way, the interference effects can be made predictable resulting into deterministic models to tasks and system behaviors.

Probabilistic models are emerging as flexible and reliable representations for tasks worst-case executions [7]. In those models, the classical deterministic Worst-Case Execution Time (WCET) is generalized with a probability distribution, the probabilistic WCET (pWCET), where it is quantified how likely an execution time may be exceeded.

Measurement-Based Probabilistic Timing Analysis (MBPTA) can be used for estimating pWCETs. It is sensitive to the measurement scenario which has been considered for measuring execution times [1, 13]. By measurement scenario it is intended, for example, a specific task mapping on multi-core processors, specific task inputs, environmental conditions, etc.. Each scenario would enforce a particular interference pattern on system resources with its specific impact to the task behavior; a scenario is representative also of interference conditions. The trace of measurements depends on such scenarios and the EVT exploits the worst-case specific to the scenario applied. The pWCET estimate would be the worst-case for only the specific scenario applied [8, 13].

In order to have safe pWCET estimates, it is necessary to determine the scenario that leads to the maximum execution time measurements, and consequently to the maximum pWCET estimate. The scenario exploration has to be efficient, since the measurement scenarios within multi-core systems can be in huge number, and reliable in offering the maximum pWCETs.

Contributions. In this work, we propose a contention analysis to explore the measurements scenarios for parallel real-time applications executed within multi-core platforms. The goal is to characterize all the scenarios and identify the worst-case from which to estimate the maximum pWCET. We name it contention analysis because it focuses on modeling interferences from contentions within cache memories and memory buses. At this stage, we do not deal with data synchronization problems e.g., deadlocks in parallel executions. Graph analyses, interference models, and contentiousness metrics are developed to characterize the worst parallel execution condition that tasks may suffer. The MBPTA is used for qualifying and comparing the execution scenarios.

We target the case of multi-core platform with shared cache, and a parallel execution simulator is developed and applied to an avionic case study. At this stage, the solution proposed is a partial one, which applies heuristics and assumptions that simplify the system behavior. It represents a first step for developing a complete approach which would be able to guarantee the worst-case behavior.

Organization of the paper. Section 2 presents the background in terms of computational modeling and analysis tools applied. Section 3 details the contention analysis and its main contributions. Analysis complexity and safety guarantees for pWCET estimates are outlined. In Section 4, it is described the experimental setup and the results of the simulation-based evaluation. Section 5 is for conclusions and future work.

Related Work. Measurement-based [probabilistic] timing analysis relies on measurements of actual task execution times for estimating either deterministic or probabilistic upper-bounds [9, 7]. An open challenge to them is the coverage problem and the so called confidence/representativity of the input measurements [9]. Our work particularly addresses this challenge with regard to task parallel execution and contention due to shared memory and memory bus.

Within a probabilistic framework, the definition of measurement scenarios and the confidence in the estimate can be addressed [1]. The confidence is related to the observation of events whose probability of occurrence is very low e.g., 10^{-15} . In our work the confidence defines the ability of selecting the worst scenario as the scenario which defines the task worst-case execution times.

In multi-core settings with competition to access shared resources, the combination of local cache misses and interference delay can be large and highly variable. The analysis of contentions represents a big challenge for the predictability of real-time embedded systems. In recent years, some progress on WCET and memory interference analysis has been achieved for multi-core systems. Some approaches have considered the impact that contention has on WCET estimates [11, 3]. They act to enrich static timing analysis models accounting for interference impacts. Some other approaches' goal is to bound interference with scheduling choices [17]. Our work copes with MBPTA and the contention analysis characterizes worst-case execution scenarios for guaranteeing more confident pWCET estimates.

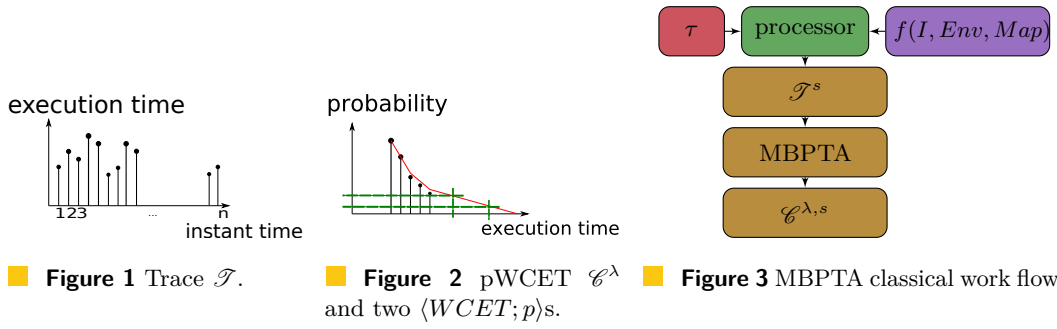
For validating our approach, it has been developed a processor simulator which is a simplified implementation of a multi-core processor. We use it to test the solution proposed and to validate the effect of certain changes to the system behavior. The simulator enables characterizing the simultaneous accesses by the cores to shared resources like cache memories and memory bus. With the assumptions made, the simulator oversimplifies the system behavior focusing on memory accessing only. The assumptions made to develop it will be released in future works in the effort to complete system modeling and converging to realistic system behaviors.

Existing simulators have been investigated before developing our own. For example, the gem5 simulator is a highly configurable architecture simulator that supports different computing architectures like ARM. It tends to simulate the real behavior of the system, but it makes it quite complex to extrapolate specific behaviors since it takes into account too many mechanisms that could happen.

SimSo [6] is a multi-core scheduling simulator that takes into account cache temporal impacts. In order to compute cache effects on task executions, SimSo makes use of the frequency of access model [6, 5]. As the number of cycles per instruction depends on the state of the processor, the best way to model is to use state machines instead of a deterministic function like the frequency of access model. Thus, the need for more realism with respect to the system behavior which is driving the development of our simulator. On the other hand, multi-core and cache aspects of SimSo inspired our work. Our cache simulator makes use of a trace of memory addresses to access, and so execute address by address. The multi-core simulation is represented with accesses concurrently applied.

2 Models and Tools

Probabilistic Worst-Case Execution Time Model. Interferences and contentions do not only increase task execution times, they also bring variability to the task behavior at runtime. Probabilistic models can better catch the underlying task execution uncertainty than deterministic models.



A sequence of execution time measurements C_j can be gathered in a trace $\mathcal{T} = (C_j)_{j \in [1;n]}$. The MBPTA estimates pWCETs, denoted by \mathcal{C}^λ , by applying the EVT to a trace of measurements \mathcal{T} [7, 8]. WCET thresholds $\langle WCET; p \rangle$ are extracted from \mathcal{C}^λ , and can be used to describe the task behavior, instead of using the whole distribution \mathcal{C}^λ . In $\langle WCET; p \rangle$, $WCET$ is the timing upper-bound on the task execution time and p is the probability for $WCET$ to be exceeded at runtime. With $\langle WCET_1; p_1 \rangle$ and $\langle WCET_2; p_2 \rangle$, by decreasing probability p $p_1 \leq p_2$, it is $WCET_1 \leq WCET_2$. This means that $WCET_1$ has more chances to be exceeded at runtime than $WCET_2$; the probability p can be seen as a level of confidence on the WCET threshold. Different probabilities can be considered e.g., 10^{-6} , 10^{-9} and 10^{-12} with the associated WCET thresholds.

Figure 1 and Figure 2 depict respectively, for an example task, a trace of execution time measurements and the pWCET estimate together with two WCET thresholds at given probabilities.

The MBPTA is sensitive to the execution scenario applied for measuring. A scenario s is an abstraction and represents a specific execution condition for the task and the system. It is an instantiation of the set of possible conditions e.g., task inputs I , environment state Env and task mapping Map . s is a function $f(I, Env, Map, \dots)$ and it affects the behavior of the task and it can also change at runtime. *The trace of measurements \mathcal{T}^s under s describes the expected execution behavior of the task under the condition. The pWCET estimation $\mathcal{C}^{\lambda,s}$ models the largest task execution times under s , Figure 3.*

An embedded real-time system has a finite number of execution scenarios $S = \{s_1, s_2, \dots, s_k\}$. Among them, there would be the worst scenario s^{worst} as the scenario which ends up into the worst measurements and the worst pWCET estimates. The problem of enumerating all the $s \in S$, is a complex problem as it could exist a large, but finite, number of parameters defining the scenarios. This work aims at determining s^{worst} that includes the worst interference from cache and memory buses due to parallel executions. s^{worst} has to be guaranteed from an effective characterization of all the possible execution conditions, including the worst ones [9].

It is important to note that we call the pWCET from each trace "worst-case", but it is only the worst-case under the considered scenario. The contention analysis here is focusing on a subset of interferences/contentions conditions. It explores them efficiently, and it defines the worst scenario among them. For validating it we make use of the MBPTA tool called DIAGXTRM [8] which accepts traces of execution time measurements as input and it estimates pWCETs from those. DIAGXTRM evaluates the confidence of EVT applicability as well as the quality of the pWCET estimates for each measurement scenario applied. It also compares multiple scenarios in terms of both average and worst-case behaviors. DIAGXTRM is developed in R and is publicly available at <https://forge.onera.fr/projects/diagxtrm2>. DIAGXTRM and MBPTA in this work, are only used to verify the soundness of the contention analysis we propose.

Directed Acyclic Graph Model. In case of precedence constraints, the execution partial ordering between real-time tasks can be represented by a Directed Acyclic Graph (DAG) $G(V, E)$ where V is a set of N nodes and E is a set of directed edges [2, 10].

Each node $n_i \in V$ corresponds to a task and it can be weighted $w(n_i)$ by the task pWCET \mathcal{C}_i^λ or WCET thresholds $\langle WCET; p \rangle$. Edges represent the order of execution between the tasks. The edge $e_{i,j}$ directly connects two nodes n_i and n_j , with n_i preceding n_j . An entry node in a DAG is a node with no predecessors; an exit node is a node without successors.

The precedence constraints encoded in DAGs impose that a node cannot start its execution before all his predecessors ended theirs. Edges can be weighted $w(e_{i,j})$ representing the communication delay which postpones task executions, i.e. task offsets. A path from n_i to n_j in a DAG exists if and only if it is possible to reach n_j from n_i ; the path is the set of nodes from n_i to n_j and the sequence of edges, $\{\{n_i, n_k, n_r, \dots, n_s, n_j\}, \{e_{i,k}, e_{k,r}, \dots, e_{s,j}\}\}$. Tasks that are linked directly by sharing an edge or by a path are said to be functionally dependent because the activation of a task requires the termination of the other one. DAGs can be used to represent mono- and multi-rate task sets. In the latter, it necessary to define DAG reduction mechanisms with multiple task instances and communication buffers [12]. The buffers are for guaranteeing the correct communication pattern and the respect of the precedence constraints across multiple task occurrences.

3 Contentions from Parallel Executions

Parallelizing task executions, whenever it is possible, enables speeding up on average the real-time application. However, co-running tasks suffer from interferences and timing anomalies which could drastically reduce the worst-case performance. Those cases have to be scrupulously modeled in order to make the system predictable.

Independence Analysis. In case of precedence constraints, the tasks that can execute simultaneously on different cores are those functionally independent. Having $G(V, E)$ representing the real-time application, two tasks n_i and n_j are said to be independent, denoted $n_i \nabla n_j$, if and only if it does not exist any path from n_i to n_j in G . At runtime, independent tasks n_i and n_j are *contenders* since they can interfere with each other by introducing contention on shared resources.

The potential contenders $\Gamma(n_i)$ of a task n_i i.e. tasks that can execute in parallel to n_i or equivalently tasks independent from n_i : $\Gamma(n_i) = \{n_j \in G \mid n_i \nabla n_j\}$. Seeking for contenders consists in determining the complement of the undirected transitive closure¹ of the DAG. Then, by taking the complement graph \bar{G} of the undirected transitive closure of the DAG, only independent tasks share an edge. The resulting graph \bar{G} is called the graph of independences, as opposed to the initial G .

$\Gamma^*(n_i)$ is the set of tasks which are independent from each other and independent from n_i : $\Gamma^*(n_i) \stackrel{def}{=} \{n_j, n_k \in G \mid n_i \nabla n_j, n_i \nabla n_k, n_j \nabla n_k\}$, and is derived from \bar{G} . Note that, n_i is included in both $\Gamma(n_i)$ and $\Gamma^*(n_i)$ and $\Gamma^*(n_i) \subseteq \Gamma(n_i)$. We call this process *independence analysis*; the next steps are for exploring $\Gamma^*(n_i)$ and its possible subsets in the quest of contentiousness.

¹ The transitive closure allows adding an edge between two nodes if they are not independent i.e. if there exists a path from one to another.

Contender List. Given $\Gamma^*(n_i)$, we now seek for the list of possible contenders to n_i . A set of c tasks, including n_i , in which every couple of tasks share an edge in \overline{G} consists of a clique $clique^c(n_i)$. The clique is relative to n_i and has size (cardinality) $c = |clique^c(n_i)|$. To note that any clique of n_i is a subset of $\Gamma^*(n_i)$, $clique^c(n_i) \subseteq \Gamma^*(n_i)$.

For each task n_i there exists a maximal size to its cliques. $clique^{\max}(n_i)$ is the maximum (largest in size) clique for n_i in \overline{G} ; $clique^{\min}(n_i)$ denotes the minimum (smallest in size) clique for n_i in \overline{G} . Both the maximum clique and the minimum clique are not usually unique. For a $clique^c(n_i)$, the $c - 1$ tasks $clique^c(n_i)/\{n_i\}$ ($clique^c(n_i)$ without n_i) are the potential contenders of n_i . The complete set of contender for n_i are all the cliques $clique^c(n_i)/\{n_i\}$ for $c \in [|clique^{\min}(n_i)|, |clique^{\max}(n_i)|]$. For a M -core processor, only up to $M - 1$ tasks can run in parallel to n_i . Then, cliques whose size c exceeds M are rejected because impossible to happen scenarios.

We assume that the more tasks are executed in parallel, the more interference between the co-running tasks there is. This is true with cache memories and the model we apply, where more concurrent tasks would evict larger portion of cache or more frequent eviction, increasing memory latencies. With memory buses it would be the same considering the same modeling with constant rates, where more concurrent accesses would increase memory communication latencies. Hence for, in order to identify the worst contention scenario, the contender list of n_i should only be composed of the largest sets of contender tasks within $\Gamma^*(n_i)$.

What can be called *valid contender list* of n_i $ContenderList(n_i)$, is defined as: $ContenderList(n_i) \stackrel{def}{=} \{clique^c(n_i)/n_i : c = \min(|clique^{\max}(n_i)|, M)\}$. The task sets in $ContenderList(n_i)$ are all of size $\min(|clique^{\max}(n_i)| - 1, M - 1)$.

Contender Classification. The objective of the worst contention analysis is identifying for n_i , its $\min(M - 1, |clique^{\max}(n_i)| - 1)$ worst contenders within $ContenderList(n_i)$. Running them in parallel together with n_i would foster the worst interferences for n_i .

The classical approach to worst contention analysis would consist in executing all the possible combinations for all the task sets in the contender list. However, the size of the contender list may be too large for an affordable exhaustive search. We define a learning procedure called *contender classification* in order to reduce the complexity of the worst contention analysis.

In this work we focus on contentions from the memory hierarchy and buses for parallel applications and co-running tasks. Inspired by it [15], we measure the number of accesses to the shared cache *accesses*. The number of accesses can be either lines fetched from the shared cache to the private caches, or writes to the shared cache. Such number of accesses has to be normalized, either in number of instructions or in time units t . We define the memory bandwidth usage *mem_band* as: $mem_band \stackrel{def}{=} accesses/t$. The rationale behind the *mem_band* metric is that a task whose memory bandwidth usage is high, leads to a lot of potential interferences for co-running tasks. *mem_band* adds to the maximal cliques for the worst-case scenarios.

Tasks with different execution lengths would impact differently the contentions. If the task under analysis has a large execution time and runs together with small execution time tasks, then the larger task would not be impacted by interferences in the last part of its execution. We overcome this issue by letting all the tasks execute continuously for the entire execution of the task under analysis. This way, the task would experience the greatest amount of interference during its entire execution: there are not zones without contention. At this stage, we do not investigate the effects of offsets between tasks; future work will be devoted to

that. Figure 4 describes the assumption we make with different interference sections (vertical lines) from the two interfering tasks τ_j and τ_k for task τ_i . The repeated executions are such that τ_j is executed continuously twice and τ_k is executed four times, while starting synchronously.

Given the *mem_band* defined as a constant rate, the synchronous case with repetition guarantees the maximum rate of contentions for the task under investigation. We are well aware that this is a simplistic assumption and perhaps an unrealistic case, but it is a starting point to study parallel task contentions. Moreover, *mem_band* accounts only for the temporal interferences at the shared memory bus level. The approach in this work is an initial step toward a heuristic able to reduce the cost for identifying worst-case execution conditions. It is obvious that the final metric would take *mem_band* into account together with other effects, and has to be developed incrementally.

3.1 Contention Analysis Discussion

A task n_i running on a M -core processor Φ^M is denoted as $\Phi^M(n_i)$. Two tasks n_i and n_j running in parallel within Φ^M are denoted by $\Phi^M(n_i \parallel n_j)$; obviously n_i and n_j are running on different cores within Φ^M . The \parallel notation also applies for p tasks $n_1 \parallel \dots \parallel n_p$.

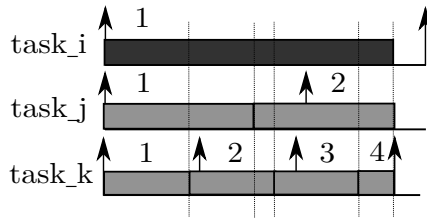
What we propose, is an approach for maximizing the execution time measurements with some possible worst interference task mapping configuration s^{worst} in order to produce a safe pWCET at system deployment. It is based on heuristics and restrictive assumptions to avoid exhaustive search. The main steps of the contention analysis can be summed up with the following four basic functions:

1. *Contender List Search* $\forall n_i \in G$, $ContenderList(n_i)$ is the result of the contender list search and the G transformation into \bar{G} . This step includes the independence analysis;
2. *Contentiousness Characterization*: $\forall n_i \in G$, $mem_band(n_i)$ is measured using τ_{mon} in case of $\Phi^M(n_i \parallel \tau_{mon}(t))$. $\tau_{mon}(t)$ is an artifact task used to monitor other task effect on shared resources;
3. *Contender Task Sets Classification*: $\forall n_i \in G$, $sort(ContenderList(n_i))$ sorts the task sets from the ones with the greatest sum of memory bandwidth usage values $sort(ContenderList(n_i))[\cdot]$ to those with the least sum of memory bandwidth usage values $sort(ContenderList(n_i))[|ContenderList(n_i)|]$;
4. *Worst Contention Scenario Measurements*: $\forall n_i \in G$, $\mathcal{T}(n_i)$ is the measurement trace under $s^{worst} \Phi^M(n_i \parallel sort(ContenderList(n_i))[1])$ to which the EVT is applied.

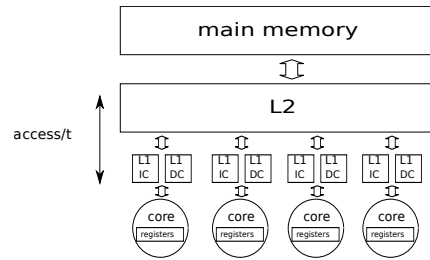
This paper offers a narrow perspective to contentions from parallel executions for the assumptions and definitions made. The proposed contentions analysis guarantees the worst execution condition among those and accounted for here.

The computation time complexity of independence analysis and contention list search (step 1) is $O(3^{N/3})$ and depends on the number of tasks N , [16].

Thanks to their memory bandwidth usage, each task set in the contender list can be ordered from the most contentious to the least. The task contentiousness characterization and classification for all the N tasks in G would take $\sum_{i=1}^N t_{n_i}$, where t_{n_i} is the execution time of n_i . One execution per task is sufficient for the task contentiousness characterization because we consider single-path tasks. The simplistic single-path task assumption could resemble to an unrealistic case. Instead, it can be applied to any actual task representation where only the worst-case path is exercised all the time, As $\sum_{i=1}^N t_{n_i} < N \times T$, where T would be the largest task execution time in G , the computation time complexity of characterizing the contentiousness (step 2 and step 3) is $O(N)$ on the number of tasks.



■ **Figure 4** Task parallel execution with different zone of contention to task_i.



■ **Figure 5** Memory hierarchy between cores with local and share memory relationship.

By executing n_i together with its most contentious tasks in its contender list, we assure the worst contention scenario s^{worst} for n_i . Measuring according to s^{worst} for all the tasks of the application (step 4) has computation time complexity $O(N)$ on the number of tasks.

4 Case Study and Simulation

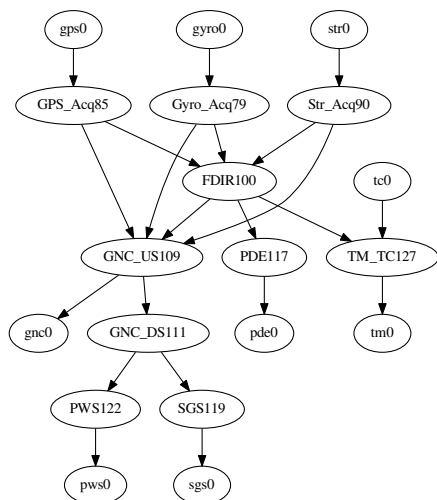
In this section we present the case study used for the experimental evaluation of the proposed contention analysis. First, we briefly describe the simulator we develop for parallel execution and contention measurements. Then, we detail the real-time application we apply for validating our contention analysis. Finally, we outline the results of the contention analysis with the simulator developed and the application selected.

Platform Simulation. One of our driving interests is to develop a generalizable and realistic multi-core parallel execution simulator with memory hierarchy. The simulator allows observing and controlling specific resources like cache memory and memory bus.

The simulated architecture is composed of four cores, with two levels of cache, L1 Data Cache (DC) and Instruction Cache (IC), and L2. The L2 cache is shared between the four cores and is accessed through a bus. The cache memories make use of the LRU policy and the write-through policy. The L1 cache memories are 4-way associative with 8 blocks per set i.e. 32 blocks in total, and whose access penalty is 1 cycle. The L2 cache memories are 4-way associative with 32 blocks per set i.e. 128 blocks in total, and whose access penalty is 4 cycles. The bus makes use of the FIFO policy. This architecture is similar to the LEON4 processor [4].

The way it has been implemented, the simulator allows choosing for the size of the cache memories and their arbitration policy, as well as the bus policy. Tasks are modeled with a trace of memory accesses i.e. a sequence of reads or writes to a memory location. With this, we represent memory accesses and we cope with the defined *mem_band*. Such traces are randomly generated with different profiles, but they can be built from the assembly code of the task for a given platform. The task profiling with traces is generic enough to apply to different task characteristics. It would suffice adapt the memory access and reproduce different task behaviors. We stress what is randomly generated are memory accesses; cache misses results from the replacement policy implemented, and are not random.

More details about the platform simulator, like the core execution and the task profiling based on a trace of memory accesses, may be found at <https://forge.onera.fr/projects/multicore-simulator>. Figure 5 details the main elements already implemented on the simulator. *mem_band* defined describes the constant rate accesses to the cache memory shared between cores through the bus.



■ **Figure 6** DAG G for the FAS case study with precedence relationships between tasks.

■ **Table 1** Results of the contender list search applied to the FAS case study.

task	$ clique^{\max}(n_i) $	$\# \Gamma(n_i)$
gps0	4	4
gyro0	4	4
str0	4	4
GPS_Acq85	4	4
Gyro_Acq79	4	4
Str_Acq90	4	4
FDIR100	1	1
tc0	4	26
GNC_US109	3	6
PDE117	4	31
TM_TC127	4	22
gnc0	4	50
GNC_DS111	4	6
pde0	4	31
tm0	4	22
PWS122	4	28
SGS119	4	28
pws0	4	28
sgs0	4	28

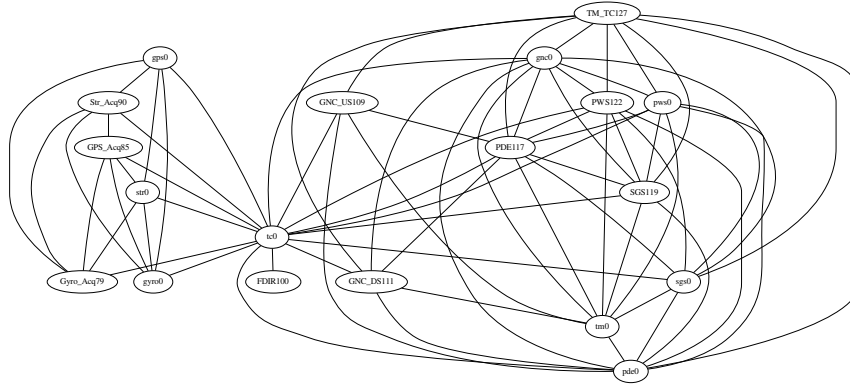
Application Setup. For this case study, we make use of the Flight Application Software (FAS) application of the Automated Transfer Vehicle designed by EADS Astrium Space Transportation for resupplying the International Space Station [12]. FAS is composed of 19 tasks and 21 precedence constraints, which can be represented by the DAG $G(V, E)$ given in Figure 6. It is a relatively small real-time application, but is already enough to show the advantage of using the contention analysis we propose.

For these experiments, we make use of the mono-rate version of the FAS application, but as already mentioned, our approach can cope with multi-rates as soon as their DAG representation is available. \overline{G} for FAS is the closure representation of G ; each arc from one node connects two independent tasks. Already with FAS, there exist lots of independent tasks, as high as 16 for task tc0. A smart and efficient contention analysis is much needed to avoid exhaustive exploration.

With respect to the implementation of the FAS tasks, we use a randomly generated trace of memory accesses for each task. Not knowing the exact original implementation, we have generated traces with different characteristic for the 19 tasks. They have been made from uniform distributions with different supports or Poisson distributions with different rates to reproduce different behaviors; the length of the traces is randomly picked from a uniform distribution. Task parameters like period and deadlines are defined according to [14]. The random generation of tasks profiles does not limit the generality of our work, since the contentiousness part would model the task for whatever access trace is considered. Also, real implementations can be included with measurements applied to characterize the access profiles.

4.1 Contention Analysis

Table 1 presents the results of the independence analysis and contender list search for the FAS case study, step 1 – *contenderList()*. As some contender lists are quite long, only the maximum clique size and the size of the contender list are given for each task. The maximal



■ **Figure 7** \bar{G} for the FAS case study with independence relationships between tasks.

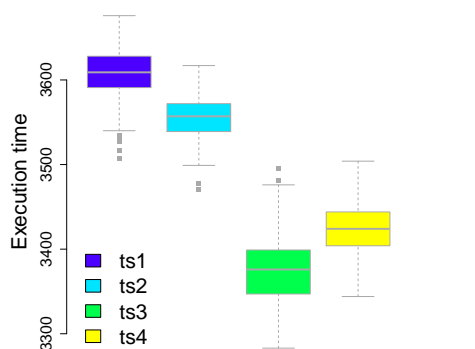
■ **Table 2** Statistics of the contentiousness characterization for the tasks in the GPS_Acq85 contender list.

task	execution time (cycles)	L1 hit/misses	L2 hit/misses	bus accesses	mem_band	rank
GPS_Acq85	1025	123/177	93/84	204	0.68	-
Str_Acq90	811	225/75	28/47	118	0.40	2
tc0	271	89/11	0/11	34	0.34	3
gyro0	374	53/47	3/44	57	0.57	1
Gyro_Acq79	641	285/15	0/15	68	0.23	5
str0	261	94/6	0/6	32	0.32	4
gps0	355	55/45	5/40	54	0.54	-

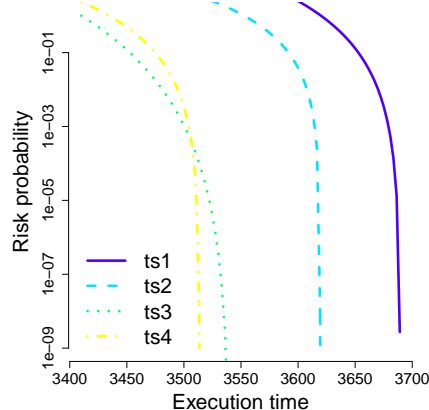
found clique for all tasks in the graph is of size five. However, as the number of available processor is four, cliques of size greater than four are not considered for the analysis. Some tasks exhibit a high number of task sets in their contender list, up to fifty for the gnc0 task. That is the main motivation for the contender classification and to avoid evaluating all the combinations.

As representative example, we detail the worst contention analysis and its experimental evaluation for the task GPS_Acq85. From the independence analysis and the contender list search, the GPS_Acq85 contender list is: $ContenderList(GPS_Acq85) = \{\{Str_Acq90, tc0, gyro0\}, \{Str_Acq90, tc0, Gyro_Acq79\}, \{str0, tc0, gyro0\}, \{str0, tc0, Gyro_Acq79\}\}$. There are six tasks to simulate for contentiousness and contention analysis.

All six tasks involved (GPS_Acq85, Str_Acq90, tc0, gyro0, Gyro_Acq79, str0) are first executed one time in isolation, $\Phi^4(n_j)$, in order to characterize their contentiousness. The results are given in Table 2 where execution times are in CPU cycles. Table 2 presents also the measurements of cache misses, bus access, and the resulting *mem_band*. As the platform simulator considered here has no prefetching mechanism, the memory bandwidth usage is computed as $mem_band = \frac{bus_accesses}{number_of_instructions}$, with the number of instructions specified by the task characteristics for the simulation. A ranking is attributed to each task, with the contentiousness defined according to the memory bandwidth usage *mem_band*. Rank 1 is for the largest memory bandwidth usage among the contenders. The task set *tsi* represents a possible execution scenario for GPS_Acq85 (cliques). The task ranking is propagated to the contender list, ordering the task sets from the most contentious set, denoted



■ **Figure 8** GPS_Acq85 and four interference scenarios tsi : expected behaviors and comparison with box plots.



■ **Figure 9** GPS_Acq85 and four interference scenarios tsi : the pWCET estimated and comparison in logarithmic scale and with the inverse cumulative distribution representation.

by $ts1$, to the least contentious task set $ts4$. It is $sort(ContenderList(GPS_Acq85)) = \{ts1, ts2, ts3, ts4\}$, with $ts1 = \{Str_Acq90, tc0, gyro0\}$, $ts2 = \{str0, tc0, gyro0\}$, $ts3 = \{Str_Acq90, tc0, Gyro_Acq79\}$, and $ts4 = \{str0, tc0, Gyro_Acq79\}$.

Once identified and ranked the tsi , for each tsi , 500 consecutive execution time measurements are obtained: the classification between $tsis$ according to their contentiousness is: $\max(ts1) > \max(ts2) > \max(ts3) \simeq \max(ts4)$. The measurements, as expected behavior, validate the ranking proposed with the contentiousness. s^{worst} for GPS_Acq85 consists of executing it in parallel of the tasks in $ts1$ i.e. $\Phi^4(GPS_Acq85 || Str_Acq90 || tc0 || gyro0)$.

DIAGXTRM is applied to the four traces \mathcal{T}^{tsi} and it provides the statistical analysis (box plots, first order, and second order statistics) as well as the worst-case analysis (pWCET) for each input trace. All the average behaviors are plotted in the box plot of Figure 8; the scenarios are compared on average. The pWCET estimates are plotted in Figure 9 with the inverse cumulative distribution representation. The worst pWCET, the greatest distribution between the four possible pWCETs, and it comes from the worst-case scenario $ts1$ $\mathcal{C}^{\lambda, ts1}$, $\Phi^4(GPS_Acq85, ts1)$. The scenario comparison as in Figure 9 validates that the worst-case scenario among the 4 considered is the one from the most contentious clique. It is worth noting that DIAGXTRM applied to the 4 scenarios passes all the tests for confident pWCET estimates; only for space reasons this is not illustrated with a plot.

5 Conclusions and Future Works

We propose a contention analysis which enables measuring, at low cost, the worst contentions for parallel executing tasks. The maximum execution times are reproduced from the worst parallel execution conditions among those investigated. The whole procedure is an heuristic approach conceived to guarantee identifying the worst contention conditions and to reduce the computation costs. It represents a first step toward an efficient and complete contention analysis.

The proposed contention analysis framework still has some limitations that have to be addressed in future works. Among others, we intend to release the single-path assumption and consider multi-path tasks. Furthermore, we intend to enrich the contentiousness metric, mem_band to account for multiple effects.

References

- 1 J. Abella, E Quiñones, F. Wartel, T. Vardanega, and F. J. Cazorla. Heart of gold: Making the improbable happen to increase confidence in mbpta. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 255–265. IEEE, 2014.
- 2 A. Al Badawi and A. Shatnawi. Static scheduling of directed acyclic data flow graphs onto multiprocessors using particle swarm optimization. *Computers & Operations Research*, 40(10):2322–2328, 2013.
- 3 Björn Andersson, Arvind Easwaran, and Jinkyu Lee. Finding an upper bound on the increase in execution time due to contention on the memory bus in cots-based multicore systems. *SIGBED Rev.*, 7(1):4:1–4:4, 2010.
- 4 J. Andersson, J. Gaisler, and R. Weigand. Next generation multipurpose microprocessor. In *Int. Conf. on Data Systems in Aerospace (DASIA), Hungary*, 2010.
- 5 Dhruva Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 340–351. IEEE, 2005.
- 6 Maxime Chéramy, Pierre-Emmanuel Hladik, and Anne-Marie Déplanche. Simso: A simulation tool to evaluate real-time multiprocessor scheduling algorithms. In *5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pages 6–p, 2014.
- 7 L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzeti, E. Quinones, and F. J. Cazorla. Measurement-Based Probabilistic Timing Analysis for Multi-path Programs. In *the 24th Euromicro Conference on Real-Time Systems*, 2012.
- 8 F. Guet, L. Santinelli, and J. Morio. On the reliability of the probabilistic worst-case execution time estimates. In *8th European Congress on Embedded Real Time Software and Systems (ERTS)*, 2016.
- 9 S. Law and I. Bate. Achieving appropriate test coverage for reliable measurement-based timing analysis. In *28th Euromicro Conference on Real-Time Systems ECRTS, Proceedings*, 2016.
- 10 Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio Buttazzo. Response-time analysis of conditional DAG tasks in multiprocessor systems. In *27th Euromicro Conference on Real-Time Systems (ECRTS)*, Lund, Sweden, July, 2015.
- 11 J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 109–118, 2014.
- 12 Wolfgang Puffitsch, Eric Noulard, and Claire Pagetti. Off-line mapping of multi-rate dependent task sets to many-core platforms. *Real-Time Systems*, 51(5):526–565, Sep 2015.
- 13 L. Santinelli, F. Guet, and J. Morio. Revising measurement-based probabilistic timing analysis. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE*, pages 199–208. IEEE, 2017.
- 14 L. Santinelli, W. Puffitsch, A. Dumerat, F. Boniol, C. Pagetti, and V. Jegu. A grouping approach to task scheduling with functional and non-functional requirements. In *Embedded real-time software and systems (ERTS)*, 2014.
- 15 L. Tang, J. Mars, and M. L. Soffa. Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, pages 12–21. ACM, 2011.

- 16 E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, 363(1):28–42, 2006.
- 17 G. Yao, R. Pellizzoni, S. Bak, H. Yun, and M. Caccamo. Global real-time memory-centric scheduling for multicore systems. *IEEE Transactions on Computers*, 65:2739–2751, 2016.

A New Hybrid Approach on WCET Analysis for Real-Time Systems Using Machine Learning

Thomas Huybrechts

University of Antwerp – imec, IDLab – Faculty of Applied Engineering, Belgium
thomas.huybrechts@uantwerpen.be

Siegfried Mercelis

University of Antwerp – imec, IDLab – Faculty of Applied Engineering, Belgium
siegfried.mercelis@uantwerpen.be

Peter Hellinckx

University of Antwerp – imec, IDLab – Faculty of Applied Engineering, Belgium
peter.hellinckx@uantwerpen.be

Abstract

The notion of the Worst-Case Execution Time (WCET) allows system engineers to create safe real-time systems. This value is used to schedule all software tasks before their deadlines. Failing these deadlines will cause catastrophic events, e.g. vehicle crashes, failing to detect dangerous anomalies, etc. Different analysis methodologies exist to determine the WCET. However, these methods do not provide early insight in the WCET during development. Therefore, pessimistic assumptions are made by system designers resulting in more expensive, overqualified hardware.

In this paper, an extension on the hybrid methodology is proposed which implements a predictor model using Machine Learning (ML). This new approach estimates the WCET on smaller entities of the code, so-called *hybrid blocks*, based on software and hardware features. As a result, the ML-based hybrid analysis provides insight of the WCET early-on in the development process and refines its estimate when more detailed features are available. In order to facilitate the extraction of code-related features, a new tool for the COBRA framework is proposed.

This paper proves the potential of the ML-based hybrid approach by conducting multiple experiments based on the TACLeBench on a first prototype. A set of annotated code features were used to train and validate eight different regression models. The results already show promising estimates without tuning any hyperparameters, proving the potential of the methodology.

2012 ACM Subject Classification Computer systems organization → Embedded and cyber-physical systems

Keywords and phrases Worst-Case Execution Time, Machine Learning, Hybrid Analysis, Feature Selection, COde Behaviour fRamework

Digital Object Identifier 10.4230/OASICS.WCET.2018.5

1 Introduction

In the last decade, embedded systems have taken a more prominent role in our environment. The possible applications with these systems are endless, e.g. smart mobile devices, cars, avionics, etc. For instance, the number of devices connected in the Internet of Things (IoT) is expected to rise to 20 billion units in 2020 [19]. Unlike general purpose computers, cyber-physical system (CPS) and IoT applications require specific context related constraints on the controller units, such as energy consumption, size, real-time behaviour (i.e. execution time), etc. This guarantees affordable, reliable and safe systems. However, these requirements also have consequences on the code running on these devices.



© Thomas Huybrechts, Siegfried Mercelis, and Peter Hellinckx;
licensed under Creative Commons License CC-BY

18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018).

Editor: Florian Brandner; Article No. 5; pp. 5:1–5:12

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

An (autonomous) car is a perfect example of a CPS with hard real-time constraints. It is from the uppermost importance that these systems not only have correct behaviour, but also are responsive. For example, the Electronic Control Unit (ECU) of the braking system should respond to the breaking pedal before a strict deadline in order to prevent catastrophic consequences. Therefore, the Worst-Case Execution Time (WCET) of code is an important value for real-time systems. However, determining this value can not always been taken for granted, as different optimisation techniques used in embedded systems and compilers influence the deterministic behaviour of the software, such as pipelining, branch prediction, pre-emption, parallelisation, etc. As a result, the WCET analysis becomes complexer to perform. In the state of practise, these influences are often simplified or neglected and compensated with a safety margin resulting in less tight or underestimated upper bound [14].

In order to determine the schedulability of software tasks, a timing analysis is required to calculate the WCET. For instance, this value is required by the scheduler of operating systems and hypervisors to schedule all tasks within specified time frames on the system [4].

In the state of the art, there are three main WCET analysis methodologies, namely the static, measurement-based and hybrid approach [13] [18]. However, a big trade-off between accuracy and computational complexity needs to be made. We believe that the hybrid methodology is the best solution as it provides the possibility to set a balance between accuracy and computational complexity depending on the needs of the user. The implementation of this hybrid approach is integrated in our COBRA framework [12] which allows us to perform code behaviour analysis on different embedded platforms.

Nevertheless, it is difficult to acquire early insight of the WCET during development with the hybrid methodology as it relies on the physical hardware and binary code to measure the execution time. In addition, the measurement process itself is time consuming. Therefore, we want to extend the hybrid methodology by applying machine learning (ML) techniques to predict the WCET of the code blocks instead of physically measuring it on the device.

In this paper, we will firstly discuss the hybrid methodology for WCET analysis. Secondly, we present a new extended approach combining the hybrid analysis with machine learning to predict the WCET without the need to actually performing physical measurements on the device unlike the measurement-based layer. Finally, we conclude with an early stage experiment proving the potential of our approach.

2 Hybrid Methodology

The hybrid WCET analysis combines the strengths of two commonly used methodologies. On the one hand, we have the *static analysis*. This approach determines the WCET based on models of the application and the target hardware without actually executing the code itself on the platform. However, the computational complexity of creating and calculating these models rises tremendously with the size of the code base and hardware optimisations on the target platform. These models, which are not always publicly available, are required to obtain sound results with a small upper bound [20]. Therefore, the static analysis becomes infeasible as the complexity of the system increases.

On the other hand, the *measurement-based analysis* is computational less complex compared to the *static analysis*. The execution time of the program is measured by running the code multiple times with different input sets, resulting in a timing distribution. This distribution leads to three important boundaries: best-case (BCET), average-case (ACET) and worst-case execution time (WCET). Depending on the analysis cost and effort that can be afforded, the number of measurements is decided. By measuring an arbitrary limited

number of input cases, it is never guaranteed that the real WCET is detected! The accuracy will drop dramatically when the amount of measurements decreases as not all systems states are covered by the given input sets. A safety margin needs therefore to be taken into account to minimise the risk of underestimating the upper bound [20].

In order to tackle the shortcomings of the previous mentioned techniques, we are using the *hybrid methodology* which combines both approaches to estimate the WCET of a software task [3] [8] [12]. The goal is to find a balance between the computational complexity of the static layer and the accuracy issue of the measurement-based layer. To apply this methodology, the source code is split into a set of smaller entities which are called “*hybrid blocks*”. Each block resembles a trace of consecutive instructions which has exact one entry and one exit point [11]. The blocks are similar to the regular “*basic blocks*” [13]. However, the size of these blocks can vary from a single instruction up to entire functions or programs depending on the accuracy and complexity we want to achieve [11]. The process starts by performing timing measurements on each block. In the second stage, all results are statically combined to acquire an estimate of the WCET.

The hybrid analysis is integrated in the *Code Behaviour fRAmework* (COBRA) tool. This open source tool is developed by the IDLab research group to examine the performance and behaviour of code on different architectures [12]. It allows developers to optimise the resource consumption on (currently) three main levels, namely WCET analysis [12], scheduler optimisation [4] and design pattern based performance optimisation for multi-core processors. First results show a significant reduction in analysis effort while keeping the WCET predictions close above the real WCET with the hybrid method compared to the static and measurement-based approach [12]. However, the source code still needs to be compiled and run on the target hardware with this technique, which still takes quite some time to perform.

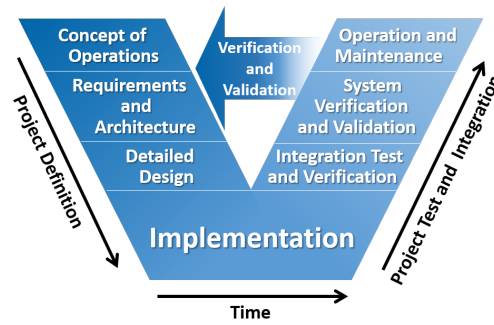
3 Early Stage Estimation

Most existing WCET tools perform the analysis on compiled binaries for specific hardware platforms. This approach requires the developers to have a compilable version of the application and the physical hardware platform before any estimate can be given [8] [17] [20]. As a result, it becomes difficult to acquire early insight of the WCET during development. Additionally, hard real-time systems have strict requirements on the WCET as failing deadlines will lead to disastrous consequences. To ensure that all deadlines are met during development, two possible scenarios occur.

On the one hand, system designers assume really pessimistic results of the upper bound to compensate for all errors and assumptions made during analysis. This results in the use of over qualified hardware which will increase the cost of the final product. Whereas small cost savings on better suited hardware will result in huge savings in mass production.

On the other hand, underestimating the final WCET during development leads to financial losses when custom developed hardware does not appear to be sufficient enough to schedule the software tasks [2]. At that point, it is important to “fail fast”, so that developers have the opportunity to correct and iterate the design much faster. “Failing faster” will limit development costs as less budget is lost due to unnecessary hardware design time and effort.

The main reason that causes the previous scenarios is the lack of insight on the WCET in the early stages of the development cycle. When we look at the V-model development process, we start with defining the project requirements and add more details to the design with each step, as shown in Figure 1.



■ **Figure 1** V-model – Development and verification process model.

In the V-model, the code development will only start in the implementation phase at the bottom of the model. As a result, the first opportunity to get insight in the WCET is after the project details are already fixed. In the case of an underestimation of the final WCET, the development process has to move up again in the V-model to adapt the design.

Altenbernd et al. [1] proposed a new methodology to gain early insight into the execution time by “training” a linear time model that translates code into basic instruction of which the timing is determined. Experiments on TACLeBench benchmarks showed promising results [1]. As the model is linear, it becomes rather difficult to model and incorporate non-linear effects on more complex platforms, e.g. caches, pipelines, etc. In addition, each basic instruction needs to be trained by generating and measuring a training program.

In order to obtain faster insight on the WCET, we believe that each system can be characterised right from the start of the development process. The characteristics of a system are described as attributes and represent the system from the high-level design down to the code- and hardware-related components. This would allow software developers to look into the influence of design choices and code changes much faster. However, in the early “*project definition*” stages there is little to no source code available to perform analysis on. Nevertheless, as the project specifications are determined, more and more system attributes are resolved that could hint the software developer to a certain interval in which the WCET is located, e.g. algorithms, code instructions, hardware model, etc. As a result, the system engineers are able to reduce the design space of suitable hardware for the system when making a decision. The key for this problem is to create a predictor to estimate the WCET value according to the system attributes which we try to resolve using machine learning.

4 Machine Learning

The execution time of a software task depends on the instructions of the followed program trace on a specific hardware platform [20]. In the case of the WCET, we are interested in the events and interactions that results in the longest path in time of the software task, such as instructions, input data, pre-emption, caches, etc. All these soft- and hardware characteristics can be described as a collection of attributes for a given software task on a specific platform. As a result, it is possible to develop models with these attributes to make predictions on the WCET. Creating a generic model with classic rules-based programming to assess a given code base for a random platform is nearly impossible. Therefore, we need another approach to create or “*train*” an estimation model with machine learning techniques.

Bonenfant et al. [3] propose a method to approximate the WCET early on in the development by applying machine learning. Their goal is to characterise source code in order to find a formula for a specific target platform and compiler toolchain, which will be achieved

by training a neural network on a set of test programs. The prediction is based on the worst-case event count. A static analysis characterises a program by counting events, which would lead to the worst-case result. The training is performed by matching the worst-case event count with the provided WCET estimates of the test programs. Eventually, the trained network will then predict the WCET of a given program with the event counts from the static analysis and the trained formula of the tested hardware [3]. In addition to the worst-case event counts attributes, the author suggests to make a classification based on the code style attributes, e.g. lines of code, loop nesting, auto-generated or handwritten code, etc.

We believe that machine learning presents a valuable solution to make early WCET predictions by classification of the complex problem statement. However, the approach presented by Bonenfant et al. might suffer from oversimplification [3]. The suggested characterisation by event counting makes a high abstraction from the source code so that valuable information of the code flow get lost. At the end, the code flow and hardware interactions will become too complicated if a program is classified based on the entire code base.

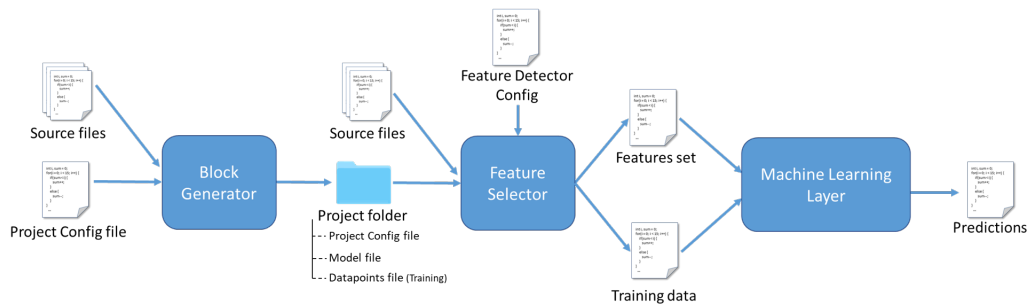
The machine learning approach provides us WCET estimates right from the start of the developed process based on the available system attributes. Therefore, rough estimates with a large deviations are available in the early stages of the development phase. However, the early predictions will provide us insight to explorer the hardware design space and exclude target platforms and configurations which will not be suitable. When the design and development process continues, more accurate attributes becomes available resulting in a gradual improvement of the WCET estimation. At that point, it is important to verify and gain trust in the trained model to obtain sound predictions.

In other research, Griffin D. et al. used a Deep Learning Neural Network (DLNN) to model the influences of other processes on shared resources [7]. The attributes used for the DLNN are the build-in Performance Monitoring Counters (PMC) in the multi-core processor. These counters keep the number of occurred events, e.g. number of cache misses, pipeline flushes, etc. The output is an interference multiplier which will be the worst-case overhead originating from the interferences of other processes. The methodology shows promising results with small underestimation errors on the final WCET [7]. However, this approach still requires a regular WCET analysis on a single core without interference to acquire the final WCET, as the obtained results needs to be multiplied with the interference multiplier. Additionally, the number of PMCs is limited which requires to run multiple measurements on the platform to acquire different parameters [7], which increases the analysis effort.

With the right approach, we are convinced that we can create a methodology based on machine learning which is able to perform accurate WCET predictions for any given architecture by combining/improving the hybrid methodology with machine learning and characterising the source code at lower levels (i.e. hybrid blocks) to avoid oversimplification (e.g. loss of code flow information, etc.) or too complex classifications. In this paper, we will focus on the software related attributes for now.

5 Feature generation

In the first step, we need to derive a set of attributes from the source code. This set of code attributes is used as features to train the machine learning layer and eventually estimate the WCET. As shown in Figure 2, the code related attributes are acquired from blocks that are generated by the *Hybrid Block Generator*. A comprehensive discussion of the block generation with the COBRA framework can be found in [12].



■ **Figure 2** Schematic overview of the Feature Selector in the COBRA-HPA chain.

After generating the hybrid blocks, a “value” for each code attribute (i.e. feature) is obtained from these blocks. Extracting these features from source code is a time consuming and error-prone task. Additionally, to train a machine learning layer that is able to provide a “solution”, i.e. WCET estimate, we need to apply a supervised learning strategy [6]. As a result, a large annotated training set needs to be created. In order to assist us in analysing and collecting features, we are developing an extension on the HPA-COBRA framework.

The *Feature Selector* module allows us to generate a formatted output file containing all features derived from the hybrid blocks in the project, as shown in Figure 2. The selection of features allows us to describe the characteristics of the code in the blocks at a higher abstraction level. This makes it less complicated to develop and train a machine learning layer that is able to generalise the problem [6]. As stated in Section 4, we need to examine which features have a significant influence on the WCET of code. Therefore, feature design requires adequate insight in the domain to compose a list of potentially relevant, quantifiable features. The next step is to assess the importance of the selected features by checking for correlations between them [9] [10] and eventually training different types of machine learning methodologies to optimise the performance on the verification set.

In order to easily generate and adjust the features for the large hybrid block collection, the *Feature Selector* has a flexible and modular design that enables us to describe a code feature in an XML-file. The detection of features is accomplished by defining and configuring basic detector modules which in turn are chainable to accommodate more complex feature detection rules. In the first prototype of this tool, there are three basic detector modules that already provides an extensive range of possibilities:

- The *Token Count Detector* counts each occurrence of a set of basic tokens and maps the result to the desired feature;
- The *Context Detector* classify if a certain syntactic rule is present in the context of the analysed hybrid block;
- The *Collection Utilities Detector* performs basic set operations on the output of two or more detectors, e.g. accumulate results, union between sets, etc.

The functionality of the basic detectors is built on the open-source parsing framework, ANTLR v4 (*ANother Tool for Language Recognition*). This tool provides the functionality to parse a text file according to a given grammar file [15]. The ANTLR framework is also part of the core of the *Block Generator* tool.

When all attributes of the blocks are determined, a list of features is generated. These features are then ready to be exported to a formatted file. Currently, a CSV exporter module is integrated in the tool which allows us to read the features in a machine learning framework,

■ **Table 1** List of code attributes extracted with the *Feature Selector*.

No. of Additive operations	No. of Multiplicative operations	No. of Division operations
No. of Modulo operations	No. of Logic operations	No. of Bitwise operations
No. of Assign operations	No. of Shift operations	No. of Comparison operations
Return statement present	No. of Evaluation operations	No. of Local variables access
No. of Local array access	No. of Global variables access	No. of Global array access

■ **Table 2** 4-Fold cross-validated Mean Relative Error (MRE) for each trained regression model.

Regression models	MRE average	MRE worst-case
Linear Regression	0.778510	1.385968
Polynomial Regression (2nd Degree)	3.005707	6.693175
Tree Regression	0.338957	0.535293
Random Forest Regression	0.518764	0.846265
Support Vector Regression (Linear Kernel)	0.272756	0.402447
Support Vector Regression (RBF Kernel)	0.497793	0.839461
K-Nearest Neighbours Regression	0.389732	0.423841
Ridge Regression	0.778263	1.303659

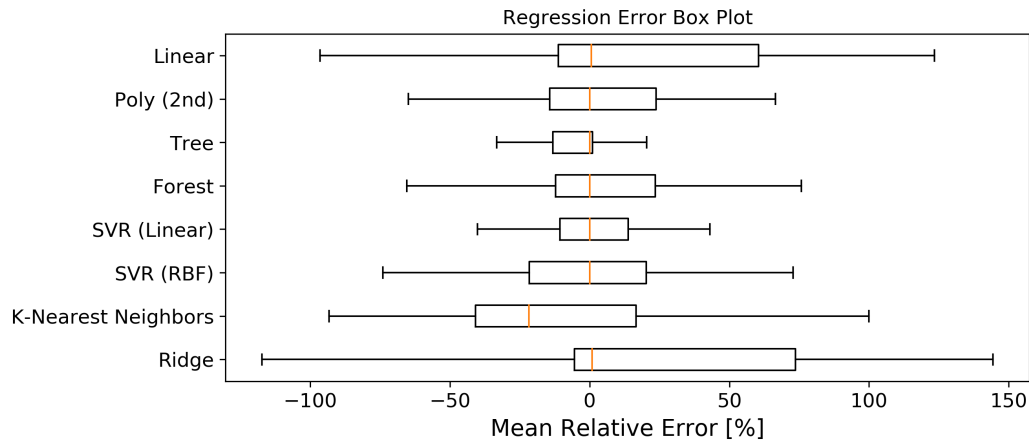
such as Scikit-Learn or TensorFlow. In addition, a WCET-annotated version can be created when the corresponding WCET results are provided. This presents the functionality to generate annotated features for training sets.

6 Experiment

The Hybrid Machine Learning methodology for WCET analysis is a new approach which is currently in the early stages of research. In order to test the functionality and performance, we need to generate data sets to train and validate different machine learning techniques. These data sets are generated from benchmark code of the TACLeBench initiative [5]. The TACLeBench is a benchmark project of the TACLe community to evaluate and compare different timing analysis tools and techniques. The benchmark programs are used by a wide community which has performed timing analysis on different platforms. This provides us a large reference database to train and validate our methodology in later stages.

For this first experiment, we have selected a set of code related attributes which are listed in Table 1. These attributes are modelled in the *Feature Selector* tool, so we are able to easily obtain features from the benchmark code. The attributes in this experiment are selected by visually inspecting the blocks and identifying which code characteristics would have a significant impact on the execution time. For this first prototype, we kept the size of generated hybrid blocks small and the number of features limited. For example, iterations statements were unrolled and not modelled as independent features, as extra features would require more training data which would made the prototype too ambitious.

At the core of the prediction mechanism is a target specific trained machine learning layer. This layer receives a set of features, as the ones in Table 1, and provides one output value that resembles its estimation for the WCET of the hybrid block. The next step is to select a set of machine learning techniques that would be suitable to perform the task. In our case, we need predictors (i.e. features) to predict a numeric value, namely the WCET. This approach is referred to as *regression* [6]. For this experiment, we have trained and evaluated



■ **Figure 3** Regression error box plot for each regression model with removed outliers.

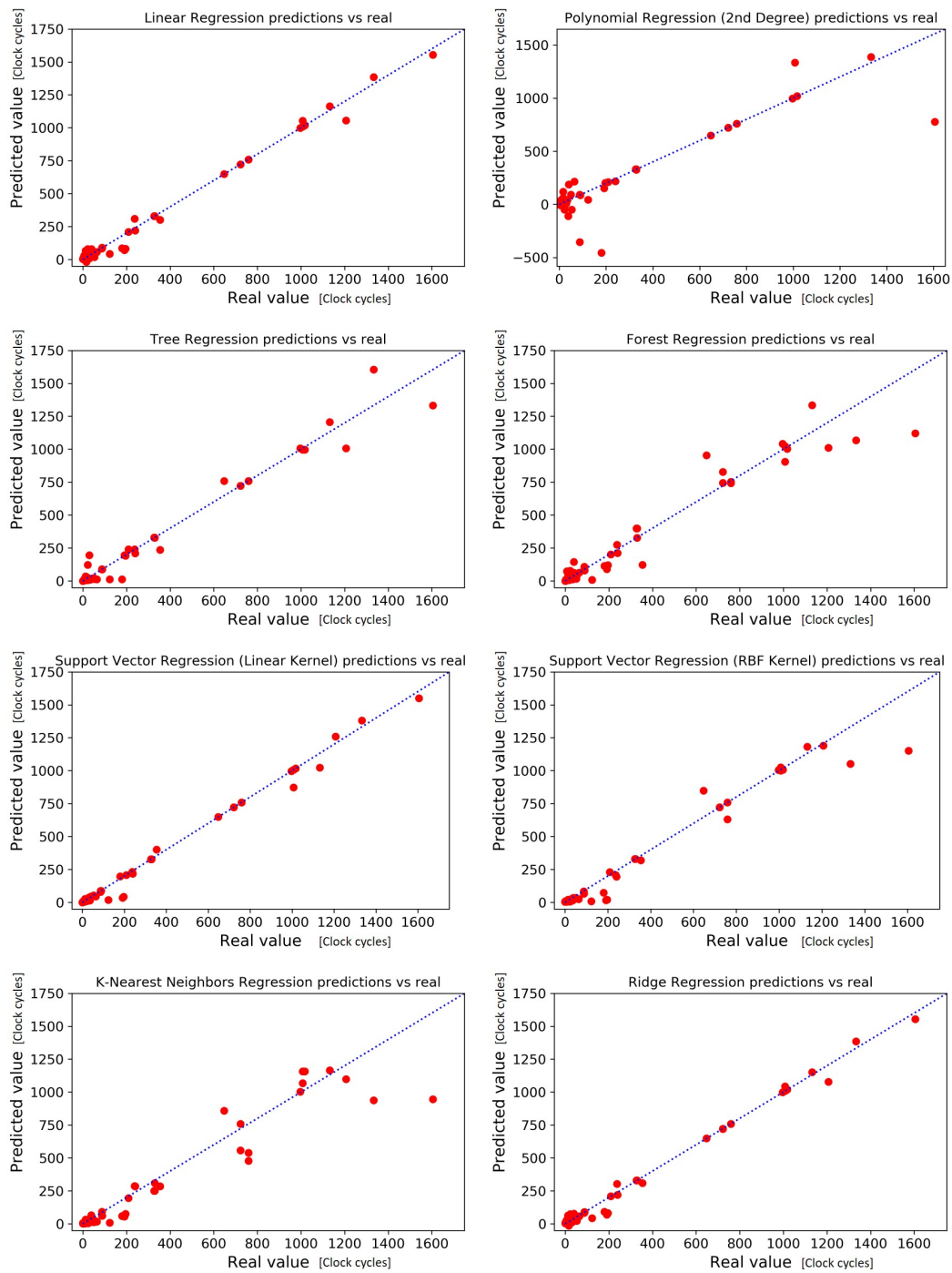
a selection of standard regression models, as listed in Table 2. All models in this experiment were implemented with the Scikit-Learn framework [16]. This framework provides a wide range of tools to create, train and validate machine learning algorithms in Python.

The training and validation sets are built from hybrid blocks generated by the *Block Generator* tool [12]. The selected blocks originate from benchmarks of the TACLeBench [5]. For this experiment, the training and validation sets have a size of respectively 75 and 25 blocks (datapoints) which are iterated in a 4-fold cross-validation process. Each of those blocks are provided with all attributes from Table 1 and annotated with a WCET value to train the model and verify the results.

The target platform used in this experiment is an ARM Cortex-M3 CPU on the EZR32 Leopard Gecko board of Silicon Labs, where the WCET of each block was measured according to the hybrid methodology explained in Section 2 [12]. To evaluate and compare the performance of the regression models, a formula is needed to get insight in the prediction error. We are mostly interested in the relative error, as an error of 5 clock cycles on big blocks with a total of 10000 clock cycles (+0.05%) is less severe than for smaller blocks with a total of 20 clock cycles (+25%!). Therefore, the Mean Relative Error (MRE) is used in the results for evaluation.

The average and worst MRE scores on the validation sets for each regression model is shown in Table 2. The best performing model in this experiment is the *Support Vector Regression* (SVR) with a linear kernel, followed by *Tree* and *K-Nearest Neighbours* Regression. The graphs in Figure 4 plot the predicted WCET values of the validation set with respect to the corresponding real measured results. The closer a point is vertically located to the dotted line, the smaller the prediction error is. However, the box plots of the error distributions in Figure 3 provides interesting insights. If we remove the outliers, we see that the *2nd Polynomial Regression* actually performs significantly better than initially thought when comparing it to the result of Table 2. As the MRE is sensitive for the large outliers the model predicted, e.g. the model had prediction errors (MRE) down to -4000% and lower!

After validating the ML models on small hybrid blocks, we performed an experiment on three TACLeBench applications to test the performance of the hybrid methodology. The results of these experiments are shown in Table 3. This table shows the errors of each model's estimation of the WCET. A negative and positive error resembles respectively an under- and overestimation of the real WCET.



■ **Figure 4** Predicted vs real WCET values for trained regression models on the validation sets.

■ **Table 3** Prediction error of the hybrid ML approach on three TACLeBench application for each trained regression model.

Regression models	Bitonic	Bsort	Recursion
Linear Regression	-49.3%	102.2%	-0.2%
Polynomial Regression (2nd Degree)	100.2%	-266.3%	-10.9%
Tree Regression	18.1%	18%	-52.8%
Random Forest Regression	-11.8%	113.7%	-14.6%
Support Vector Regression (Linear Kernel)	-24%	8.5%	-55.3%
Support Vector Regression (RBF Kernel)	-31.9%	-36.6%	-45.6%
K-Nearest Neighbours Regression	-45,9%	38.5%	-54.1%
Ridge Regression	-47,1%	56.8%	0.5%

The results in Table 3 show good results for the *Tree* and *SVR with Linear Kernel* models. However, they perform significantly worse for the *Recursion* benchmark compared to the other ML models. This benchmark has a small code base which repeatedly gets called recursively. In this case, a small error on a hybrid block will result in a rising total error when the number of recursive calls increases.

7 Discussion and Future Work

In this paper, early stage experiments show that machine learning based WCET prediction is a high potential technique. In the context of the WCET, we are mostly interested in the worst performance as we need to evaluate the error margin in order to obtain the upper bound. The results of the first prototype (Table 2) indicate that the *Support Vector Regression* with a linear kernel has worst-case the lowest MRE of 40.2%, however it reached an average of 27.3%. In addition, we see in Figure 4 that the SVR (Linear Kernel) accurately predicts all features of the verification set with just a few small outliers compared to the other trained regression models. The worst performing cross-validated iteration of the SVR is possibly due to a validation bucket that contained less trained attributes. This problem can be mitigated by further extending the labelled dataset.

The higher performance of the linear kernel models probably is because of the linear characteristics of the trained features on a “simple” architecture, e.g. single core, no caches, etc. Therefore, these models are better in generalising the problem, as more complex models will overfit the solution [6]. The well-performing SVR model tries to fit the data such that the distance between the data points and the fit, which is referred to as the supporting vectors, is maximised [6]. On the other hand, the Tree model partitions the data points in clusters for which a value is assigned to. In this first experiment, we achieved good results. This approach however, estimates discrete values. If we want to have high accurate output values, we need a high partitioning of the data space which results in large complex trees.

Nevertheless, none of the tested machine learning models will be excluded from further research at this point, as these are still preliminary results in this experiment. The results in Table 2 were acquired with the default configuration of each regression model without tuning any of the hyperparameters, as the goal of this experiment is to examine the feasibility of applying machine learning techniques to predict WCET values. Therefore, we can conclude that the WCET estimations show already promising results on small hybrid blocks. We believe that additional tuning of the attributes and models will further improve the accuracy of the predictions.

In future work, we will continue to improve the prediction models [6] [21], selecting the right features in a systematic approach (feature engineering) [9] [10], examine the accuracy/computational complexity trade-off for bigger blocks and extend the list with ensemble models and neural networks. In the case of neural networks, more labelled data is required to train the model in order to achieve acceptable results. In addition, each trained model is platform specific. By including hardware/toolchain related attributes to the model however, we believe a better performing, more generic model can be trained for related architectures. A more general predictor model will lower the effort to train target specific models. In addition, modelling the interferences on shared resources with a DLNN seems a feasible approach as shown by [7]. Therefore, it provides a first step to extend our methodology to more complex hardware.

8 Conclusion

The early stage experiments in this paper show that machine learning-based hybrid WCET prediction is a high potential methodology. In a first stage, we discussed the importance of early stage WCET prediction where current analysis methods falls short, as they rely on an existing code base. In the second stage, we propose to add a predictor model using machine learning to our hybrid methodology. This combined approach is a new concept where code features are utilised to estimate the WCET of a hybrid block.

In order to prove the potential of this methodology, we created the COBRA-HPA framework that splits code into blocks according to the hybrid methodology and dynamically generates corresponding features based on a configurable detector chain. The resulting features can be used to estimated the WCET of the block or to train machine learning models.

Finally, we trained and compared the performance of eight different regression models with code features generated by the *Feature Selector*. The results of this first prototype show that the *Support Vector Regression* with a linear kernel has the best performance. In overall, the WCET estimations of all models have promising results. Additional tuning of the attributes and models will further improve the accuracy of the predictions.

We believe that this approach will be the solution to early stage WCET prediction. Therefore, we will continue to improve the regression models by focussing on feature engineering, tuning prediction models, acquiring more data, extending the models with (deep) neural networks and ensemble models, and integrate hardware/toolchain related features.

References

- 1 Peter Altenbernd et al. Early execution time-estimation through automatically generated timing models. *Real-Time Systems*, 52(6):731–760, Nov 2016. doi:10.1007/s11241-016-9250-7.
- 2 B. W. Boehm et al. *Software Engineering Economics*. Prentice-Hall PTR, Englewood Cliffs, NJ, 1981.
- 3 Armelle Bonenfant et al. Early WCET Prediction Using Machine Learning. In Jan Reineke, editor, *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 57, pages 5:1–5:9, 2017. doi:10.4230/OASIcs.WCET.2017.5.
- 4 Yorick De Bock et al. Task-Set Generator for Schedulability Analysis using the TACLeBench benchmark suite. In *Proceedings of the Embedded Operating Systems Workshop : EWiLi 2016*, pages 1–6, 2016. URL: <http://ceur-ws.org/Vol-1697/>.

- 5 H. Falk et al. TACLeBench: a benchmark collection to support worst-case execution time research. *Proceedings of the 16th International Workshop on Worst-Case Execution Time Analysis (WCET'16)*, 2016.
- 6 A. Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2017.
- 7 David Griffin et al. Forecast-based Interference: Modelling Multicore Interference from Observable Factors. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems, RTNS '17*, pages 198–207, 2017. doi:10.1145/3139258.3139275.
- 8 Jan Gustafsson et al. *Approximate Worst-Case Execution Time Analysis for Early Stage Embedded Systems Development*, pages 308–319. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi:10.1007/978-3-642-10265-3_28.
- 9 Isabelle Guyon and André Elisseeff. An Introduction to Variable and Feature Selection. *J. Mach. Learn. Res.*, 3:1157–1182, 2003.
- 10 Mark A. Hall. Correlation-based Feature Selection for Discrete and Numeric Class Machine Learning. In *Proceedings of the 17th International Conference on Machine Learning, ICML '00*, pages 359–366, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.
- 11 T. Huybrechts et al. Hybrid Approach on Cache Aware Real-Time Scheduling for Multi-Core Systems. In Fatos Xhafa, Leonard Barolli, and Flora Amato, editors, *Advances on P2P, Parallel, Grid, Cloud and Internet Computing*, pages 759–768, Cham, 2017. Springer International Publishing.
- 12 T. Huybrechts et al. COBRA-HPA: a Block Generating Tool to Perform Hybrid Program Analysis. *Int. J. of Grid and Utility Computing*, in press 2018.
- 13 P. Lokuciejewski and P. Marwedel. *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems*. Springer Netherlands, 2011. doi:10.1007/978-90-481-9929-7.
- 14 Enrico Mezzetti and Tullio Vardanega. On the Industrial Fitness of WCET Analysis. In *The 11th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2011.
- 15 Terence Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Bookshelf, 2013.
- 16 F. Pedregosa et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- 17 P. Puschner and Ch. Koza. Calculating the Maximum, Execution Time of Real-time Programs. *Real-Time Systems*, 1(2):159–176, 1989. doi:10.1007/BF00571421.
- 18 Jan Reineke. *Caches in WCET Analysis*. PhD thesis, University of Saarlandes, 2008.
- 19 Rob van der Meulen. Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016, 2017. URL: <http://www.gartner.com/newsroom/id/3598917>.
- 20 Reinhard Wilhelm et al. The Worst-case Execution-time Problem - Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008. doi:10.1145/1347375.1347389.
- 21 Dani Yogatama and Gideon Mann. Efficient Transfer Learning Method for Automatic Hyperparameter Tuning. In *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics*, volume 33, pages 1077–1085, 2014.

TASKers: A Whole-System Generator for Benchmarking Real-Time-System Analyses

Christian Eichler

Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Tobias Distler

Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Peter Ulbrich

Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Peter Wägemann

Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Wolfgang Schröder-Preikschat

Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Abstract

Implementation-based benchmarking of timing and schedulability analyses requires system code that can be executed on real hardware and has defined properties, for example, known worst-case execution times (WCETs) of tasks. Traditional approaches for creating benchmarks with such characteristics often result in implementations that do not resemble real-world systems, either due to work only being simulated by means of busy waiting, or because tasks have no control-flow dependencies between each other. In this paper, we address this problem with TASKERS, a generator that constructs realistic benchmark systems with predefined properties. To achieve this, TASKERS composes patterns of real-world programs to generate tasks that produce known outputs and exhibit preconfigured WCETs when being executed with certain inputs. Using this knowledge during the generation process, TASKERS is able to specifically introduce inter-task control-flow dependencies by mapping the output of one task to the input of another.

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

Keywords and phrases benchmarking real-time-system analyses, task-set generation, whole-system generation, static timing analysis, WCET analysis

Digital Object Identifier 10.4230/OASICS.WCET.2018.6

Supplement Material The source code of TASKERS is available at: <https://gitlab.cs.fau.de/taskers>.

Acknowledgements This work is supported by the German Research Foundation (DFG), in part by Research Grants no. SCHR 603/9-2 (AORTA), SCHR 603/13-1 (PAX), the Transregional Collaborative Research Centre “Invasive Computing” (SFB/TR 89, Project C1), and the Bavarian Ministry of State for Economics under grant no. 0704/883 25.

1 Introduction

Timing and schedulability analyses are essential steps in the development process of a real-time system, allowing developers to ensure, for example, that all tasks meet their deadlines. As analysis runtimes increase with system complexity [6], such analyses usually cannot explore the entire problem space [15], but in part need to make conservative assumptions on system



© Christian Eichler, Tobias Distler, Peter Ulbrich, Peter Wägemann, and Wolfgang Schröder-Preikschat;

licensed under Creative Commons License CC-BY

18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018).

Editor: Florian Brandner; Article No. 6; pp. 6:1–6:12

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

behavior to be feasible and to complete within an acceptable amount of time. When determining upper bounds for the response times of tasks in a system, a typical approach, for example, is to use pessimistic estimates of operating system overheads, because precisely determining their influence often is too expensive [3, 18]. For the same reason, when analyzing the schedulability of a task set, many analysis algorithms assume the absence of inter-task dependencies [1, 4, 5]. Relying on these kinds of assumptions, on the one hand, ensures feasibility, but on the other hand typically also prevents timing and schedulability analyses from producing exact results. This pessimism entails two main consequences: First, with different analysis techniques possibly providing different results, it creates the need to assess the accuracy of each method, both on an absolute scale as well as in relation to alternative techniques. Second, the fact that analysis results are partially based on assumptions makes it necessary to verify whether the guarantees determined still hold when the system is executed on the targeted platform. Both problems can be addressed by benchmarks that provide known baselines, for example, by conducting evaluations in which system code runs on real hardware and execution times are first determined by measurement and then compared to the predicted results.

Comprehensive evaluations require a large number of benchmarks with varying properties and therefore, in general, cannot be performed using real-world programs with given temporal properties only. As a result, it is essential to rely on generated benchmark systems which, to obtain meaningful results, must be configurable yet resemble actual systems as closely as possible. Consequently, such generative benchmarking approaches should, in the optimal case, provide traceable and reproducible inter-task control-flow dependencies, include scheduling and preemption effects, and consider interference caused by the hardware (e.g., caches).

In this paper, we focus on an essential step towards this ultimate goal: The automatic generation of realistic benchmark-system implementations with predefined properties. Most notably, the generated systems have to meet the following requirements: (1) The task implementations of these systems should resemble real-world task programs; in particular, they must not only simulate work by performing busy waiting [10, 14, 23]. (2) The worst-case execution times (WCETs) and the number of generated tasks need to be configurable to be able to create customized benchmark systems that are tailored to specific use cases. Their execution time must vary for different inputs [24, 25], as it is the case for most real-world tasks. (3) The tasks of a generated system need to have dependencies between each other, for example, due to one task's output serving as input for a subsequent task. Inter-task dependencies usually pose a challenge to static analysis techniques and therefore are an ideal means to evaluate the individual strengths and weaknesses of different methods, especially in the context of whole-system timing analysis [6].

To provide the properties discussed above, we designed and implemented TASKERS, an approach to automatically generate realistic system implementations that serve as a reliable baseline for benchmarking static analyses. Given a predefined platform and timing configuration, TASKERS combines different code patterns that are typical for real-world programs to create a task-set implementation whose tasks match the specified WCETs when being executed with known (configurable) worst-case input values. Starting the system with other input values, on the other hand, results in different (lower) execution times due to task implementations comprising multiple possible program paths. During the task-set generation process, TASKERS connects tasks by precedence constraints and thereby introduces control-flow dependencies. Using the generated task set, in a final step TASKERS then creates the entire executable benchmark system, including an OSEK-compliant operating system [20].

In summary, this paper makes the following contributions: (1) It presents the TASKERS approach for generating benchmarks systems with predefined task WCETs, inter-task control-

flow dependencies, and a defined overall worst-case response time (WCRT, the time between the release of an event and the response to it). (2) It discusses details of our current prototype, the TASKERS tool. (3) It evaluates the TASKERS tool on an ARM Cortex-M4 platform.

The remainder of this paper is structured as follows: Section 2 details the challenges arising from generating complex task sets that resemble real-world systems. Section 3 describes our TASKERS approach of generating benchmark programs comprising whole real-time systems. Section 4 presents our prototype that is used for the evaluation in Section 5. Section 6 discusses related work. Finally, Section 7 outlines future work and concludes.

2 Problem Statement

In this section, we discuss two core challenges of generating realistic benchmark systems: (1) Sound synthesis of configurable whole-system implementations that are composed of tasks with inter-task control-flow dependencies and (2) generation of code for said systems that conforms to predefined temporal properties. To complete the problem statement, we furthermore provide details on TASKERS' underlying system model and assumptions.

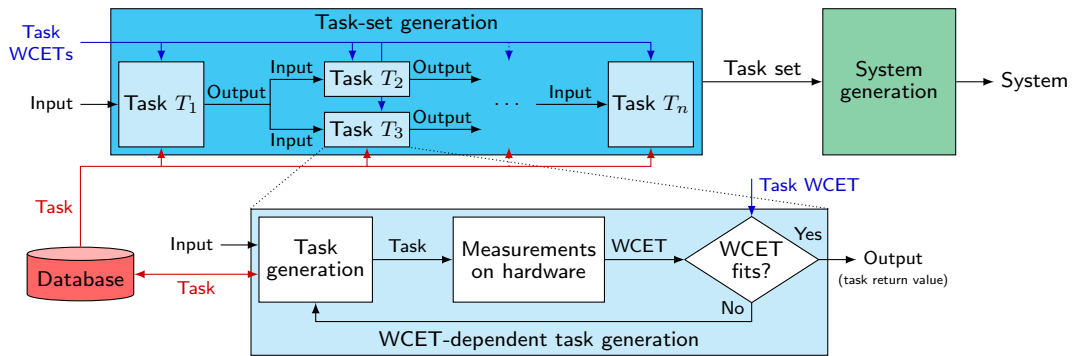
Challenge #1: Configurable Systems Synthesis with Inter-Task Dependencies

The typical approach to model benchmark systems by artificially consuming execution time (i.e., busy waiting) is precise in time and simple to configure [4, 5, 18]. However, due to the absence of executable code, this approach is inapplicable for the assessment of static analysis techniques. Furthermore, it excludes effects that result from task interactions. For example, given a producing and a consuming task, the consumer's local worst-case path may be infeasible combined with the producer's worst-case path due to mutually exclusive effects. Contrary, evaluation schemes based on real-world systems, or parts of them, are universally applicable as benchmarks. In addition, they inherently incorporate inter-task effects and thus resemble the actual system behavior much more accurately. However, in general, it is challenging to manipulate the temporal properties of functional code to a given specification; not to mention tuning the system's global worst-case path. The resulting problem is, therefore, the generation of benchmark systems with configurable temporal properties that feature realistic precedence constraints and dependencies and provide a common baseline for the assessment of both runtime evaluation and static code analysis.

Our approach: We synthesize evaluation systems with given utilization, number of tasks, and worst-case properties under consideration of target platform and real-time operating system used. To incorporate inter-task dependencies, we rely on task-dependency graphs, which are used to form task systems that fulfill the configured temporal parameters. Therefore, we derive the tasks' WCETs as well as an overall worst-case path budget, which is the result of their composition.

Challenge #2: WCET-Adhering Code Generation

A sound composition and code generation for non-trivial benchmark systems with numerous tasks and interdependencies is challenging: First of all, paths that are locally feasible may become globally infeasible, changing the costs of the global worst-case path. Similarly, inter-task dependencies, such as shared memory, may further affect individual execution costs. Consequently, path budgets and task WCETs must be adapted to compensate for these effects without jeopardizing the overall soundness. These effects virtually impede a straight-line code generation for individual tasks.



■ **Figure 1** Overview of the TASKERS approach.

Our approach: From the generated task set, we derive worst-case paths and synthesize operational code (i.e., resembles functional code) along with the associated worst-case inputs/outputs. We use a linear regression approach to model the relationship among desired path length and the actual WCET (i.e., number of instructions). In the generation process, we assemble the tasks' implementations to be longest if executed with the preselected worst-case inputs while ensuring that all other paths are shorter. To verify the actual WCETs and the resulting WCRTs, we leverage knowledge of the generated system to trigger and measure local and global worst-case paths at runtime on the target platform. Potential deviations are fed back until all specifications are met.

System Model and Assumptions

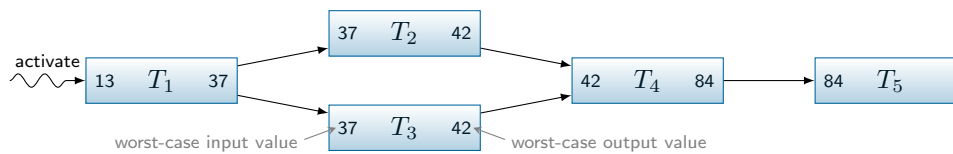
TASKERS targets embedded real-time systems with a single processing core and an operating system that complies with the OSEK standard [20] (i.e., uses fixed-priority scheduling). Consequently, we make the following two assumptions:

(1) The generated task set is OSEK BCC1 compliant. That is, tasks are executed run-to-completion (i.e., non-blocking) in the absence of asynchronous interrupts and events. TASKERS generates tasks and task sets under consideration of the target system's scheduling semantics such that their schedule at least becomes deterministic in the worst case (i.e., the worst-case path is triggered). This assumption is not a general limitation: more elaborate operating systems and scheduling variants can be incorporated, as long as the worst-case behavior can be mapped to a defined, deterministic equivalence class of schedules.

(2) Emanating from a given state, the processor is deterministic for any consecutive code sequence. That is, a task's WCET can be measured by the worst-case inputs. Even though this might seem like a severe limitation at first, the assumption of an adequately controllable hardware state is entirely feasible: TASKERS exercises control over all code that is executed. This includes, in particular, all initialization processes, whose appropriate size ensures, for example, a known cache state [21]. Sufficiently predictable hardware architectures that feature, for example, an LRU cache replacement strategy are generally available in the embedded domain (e.g., Infineon's Aurix lineup).

3 The TASKERS Approach

In this section, we present the whole-system generator TASKERS and discuss how it solves the challenges identified in Section 2. In a nutshell, TASKERS constructs operational real-time systems by combining generated tasks, thereby ensuring certain properties of both the tasks



■ **Figure 2** Example of a dependency graph for a task set with tasks T_1, \dots, T_5 .

and the overall system. Figure 1 outlines the three basic steps of the system-generation process: (1) In the first step, TASKERS randomly generates a directed acyclic dependency graph based on the number of tasks, their WCETs, and the input value provided by the user. The generated graph describes the producer-consumer relationship and therefore the precedence constraints, while its acyclicity guarantees that there are no cyclic dependencies and thus all dependencies can be fulfilled. These inter-task dependencies mimic the behavior of real-world applications, such as the dependence on a value that is read from a sensor [23]. (2) In the second step, TASKERS generates the code for each task in such a way that a task's worst-case path is either triggered by the user-provided input (for the initial task) or by the calculated output of the task's predecessor(s) in the dependency graph. To create a task implementation with a predefined WCET, TASKERS first generates a program whose longest path comprises a specific number of instructions and then determines the actual WCET of the program by measurement on the target hardware. In case the measured WCET does not yet match the intended WCET, in an iterative process, the generator further refines the program by adding or removing instructions. During the generation process, TASKERS stores information about programs and their measured WCETs in a database to speed up the creation of subsequent tasks. (3) In the final step, TASKERS uses the previously generated task set to produce the final, fully operational operating-system binary.

Solution #1: Composing Whole-System Benchmarks with Known Properties

In order to create realistic benchmark systems with inter-task control-flow dependencies, TASKERS starts the generation process by constructing a directed acyclic graph that models all the dependencies between tasks in the final system. As illustrated in Figure 2, for each task this graph also contains information on (1) the input value that should trigger the task's worst-case path and (2) the corresponding output value to be produced for this input. Consequently, TASKERS is able to introduce control-flow dependencies between tasks by mapping the worst-case output value of one task to the worst-case input value of another task. Using the worst-case input value specified by the user, the initial task (i.e., T_1 in the example given in Figure 2) of a generated set is activated through an external signal.

Having constructed the dependency graph, TASKERS then starts to generate code for the individual tasks that meet the requirements specified in the graph. For this step, TASKERS relies on a modified version of the task generator GENE [24], a tool that automatically composes program patterns of real-world tasks to create realistic programs with input-dependent execution times. Most importantly, the programs generated by GENE have known worst-case paths of configurable length (i.e., the number of executed instructions) that are triggered by user-specified worst-case input values. Due to GENE's task generation process being unable to provide specific worst-case output values, we extend the generation approach for TASKERS by introducing a mechanism to track the range of values every variable may have over the course of a task's execution. Knowing the contents of variables for the worst-case input value, TASKERS therefore is able to force a task program to produce a predefined, nevertheless input-dependent, worst-case output value by making the program return the value of an input-dependent variable plus/minus a suitable constant.

Solution #2: Generating Task Implementations with Predefined WCETs

As described above, TASKERS' task generator produces task implementations with a configurable number of instructions on the worst-case path. However, with different instructions usually having different execution times, additional measures are necessary to obtain programs that exhibit predefined WCETs (in terms of milliseconds). For this purpose, TASKERS uses an iterative process that repeatedly adjusts the *path budget* for a task (i.e., the length of the worst-case path in terms of instructions) until the actual WCET of the generated implementation on the target hardware matches the user-specified WCET for the task.

Having generated a task program based on an initial, estimated path budget, TASKERS executes the program on the target hardware with the predefined worst-case input value and measures the resulting execution time. If the measured WCET deviates from the user-specified WCET by less than a configurable margin (e.g., 0.1%), the task-generation process is complete. Otherwise, TASKERS starts an additional iteration and generates another program, this time with a different path budget. To determine the new path budget, the tool exploits the fact that, for its generated programs, there is a linear correlation between path budget and WCET (see Section 5.1). In particular, TASKERS calculates the linear regression between path budget and measured WCET using several interpolation points, and estimates a fitting anchor value for the new path budget. If necessary, the tool further refines the estimated budget in subsequent iterations by repeated sampling close to the current anchor value and enhancing the linear regression.

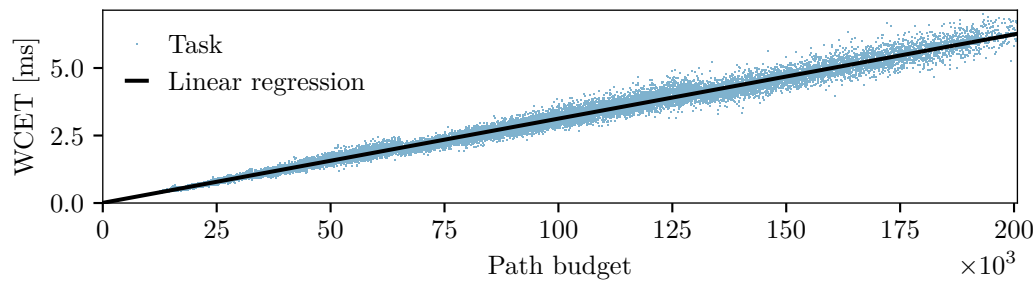
To speed up the creation of additional tasks, TASKERS stores knowledge on the generated task programs and their respective WCETs in a database. As a consequence, if, for example, a subsequent task is requested to have a similar WCET, the tool does not need to start the generation process from scratch, but can continue based on a previously generated program.

4 Implementation

TASKERS' task generator is implemented on top of the LLVM compiler infrastructure [17]. The tasks are generated using LLVM's intermediate representation (LLVM IR), a representation closely related to machine code, while still being independent of the concrete target architectures. For maintaining all information during the lowering from LLVM IR to machine code, the tool relies on control-flow relation graphs [12].

The TASKERS task generator is a modified and extended version of the GENE tool [24] and constructs tasks by weaving together different code patterns in a systematic fashion that allows the generator to define the worst-case path. To produce realistic task programs, the code patterns used for this purpose closely resemble the typical building blocks of real-world applications. Amongst other things, this includes input-dependent computations, assignments, function calls, conditional branch statements, as well as different shapes of loops. The fact that most code patterns are parameterizable enables TASKERS to create complex programs in which, for example, the execution of branches depends on the program's input and the bodies of loops may contain other, nested code patterns, possibly even other loops. Prior to the start of the generation process, a TASKERS user is able to influence the composition of the resulting benchmark by configuring the probability with which each code pattern is selected and woven into the program. This way, a user can rely on TASKERS to generate benchmark-system implementations that possess the characteristics of real-time systems from a particular domain (e.g., digital-signal-processing applications).

In contrast to GENE, TASKERS' task generator does not use the path budget as seed for pseudorandomly selecting the next program pattern to weave into the task program. Instead,



■ **Figure 3** Linear correlation between path budget and the resulting task’s WCET.

TASKERS limits the impact small changes to the path budget have on the structure of the generated program, thereby ensuring that two programs resemble each other if they were generated with similar (but different) path budgets. As a key benefit, this approach improves the linearity between the path budget and a task implementation’s WCET, and therefore significantly speeds up the process of creating a program with the requested WCET.

The task generator is integrated into and driven by a Python framework that, in addition, takes care of producing the task set, building the system binary, measuring on the target hardware and postprocessing of the measured values. The final system binary is assembled on basis of *dOSEK* [11], an OSEK-compliant real-time operating system.

5 Evaluation

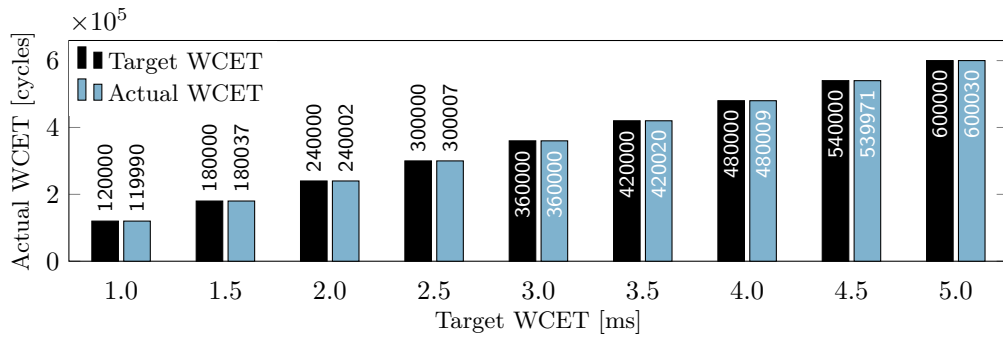
In this section, we move on to the evaluation of TASKERS. First, we assess our approach to correlate desired path budget and actual WCET of the operational code by linear regression. Based on this, we demonstrate that in the worst case the generated tasks meet specifications and otherwise exhibit realistic behavior. Finally, we showcase the performance and worst-case compliance of a whole system and its suitability as a benchmarking baseline.

Experimental Setup

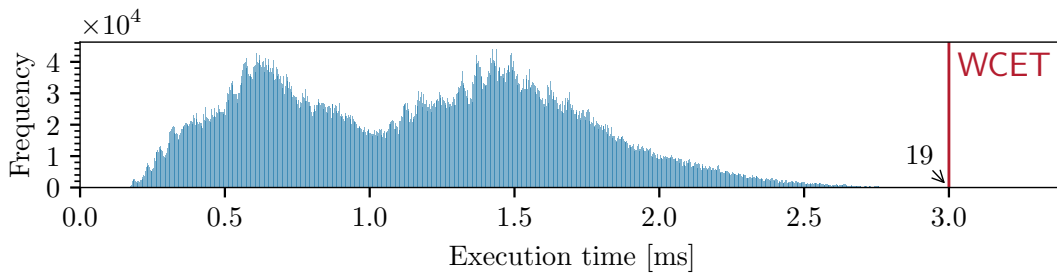
We used an Infineon XMC4500 development board with an ARM Cortex-M4 processor as an available hardware platform with the instruction timing, pipeline behavior, and memory-access latencies being sufficiently predictable and well documented [13]. The 32-bit processor runs at 120 MHz and features a 3-stage pipeline, 1024 KB flash memory with 4 KB instruction cache (2-way set associative, LRU cache-replacement policy). Time measurements were conducted using the cycle-accurate counters (i.e., DWT) provided by the Cortex-M4 core. For the measurements, we applied our system model as described in Section 2. All tasks are generated using the default configuration of TASKERS’ task generator (i.e., all code patterns are enabled and use their default probabilities).

5.1 Correlation of Path Budget and WCET

TASKERS’ task generator exploits the linear correlation between the path budget and the WCET of the resulting operational code to estimate the budget required to generate a fitting task. To confirm the assumption of linear correlation between path budget and a task’s WCET, for our first experiment we generated 20,000 tasks with different path budgets, while retaining the generator’s other parameters, and determined the tasks’ WCETs. Figure 3 illustrates the correlation between the path budget and the tasks’ WCET: Each data



■ **Figure 4** Comparison between generated and target WCETs.



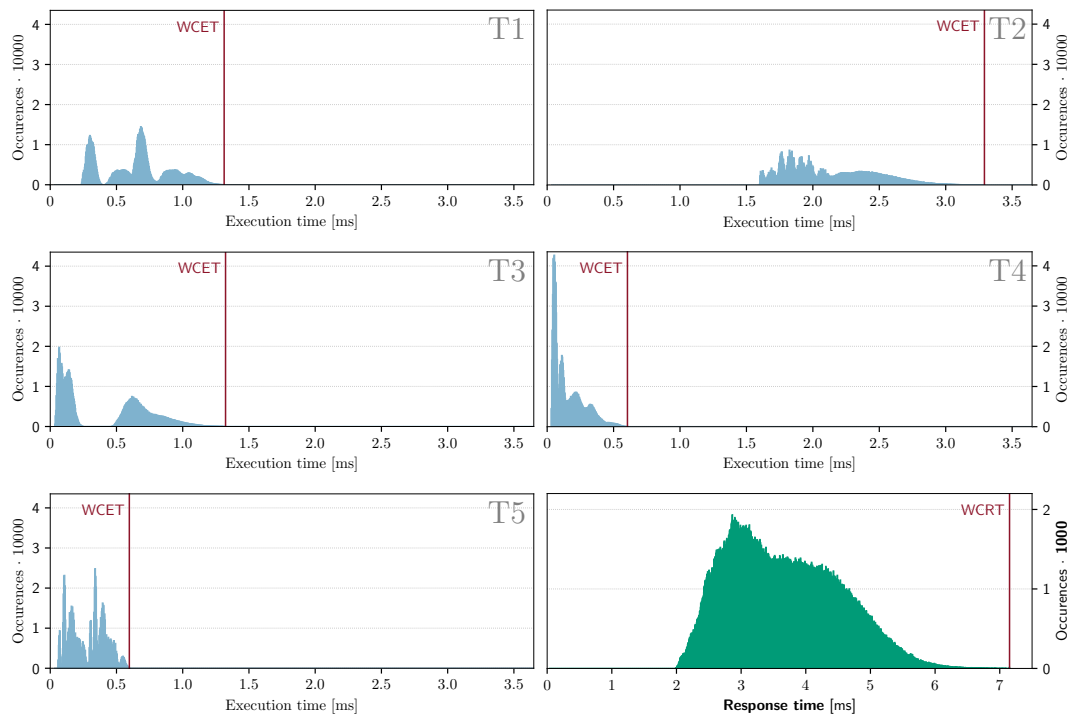
■ **Figure 5** Variation of the execution time for a single task with varying input values.

point (dot) represents a single task whose WCET is plotted against its predefined budget. The regression used by TASKERS to estimate a value for the path budget is given as a line. The adjacency and concentration of the measured data points close to the regression line substantiate the assumption of the linear correlation between path budget and WCET and thus TASKERS' approach to generating tasks with predefined WCETs.

5.2 Temporal Behavior and Realism of Tasks

In a next step, we evaluated the accuracy of the operational code generation itself. Therefore, we generated tasks with budgets in the range from 1 ms to 5 ms and a maximum deviation of 0.1% and measured the actual WCET by triggering the generated worst-case path at runtime. Figure 4 shows the results of this evaluation: For all 9 tasks, the actual WCET fits the predefined budget with great accuracy. The maximum deviation in this experiment was 0.02% or 37 cycles (1.5 ms task), well below the configured threshold of 0.1%. For all practical purposes, TASKERS' code generation accuracy per task is more than sufficient for the composition of whole-system benchmarks.

The tasks generated by TASKERS show a WCET that is defined at generation time and can be triggered by a user-defined input. Moreover, as identified in Section 2, tasks should in practice also show input-dependent variations of their execution times, adding a jitter component to the benchmark. To assess this aspect of the code generation, we measure the 3 ms-task's execution time for $2^{24} = 16,777,216$ different input values; Figure 5 graphs the resulting distribution. First of all, the traversal of the worst-case execution path is triggered by 19 different input values, including the designated worst-case input value used during the generation process. The WCET (i.e., the execution time of the task's worst-case execution path) is 360,000 cycles (3.0 ms at 120 MHz) and matches the predefined WCET of 3.0 ms. Moreover, the task exhibits a wide range (0.2 to 3 ms) and distribution of execution times.



■ **Figure 6** Execution and response times for task set of five tasks T_1, \dots, T_5 .

The variation also indicates the multitude of different paths through the task.

In summary, the generated operational code complies with the preassigned temporal properties with high accuracy and, at the same time, resembles real-world execution behavior.

5.3 Whole-System Temporal Behavior and Worst-Case Path

This final evaluation is devoted to the temporal behavior of a whole system. Therefore, we took as an example a task set consisting of five interdependent tasks and leveraged the knowledge about the generated system to incorporate the overheads induced by the underlying operating system. To stay unbiased, we randomly chose the tasks' WCETs (71,637 cycles, 72,519 cycles, 158,795 cycles, 394,958 cycles, 157,532 cycles for tasks T_1, \dots, T_5) using the UUniFast [2] algorithm. For the generated benchmark system, we measured the individual execution times of all five tasks, as well as the overall response times for different input values. Figure 6 gives the frequencies of the measured execution and response times for $2^{20} = 1,048,576$ randomly selected input values. All tasks T_1, \dots, T_5 exhibit varying execution times, ranging from 4,257 cycles ($\sim 35.5 \mu\text{s}$, T_2) to 394,958 cycles ($\sim 3.3 \text{ ms}$, WCET of T_3), again substantiating the variety of local execution paths. Except for T_1 , each task's execution path depends on the output of the previous task's calculation, the high distribution of execution times indicates a substantial variation of global control-flow paths through the generated system that depends on the respective input and output values.

The longest execution path through the task set (calculated as the sum of the corresponding tasks' execution times) takes $7,129 \mu\text{s}$ and is traversed whenever the predefined worst-case input value is passed to T_1 . Incorporating the overheads induced by the operating system, the system's actual response time is higher. In case of this experiment, the actual worst-case response time measured by TASKERS is $7,151 \mu\text{s}$; $22 \mu\text{s}$ higher than the lower bound (i.e., the summation of individual execution times).

In summary, our evaluation demonstrates that the overall system also conforms to predefined temporal properties and exhibits known WCRTs.

6 Related Work

To our knowledge, TASKERS is the first whole-system generator for benchmarking real-time systems. The generator is able to create tasks with arbitrarily configurable WCETs and complex inter-task dependencies. The closest related work to ours is the task-set-generation approach from de Bock et al. [4]. They use existing benchmarks from the TACLEBENCH suite [8], which are written in C, and execute several programs in a loop in order to achieve a predefined execution time. In contrast to their approach, we do not rely on already available benchmarks but generate each task from scratch using a low-level representation (i.e., LLVM IR) and consequently do not depend on the immutable execution times of existing programs. Generating both the tasks as well as the enclosing system from scratch has the major benefit of enabling TASKERS to insert inter-task dependencies on a fine-grained level. Additionally, TASKERS lays the foundation for automatically creating programs to benchmark memory-consumption and caching-behavior analysis, which cannot be achieved when relying on memory consumption and access patterns from existing benchmark programs written in a high-level programming language. In contrast, using LLVM's intermediate representation results in our benchmarks being more resilient against compiler optimizations compared to the TACLEBENCH suite [24, 25]. The most important difference is that, even though TASKERS' generated tasks have varying execution times, the worst-case path of the overall system is known. Executing this non-trivial path with the predefined worst-case input leads to the WCRT of the task set, thereby providing a baseline for timing-analysis approaches.

Several real-world benchmarks and suites are used for benchmarking real-time system analyses [8, 9, 10, 14, 16, 19]. However, in order to evaluate approaches on a global scale, the ground truth, such as the worst-case path, is inevitable, which can never be safely determined when trying to extract it from existing code [22]. Instead, we solve the problem of baselines by *generating* the entire executable real-time system with known properties.

Tools exist that output the parameters of tasks, such as UUniFast [2] or the task-set generator from Emberson et al. [7]. These generators are used for the theoretic evaluation of scheduling algorithms since they do not produce any executable code. TASKERS fills this gap by considering these task parameters and producing an executable system for comprehensive evaluations of timing-analysis approaches on real embedded platforms.

7 Conclusion & Future Work

In this paper, we presented the TASKERS generator that produces real-time systems with known properties for the purpose of benchmarking timing-analysis approaches. TASKERS generates entire systems that execute precedence-constrained tasks with configurable WCETs, which are scheduled by their fixed priorities. Since all relevant knowledge about the generated systems is either directly available or can be determined by measurement (e.g., a system's WCRT), TASKERS enables a comprehensive evaluation of timing and scheduling-analysis techniques on an absolute scale. Our evaluation confirms TASKERS' accuracy, showing that the actual WCETs of generated tasks differ less than 0.1% from the WCETs requested.

As part of future work, we will extend the current prototype of TASKERS to make it more aware of the target platform's micro-architectural properties, and thereby further

challenge analysis approaches. Specifically, we will integrate challenging code patterns to generate benchmarks that target whole-system cache and pipeline analyses.

The source code of TASKERS is available at: <https://gitlab.cs.fau.de/taskers>

References

- 1 M. Bambagini, M. Marinoni, H. Aydin, and G. Buttazzo. Energy-aware scheduling for real-time systems: A survey. *ACM TECS*, 15(1), 2016.
- 2 E. Bini and G. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30, 2005.
- 3 B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser. Timing analysis of a protected operating system kernel. In *Proc. of RTSS '11*, 2011.
- 4 Y. De Bock, S. Altmeyer, J. Broeckhove, and P. Hellinckx. Task-set generator for schedulability analysis using the taclebench benchmark suite. In *Proc. of EWiLi '16*, 2016.
- 5 A. Burns and R. Davis. Mixed criticality systems - a review. Technical report, Department of Computer Science, University of York, 2018.
- 6 C. Dietrich, P. Wägemann, P. Ulbrich, and D. Lohmann. SysWCET: Whole-system response-time analysis for fixed-priority real-time systems. In *Proc. of RTAS '17*, 2017.
- 7 P. Emberson, R. Stafford, and R. I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *Proc. of WATERS '10*, 2010.
- 8 H. Falk et al. TACLeBench: A benchmark collection to support worst-case execution time research. In *Proc. of WCET '16*, 2016.
- 9 J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks: Past, present and future. In *Proc. of WCET '10*, 2010.
- 10 M. Guthaus et al. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of WWC '01*, 2001.
- 11 M. Hoffmann, F. Lukas, C. Dietrich, and D. Lohmann. dOSEK: The design and implementation of a dependability-oriented static embedded kernel. In *Proc. of RTAS '15*, 2015.
- 12 B. Huber, D. Prokesch, and P. Puschner. Combined WCET analysis of bitcode and machine code using control-flow relation graphs. In *Proc. of LCTES '13*, 2013.
- 13 Infineon Technologies AG. XMC4500 reference manual, 2012.
- 14 F. Kluge, C. Rochange, and T. Ungerer. Emsbench: Benchmark and testbed for reactive real-time systems. *Leibniz Trans. on Embedded Systems*, 4(2), 2017.
- 15 J. Knoop, L. Kovács, and J. Zwirchmayr. WCET squeezing: On-demand feasibility refinement for proven precise WCET-bounds. In *Proc. of RTNS '13*, 2013.
- 16 S. Kramer, D. Ziegenbein, and A. Hamann. Real world automotive benchmarks for free. In *Proc. of WATERS '15*, 2015.
- 17 C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of CGO '04*, 2004.
- 18 M. Lv, N. Guan, Y. Zhang, Q. Deng, G. Yu, and J. Zhang. A survey of WCET analysis of real-time operating systems. In *Proc. of ICESSE '09*, 2009.
- 19 F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun, and M. De Michiel. PapaBench: A free real-time benchmark. In *Proc. of WCET '06*, 2006.
- 20 OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, February 2005.
- 21 Jan Reineke. *Caches in WCET Analysis: Predictability, Competitiveness, Sensitivity*. PhD thesis, Saarland University, 2008.
- 22 H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. of the AMS*, 1953.

- 23 P. Ulbrich, R. Kapitza, C. Harkort, R. Schmid, and W. Schröder-Preikschat. I4Copter: An adaptable and modular quadrotor platform. In *Proc. of SAC '11*, 2011.
- 24 P. Wägemann, T. Distler, C. Eichler, and W. Schröder-Preikschat. Benchmark generation for timing analysis. In *Proc. of RTAS '17*, 2017.
- 25 P. Wägemann, T. Distler, P. Raffeck, and W. Schröder-Preikschat. Towards code metrics for benchmarking timing analysis. In *Proc. of RTSS WiP '16*, 2016.

Experimental Evaluation of Cache-Related Preemption Delay Aware Timing Analysis

Darshit Shah


Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
s8dashah@stud.uni-saarland.de

Sebastian Hahn

Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
sebastian.hahn@cs.uni-saarland.de

Jan Reineke

Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
reineke@cs.uni-saarland.de

 <https://orcid.org/0000-0002-3459-2214>

Abstract

In the presence of caches, preemptive scheduling may incur a significant overhead referred to as *cache-related preemption delay* (CRPD). CRPD is caused by preempting tasks evicting cached memory blocks of preempted tasks, which have to be reloaded when the preempted tasks resume their execution.

In this paper we experimentally evaluate state-of-the-art techniques to account for the CRPD during timing analysis. We find that purely synthetically-generated task sets may yield misleading conclusions regarding the relative precision of different CRPD analysis techniques and the impact of CRPD on schedulability in general. Based on task characterizations obtained by static worst-case execution time (WCET) analysis, we shed new light on the state of the art.

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

Keywords and phrases real-time systems, timing analysis, cache-related preemption delay

Digital Object Identifier 10.4230/OASICS.WCET.2018.7

Funding This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the project PEP, and by the Saarbrücken Graduate School of Computer Science which receives funding from the DFG as part of the Excellence Initiative of the German Federal and State Governments.

Acknowledgements We thank Tina Jung who contributed the ECB and UCB analyses to LLVMTA.

1 Introduction

In real-time systems, it is often necessary to schedule tasks preemptively in order to meet all tasks' deadlines. Most work on preemptive scheduling is based on the assumption that the overhead incurred by preemptions is negligible and may thus be subsumed into the worst-case execution time of each task. When tasks are executed on complex microarchitectures with caches, this assumption is problematic: Preempting tasks may alter the state of the cache, leading to an increased execution time of preempted tasks once they are resumed, because their data has been evicted from the cache and needs to be reloaded from main memory. The additional execution time due to such reloads is known as *cache-related preemption delay* (CRPD).



© Darshit Shah, Sebastian Hahn, and Jan Reineke;
licensed under Creative Commons License CC-BY

18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018).

Editor: Florian Brandner; Article No. 7; pp. 7:1–7:11

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

As the CRPD depends both on the “low-level” cache aspect and on “higher-level” scheduling decisions, it is tackled by a combination of low-level static analysis, characterizing each task’s “cache footprint”, and CRPD-aware response-time analysis, bounding a task’s response time using the low-level characterizations. Altmeyer et al. [1] provide an overview of the state-of-the-art techniques to account for CRPD during response-time analysis. To experimentally evaluate the different CRPD-aware response-time analyses, Altmeyer et al. [1] use synthetically-generated task sets with synthetically-generated task characteristics.

In this paper, we experimentally evaluate the state of the art concerning CRPD-aware timing analysis to gain further insights into where future research on CRPD may be profitable. To this end, we attempt to answer the following questions:

- Can we reproduce the results obtained in the experimental evaluation of Altmeyer et al. [1] based on synthetic task sets?
- Do we obtain similar per-task characteristics (worst-case execution time (WCET) values, number of Evicting Cache Blocks (ECBs), and number of Useful Cache Blocks (UCBs)) as Altmeyer et al. [1] based on our low-level analysis toolchain?
- If we base the experimental evaluation on task characterizations obtained by static WCET analysis, do we observe similar trends as those observed in [1] based on synthetic task sets? If not, why? Related to the previous question: Are the parameters for the synthetic task set generation meaningful?

Our paper is structured as follows: We summarize the background concerning caches and CRPD-aware timing analysis in Section 2. In Section 3 we discuss relevant details of our analysis implementation. Then, in Section 4 we present the results of our experimental evaluation, partially answering some of the questions listed above. We conclude the paper with a summary of our findings in Section 5.

2 Background and Related Work

2.1 Caches

Caches are small but fast memories that store a subset of the main memory’s contents to bridge the latency gap between processors and DRAM-based main memory.

To profit from spatial locality and to reduce management overhead, main memory is logically partitioned into a set of *memory blocks* of a certain size. Blocks are cached as a whole in cache lines of the same size. When accessing a memory block, the cache logic has to determine whether the block is stored in the cache, a *cache hit* or not, a *cache miss*.

To enable an efficient look-up, each block can only be stored in a small number of cache lines. For this purpose, caches are partitioned into equally-sized *cache sets*. The size of a cache set is called the *associativity* of the cache. Caches of associativity one are called *direct mapped*. Most work concerning CRPD-aware response time analysis has been conducted in the context of direct-mapped caches. Such caches are also the focus of this work.

2.2 Timing Analysis for Preemptive Scheduling

Timing analysis is traditionally separated into two phases:

1. Worst-case execution time (WCET) analysis, which determines bounds on each task’s execution time. Usually C_i denotes the WCET bound of task τ_i .
2. Response-time analysis (RTA), which determines bounds on each task’s response time under a particular scheduling algorithm; based on the tasks’ WCET bounds, minimum inter-arrival times (also referred to as periods), denoted by T_i , and release jitter, denoted by J_i .

If no task's response time may exceed its relative deadline D_i , then the task set is determined to be schedulable.

Traditional response-time analysis assumes that preemptions are free, i.e., context switches are performed instantaneously and the execution times of tasks are not affected by the execution of preempting tasks. For fixed-priority preemptive scheduling, the least solution of the following recursive equation [3,12] then determines a task's response time:

$$R_i = C_i + \sum_{j \in \text{hp}(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil \cdot C_j, \quad (1)$$

where $\text{hp}(i)$ denotes the indices of those tasks that have a higher priority than task τ_i .

Unfortunately, context switches cannot be performed instantaneously for several reasons: 1. The scheduler takes some time to select the next task to execute. 2. Upon a context switch, the hardware needs to save the contents of registers and restore the contents of the task whose execution is to be resumed. This process also results in flushing the pipeline. It is commonly assumed that the cost of these two actions can be bounded by a constant, which can then be taken into account by appropriately inflating each task's WCET bound.

In the presence of caches, however, preemptive scheduling may incur an additional overhead, called *cache-related preemption delay* (CRPD): The execution time of preempted tasks may be prolonged due to additional cache misses caused by cache evictions of preempting tasks.

To account for CRPD, Equation (1) can be extended by $\gamma_{i,j}$ representing the preemption cost due to each job of a higher-priority preempting task τ_j executing within the worst-case response time of task τ_i [6]:

$$R_i = C_i + \sum_{j \in \text{hp}(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil \cdot (C_j + \gamma_{i,j}). \quad (2)$$

Note, that the response time calculation inherently relies on timing compositionality [10] of the cache-related preemption effects. Recent work in the area of multi-core timing analysis [9] has shown that the compositionality assumption is often violated even on simple hardware platforms. However, the analysis techniques of [9] could be adopted to still allow for a compositional reasoning.

2.3 Characterizing a Task's Cache Footprint

To bound $\gamma_{i,j}$ one needs to bound the number of additional cache misses in preempted tasks τ_i due to the execution of preempting tasks. The number of such cache misses, depends on the memory-access behavior of both the preempted task and its preempting tasks.

To characterize preempting tasks, Busquets-Mataix et al. [6] introduced the notion of ECBs: A memory block is an ECB of task τ_j if it may be accessed during τ_j 's execution. For the computation of cache-related preemption delays the precise identity of a memory block is irrelevant. What is important is which cache set an ECB maps to, which is where it may potentially evict cached memory blocks of a preempted task. Further, in case of direct-mapped caches, if two or more ECBs map to the same cache set, they may not do a greater damage than an individual ECB mapping to this cache set. Thus, in case of direct-mapped caches – which we focus on in this paper – one may characterize the set of ECBs of a task by a set $ECB_j \subseteq \{0, \dots, N-1\}$, capturing the subset of cache sets that a task's evicting cache blocks map to.

To characterize preempted tasks, Lee et al. [13] introduced the notion of UCBs: A memory block m is a UCB of task τ_i if there is a program point P within τ_i , such that m may be cached at P and m may be reused at a later program point P' , which may be reached from P without eviction of memory block m along the execution from P to P' . Intuitively, only useful cache blocks may result in additional cache misses due to preemptions. As there may be at most one useful cache block in each cache set of a direct-mapped cache at any point in time, the set of UCBs of a task may be represented by a set $UCB_i \subseteq \{0, \dots, N - 1\}$, capturing the cache sets that a task's useful cache blocks map to.

Later, Altmeyer and Maiza [2] introduced the notion of *definitely-cached useful cache blocks* (DC-UCBs): A memory block m is a DC-UCB of task τ_i if there is a program point P within τ_i , such that m must be cached at P and m may be reused at a later program point P' , which may be reached from P without eviction of memory block m along the execution from P to P' , and, crucially, m is considered to be a cache hit at P' by the WCET analysis. The set of DC-UCBs is always a subset of the set of UCBs. The observation in the definition of DC-UCBs is that CRPD analysis needs to only account for preemption-induced cache misses that are not already conservatively accounted for during WCET analysis.

2.4 CRPD-Aware Response-Time Analysis for Fixed-priority Scheduling

Based on the sets of ECBs and UCBs of all tasks, there are different ways of defining $\gamma_{i,j}$, such that the response times of tasks are correctly bounded by solutions of (2). In the following, we briefly summarize the six state-of-the-art approaches from Altmeyer et al. [1] that apply to direct-mapped caches. More details can be found in [1]. The methods can be extended to be applied to set-associative caches with LRU replacement, but not to caches with FIFO or PLRU replacement [5]. Those six approaches follow from two different interpretations of $\gamma_{i,j}$ that differ in case of nested preemptions:

1. “Effect of the preempting task”: In this case $\gamma_{i,j}$ bounds the cost of additional misses in the preempted tasks due to execution of the preempting task τ_j .
2. “Effect on the immediately preempted task”: In this case $\gamma_{i,j}$ bounds the cost of additional misses in the task immediately (i.e. not in a nested fashion) preempted by τ_j , due to τ_j 's execution *and* the execution of higher-priority tasks which may in turn have preempted τ_j .

Effect of the Preempting Task

Busquets-Mataix [6] and later Tomiyama and Dutt [16] used the number of ECBs of the preempting task to bound the preemption cost, in what Altmeyer et al. [1] termed the *ECB-Only* approach:

$$\gamma_{i,j}^{ECB} = \text{BRT} \cdot |\text{ECB}_j|, \quad (3)$$

where BRT denotes the *block reload time*, i.e., the time to fetch one memory block from main memory into the cache.

Tan and Mooney [15] improved upon *ECB-Only* by also considering the set of UCBs of all tasks that may be affected by the preemption by τ_j . This approach is termed *UCB-Union* in [1]. Altmeyer et al. [1] introduced the *UCB-Union-Multiset* approach, which improves upon *UCB-Union* by taking into account how often different tasks may preempt each other based on their minimum inter-arrival times.

Effect on the Immediately-Preempted Task

Lee et al. [13] introduced the *UCB-Only* approach, which bounds cost of a preemption in the immediately-preempted task by considering its UCBs:

$$\gamma_{i,j}^{UCB} = \text{BRT} \cdot \max_{\forall k \in \text{aff}(i,j)} \{ |UCB_k| \}, \quad (4)$$

where $\text{aff}(i, j) = \text{hep}(i) \cap \text{lp}(j)$ is the set of tasks that have a lower priority than task τ_j but a higher-or-equal priority than task τ_i , the task under analysis. Thus $\text{aff}(i, j)$ is the set of tasks that task τ_j may immediately preempt during task τ_i 's response time.

Altmeyer et al. [1] later introduced the *ECB-Union* and the *ECB-Union-Multiset* approaches, which additionally take into account the ECBs of the preempting tasks, and the number of times tasks may preempt each other based on their minimum inter-arrival times.

As the *UCB-Union-Multiset* and the *ECB-Union-Multiset* approaches are incomparable, they may be combined by taking the minimum response time of the two approaches for each task, to yield an approach, coined *Combined-Multiset*, that dominates the two [1].

3 Implementation

In this section we describe the tools used for the experiments presented later in this paper. This includes the tools for analysis of tasks to determine task characteristics, for generation of synthetic task sets and for running the schedulability analysis.

3.1 Obtaining Task Characteristics

We use our low-level timing analysis tool LLVMTA [9] to compute not only the worst-case execution time bound for a given task, but also its preemption-related characteristics: ECBs, UCBs, and DC-UCBs. LLVMTA supports the detailed microarchitectural analysis of different processors. In this paper, we use the model of a conventional in-order pipeline with five stages [11], static branch prediction, and native support for floating-point instructions. The main memory is accessed via separate instruction cache and data scratchpad. We employ standard must- and may-analyses [7] to predict the instruction cache behavior. Similar to [2], our analyses are context sensitive: they (virtually) peel the first iteration of loops and distinguish the different call sites for each function. This is important to achieve precise analysis results [7].

The microarchitectural analysis results in a *microarchitectural execution graph* [9] whose nodes correspond to abstract microarchitectural states and whose edges represent the possible execution flow. To obtain the worst-case execution time bound, we use an integer linear program to calculate the longest path through the microarchitectural execution graph – also known as implicit path enumeration [14].

This microarchitectural execution graph also contains detailed information about the memory accesses initiated in the pipeline. Unlike the control-flow graph of a program, it explicitly includes speculative accesses triggered by branch prediction or even access reorderings in out-of-order pipelines. To ensure soundness, we thus perform the ECB and DC-UCB analysis on this microarchitectural execution graph rather than on the control-flow graph of the program. Our implementation follows the data-flow description of the analysis in [2].

The set of useful cache blocks is a property of a particular program point. To be able to use the sets in an efficient way in the schedulability analysis, we need to combine the program-point-sensitive results. In accordance with [1], we calculate the set of those cache sets that exhibit a useful cache block at any program point.

Instructions that share a single cache line are usually executed consecutively in straight-line code fragments (spatial locality). As a consequence, the cache line is useful at the program point between two instruction of that cache line. In the program-point-insensitive result, almost every cache line in the program is (definitely-cached) useful due to spatial locality if the line size exceeds the word size. Similar to [2], blocks that are useful only due to immediate spatial locality can be ignored in the low-level analysis, if they are compensated for by one additional cache reload per preemption in the schedulability analysis.

3.2 Task Set Generation and Schedulability Analysis

The generation of synthetic task sets and the CRPD-aware schedulability analysis is done using a new tool that we developed using the Rust programming language. This tool performs CRPD-aware schedulability analyses of [2] on synthetically generated task sets using either the task characteristics provided by LLVMTA or by synthetically generating them.

4 Evaluation

In this section we experimentally evaluate the differences between the various approaches discussed in Section 2.4 using both synthetically-generated task characteristics and task characteristics derived through our WCET analysis tool, LLVMTA. In the latter case, we use the programs from the Mälardalen test bench [8] as tasks.

As in most previous work on cache-related preemption delay, including [1, 2], we assume a direct-mapped instruction cache and a data scratchpad. The size of the data scratchpad is sufficient to not cause preemption-induced reloads. To compare our results to [1], we use a cache with a line size of 8 bytes and 256 sets, i.e. a total size of 2 kB.

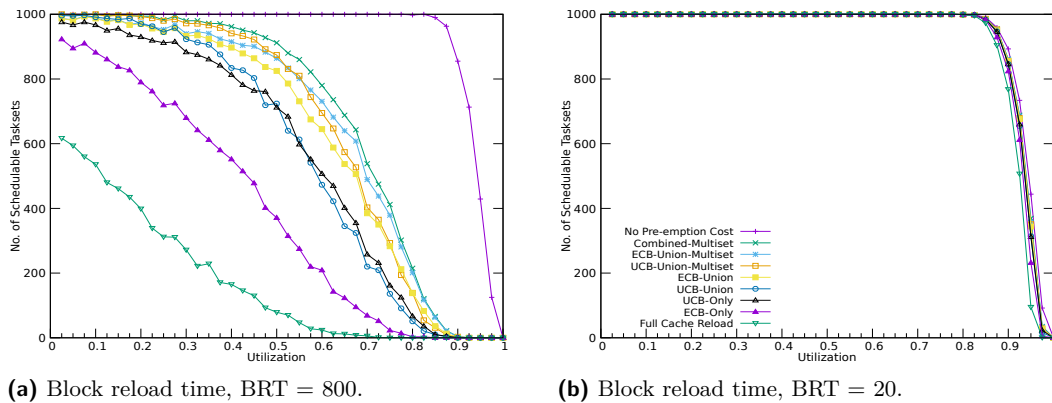
In each of our evaluations, we have compared the six CRPD approaches listed in Section 2.4, along with the optimistic *No Preemption Cost* case where preemptions are considered to be completely free and the extremely pessimistic, *Full Cache Reload* case where each preemption causes the entire cache of the preempted task to be reloaded.

4.1 Timing Analysis With Synthetically-Generated Task Characteristics

In order to replicate the results of [1], we used our tool to generate synthetic task sets with the same parameters as in the original experiment. The task characteristics such as WCET, UCB, and ECB values are also synthetically generated following the methodology used in [1]. Since, the clock frequency is not mentioned in the original study, we assumed a frequency of 100 Mhz¹.

For the experimental evaluation, 1000 task sets with 10 tasks each were generated for every utilization value from 0.025 to 1.000 in steps of 0.025 using the UUnifast [4] algorithm. The task periods were generated according to a log-uniform distribution with the minimum period as 500,000 cycles (5 ms) and a maximum period of 50,000,000 cycles (500 ms). The cost of a cache miss, also known as the Block Reload Time, is considered to be 800 cycles (8 μ s). The cache utilization (CU) of a task is the ratio of the number of ECBs of a task to the total number of cache sets available. The cache utilizations for the tasks in a task

¹ As all task parameters are given in terms of wall-clock time, the particular processor frequency has a negligible effect on the experimental results.



■ **Figure 1** Results using synthetic task sets with synthetically-generated task characteristics.

set were generating using UUnifast², considering a total CU = 10. The number of ECBs of a task are then computed using the generated cache utilization values. The ratio of the number of UCBs of a task to the number of ECBs is defined as the reuse factor (RF). For this experiment, the reuse factor for each task was picked uniformly at random within the range [0, 0.3]. All of these values are taken from the parameters of the original study.

For the results of the schedulability tests, see Figure 1a. Our first observation is that the resulting schedulability graph closely matches the corresponding graph in Figure 9 of [1], which confirms the correctness of the independent implementations of schedulability analyses used in both papers.

The latency of actual main memory modules³ are at least an order of magnitude lower than the 8 μ s (800 cycles) used in the original experiment. In [2], Altmeyer et. al. use a memory latency of just 4 cycles for their WCET analysis and yet the generation of synthetic task characteristics uses a latency of 800 cycles. To gauge the effect of a more realistic memory latency, we re-ran the experiment, however this time with a lower memory latency of 20 cycles. The results of this experiment can be seen in Figure 1b. As is clearly evident from the figure, reducing the memory latency almost completely eliminates the effect of CRPD on overall schedulability. We must emphasize here that we do not claim that CRPD has a negligible effect on the overall schedulability, but rather that the methods used to synthetically generate task characteristics provide misleading results. With a lower memory latency, we see that even in the case of having to pay the penalty of a full cache refresh for each preemption, the schedulability of the task sets closely follows the case where preemption is considered to be free. The reason for such behavior is that the generation of synthetic task characteristics does not take into account the fact that both the WCET and the CRPD depend on the memory latency, and are thus correlated. However, in our experiment, decreasing the memory latency reduces the CRPD without similarly reducing the execution time of the tasks at all. At this point, the WCET almost completely dominates the response time of the task.

² UUnifast may generate values greater than one, indicating that the ECBs fill the entire cache. The number should be capped to the number of cache sets. However, for the computation of UCBs, the original value of ECBs is used.

³ As example, consider this automotive SDRAM https://www.micron.com/~/media/documents/products/data-sheet/dram/mobile-dram/low-power-dram/lpddr/256mb_x8x16_at_ddr_t66a.pdf

4.2 Comparison of Task Characteristics

In this section, we compare the per-task characteristics provided in [1, 2] for a subset of the Mälardalen benchmarks with the numbers calculated using LLVMTA. The results are shown in Table 1 for the subset of benchmarks analyzed in [1, 2] in the upper half and for the remaining benchmarks in the lower half. The apparent gap between the WCET bounds is mostly explained by different memory latencies. While we use a memory latency of 20 cycles throughout the paper, they use a low latency of only 4 cycles as described in [2]. Furthermore, the different compilers (gcc versus clang+LLVM), hardware platforms, and analysis settings (e.g. loop bound annotations) contribute to this gap. The lower WCET bound (and significantly lower ECB value) – compared to [1] – for some benchmarks, such as `sqr` and `qurt`, are explained by the native floating point support of our pipeline model in contrast to software emulation, which is assumed in [1].

Taking the above differences in the underlying model into account, our ECB values and the cache utilizations resemble the ones provided by Altmeyer et al. [1].

As described in Section 3.1, column *DC-UCB* of Table 1 shows the size of the set containing all cache sets that exhibit a useful cache block at some program point. In addition, column *Max DC-UCB* shows the maximum number of cache blocks that are (definitely-cached) useful at a single program point. This number can be used to improve the *UCB-Only* approach, which we call *UCBMax-Only*. Note that *DC-UCB* and *Max DC-UCB* results from different schemes to aggregate program-point specific information of the static cache analysis. Thus, there is no reason in considering *UCBMax-Only* in Section 4.1 when task characteristics are synthetically generated.

Unlike the ECB values, our DC-UCB values differ significantly from the values in [1]. Compiling our benchmarks with optimizations⁴ reduces the number of DC-UCBs, but also the number of ECBs which resulted in significant gaps for ECBs and DC-UCBs. The only way to get similarly low DC-UCB values using our toolchain is to disable the (virtual) loop peeling which worsens the must cache analysis and thus decreases the number of DC-UCBs. However, according to [2], their numbers were obtained with loop peeling enabled. As a result of the higher number of DC-UCBs in our analysis, we also see a higher value of the reuse factor in our tasks.

4.3 Timing Analysis With Analysis-Derived Task Characteristics

In this section, we conduct schedulability experiments with task characteristics derived using LLVMTA rather than generated synthetically. For deriving task characteristics, we used the programs in the Mälardalen suite and derived the WCET and the ECB and DC-UCB sets for each task as described in Section 3.1 and discussed in Section 4.2. These characteristics were then used to generate synthetic task sets. In addition to the approaches outlined in Section 4, in this section, we also consider the *UCBMax-Only* approach which was introduced in Section 4.2.

As before, 1000 task sets were generated for each utilization value from 0.025 to 1.000 in steps of 0.025. Each task set consists of 10 randomly selected tasks from the set of tasks found in Table 1. The utilization U_i of each of the 10 tasks within a task set was generated using the UUnifast algorithm [4] and the periods were computed based on the following equation: $T_i = C_i/U_i$. Since the WCETs of the tasks were derived through analysis, we did not have control over the range of task periods. Implicit task deadlines, i.e. $D_i = T_i$, were

⁴ It is not specified whether the programs in [1] have been compiled with or without optimizations.

■ **Table 1** Comparison of task characteristics. In parenthesis: results without virtual loop peeling. * denotes use of floating-point calculations.

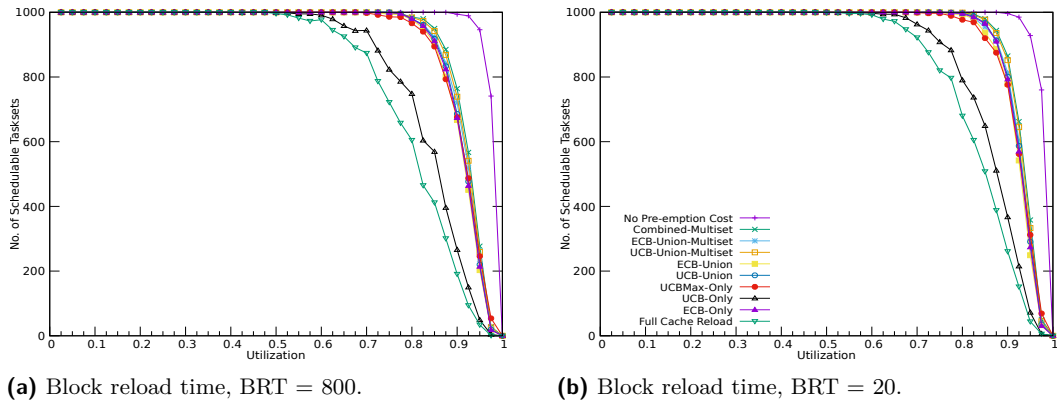
	WCET		ECB		DC-UCB		Max	CU		RF		
	Theirs	Ours	Theirs	Ours	Theirs ⁵	Ours	DC-UCB	Theirs	Ours	Theirs	Ours	
bs	445	3052	35	43	5	23	(9)	20	0.137	0.168	0.143	0.535
bsort100	1567222	3146185	62	57	8	40	(11)	30	0.242	0.223	0.129	0.702
crc	290782	1621246	144	127	14	63	(28)	35	0.562	0.496	0.097	0.496
fibcall	1351	8406	24	28	5	16	(6)	16	0.094	0.109	0.208	0.571
fir	29160	12406071	105	90	9	41	(16)	22	0.410	0.352	0.086	0.456
insertsort	6573	11291	41	29	10	16	(5)	15	0.160	0.113	0.244	0.552
matmult	742585	1447379	100	85	23	51	(26)	31	0.391	0.332	0.230	0.600
ns	43319	126865	64	55	13	37	(12)	34	0.250	0.215	0.203	0.673
qsort-exam *	22146	163089	170	142	15	83	(38)	39	0.664	0.555	0.088	0.585
qurt *	214076	71655	484	130	14	40	(32)	26	1.891	0.508	0.029	0.308
select *	17088	6306	151	159	15	73	(35)	55	0.590	0.621	0.099	0.459
sqrt *	39962	22436	477	53	14	21	(13)	12	1.863	0.207	0.029	0.396
adpcm	-	82368867	-	256	-	229		108	-	1.000	-	0.895
cnt	-	127558	-	123	-	58		44	-	0.480	-	0.472
compress	-	1098331	-	247	-	149		57	-	0.965	-	0.603
cover	-	71967	-	256	-	38		15	-	1.000	-	0.148
edn	-	739514	-	256	-	220		120	-	1.000	-	0.859
expint	-	2144875	-	113	-	63		36	-	0.441	-	0.558
fdct	-	10258	-	126	-	113		62	-	0.492	-	0.897
fft1 *	-	257657	-	222	-	154		63	-	0.867	-	0.694
janne_complex	-	33778	-	39	-	28		27	-	0.152	-	0.718
jfdctint	-	21742	-	132	-	122		54	-	0.516	-	0.924
lcdnum	-	6129	-	50	-	14		10	-	0.195	-	0.280
lms *	-	10793664	-	242	-	134		38	-	0.945	-	0.554
ludcmp *	-	116312	-	210	-	168		44	-	0.820	-	0.800
minver *	-	67157	-	256	-	178		47	-	1.000	-	0.695
ndes	-	1050167	-	253	-	178		38	-	0.988	-	0.704
nsichneu	-	201969	-	256	-	183		2	-	1.000	-	0.715
prime	-	7726328	-	79	-	50		38	-	0.309	-	0.633
st *	-	3763684	-	192	-	95		52	-	0.750	-	0.495
statemate	-	41776	-	256	-	111		2	-	1.000	-	0.434
ud	-	349120	-	188	-	161		42	-	0.734	-	0.856

considered and tasks priorities were assigned in deadline-monotonic order. We considered two values for the block reload time, BRT: 20 cycles and 800 cycles, which are used both in the calculation of the WCET and the CRPD bounds. The resulting schedulability graphs can be found in Figures 2b and 2a respectively.

The first observation is that the results in Figures 2a and 2b look very similar; very much unlike those in Figures 1a and 1b. As the BRT value is taken into account *both* during WCET and during CRPD analysis, the WCET and CRPD values in the BRT=800 case are both about 40 times higher than in the BRT=20 case. As the task periods are generated based on the WCET values, the periods are similarly 40 times higher on the average, and thus the relative impact of the preemptions is essentially the same in both cases.

In contrast to the results based on synthetically-generated task characteristics, the *UCB-Only* approach performs considerably worse than the other approaches, in particular much worse than *ECB-Only*. This is due to the higher number of DC-UCBs in our analysis as we

⁵ Note, that the authors of [1] refer to DC-UCBs as UCBs in their evaluation.



(a) Block reload time, BRT = 800.

(b) Block reload time, BRT = 20.

■ **Figure 2** Results using synthetic task sets with experimentally determined task characteristics.

have explained in Section 4.2. The *UCBMax-Only* approach however, performs similarly well as the rest of the approaches.

In contrast to the results based on synthetically-generated task characteristics, the *ECB-Only* approach performs similarly to the more sophisticated approaches that take into account both ECBs and DC-UCBs. This happens due to the fact that the high number of DC-UCBs does not help to improve the performance of these other approaches much.

Comparing Figure 2b with Figure 1b, we observe a wider gap between the optimistic *No Preemption Cost* and the pessimistic *Full Cache Reload* approaches. This implies that there is a greater CRPD overhead in the task sets than is apparent in the case of synthetically-generated task characteristics at BRT=20, but less than at BRT=800.

5 Conclusions and Open Questions

In this paper, we experimentally evaluate state-of-the-art CRPD-aware timing analysis approaches. First, we reproduce the response-time analysis results of prior work [1] for purely synthetic task sets based on our own independent implementation of schedulability analyses, thereby increasing confidence in the correctness of both implementations.

Choosing a more realistic value for the block reload time than prior work, we obtain very different results that superficially seem to indicate that the CRPD overhead would be negligible. To further investigate this issue, we next obtain task characterizations using static analysis for all benchmarks in the Mälardalen suite. Our analysis results closely match prior analysis results in case of ECBs, but, surprisingly, not in case of DC-UCBs. Based on these task characterizations, we generate synthetic task sets and use them to evaluate the state-of-the-art approaches. We find that

1. The effect of the cache-related preemption delay on the overall response times – and thus overall schedulability – is not negligible, but smaller than suggested by the original results in [1].
2. The block reload time does not significantly influence the impact of the CRPD on overall schedulability, *if* it is taken into account both during WCET and CRPD analysis.
3. The simple *ECB-Only* approach is competitive with the more sophisticated alternatives, such as the *UCB-Union-Multiset* approach. This is due to the fact that our static analysis classifies more blocks as DC-UCBs than prior work.

Our findings lead us to conclude that *either* synthetic task sets should be generated from

characteristics derived by low-level analysis of actual programs rather than synthetically-generated characteristics; *or* better task characteristics generators are required that do not miss important dependencies between different characteristics such as WCET and number of ECBs/UCBs. Analogously, we suspect existing synthetically-generated task sets are not representative of real-world system-level workloads. Thus, obtaining real-world system-level benchmarks together with the low-level characteristics of the involved tasks is an important future step.

References

- 1 Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Syst.*, 48(5):499–526, 2012.
- 2 Sebastian Altmeyer and Claire Maiza. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *Journal of Systems Architecture*, 57:707–719, August 2011.
- 3 Neil C. Audsley et al. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 1993.
- 4 Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005. doi:10.1007/s11241-005-0507-9.
- 5 Claire Burguière, Jan Reineke, and Sebastian Altmeyer. Cache-related preemption delay computation for set-associative caches—pitfalls and solutions. In *WCET*, June 2009.
- 6 José V. Busquets-Mataix et al. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *RTAS*, pages 204–212, 1996.
- 7 Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. *Sci. Comput. Program.*, 35(2):163–189, 1999. doi:10.1016/S0167-6423(99)00010-6.
- 8 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks: Past, present and future. In *WCET*, pages 136–146, 2010.
- 9 Sebastian Hahn, Michael Jacobs, and Jan Reineke. Enabling compositionality for multicore timing analysis. In *RTNS*, pages 299–308, 2016. doi:10.1145/2997465.2997471.
- 10 Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards compositionality in execution time analysis: definition and challenges. *SIGBED Review*, 12(1):28–36, 2015. doi:10.1145/2752801.2752805.
- 11 John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann, 2012.
- 12 Mathai Joseph and Paritosh Pandya. Finding response times in real-time system. *The Computer Journal*, 29(5), 1986.
- 13 Chang-Gun Lee et al. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.
- 14 Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, & Tools for Real-Time Systems (LCT-RTS)*, pages 88–98, 1995. doi:10.1145/216636.216666.
- 15 Yudong Tan and Vincent John Mooney III. Timing analysis for preemptive multitasking real-time systems with caches. *ACM Trans. Embedded Comput. Syst.*, 6(1):7, 2007. doi:10.1145/1210268.1210275.
- 16 Hiroyuki Tomiyama and Nikil D. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *CODES*, pages 67–71, 2000.

Embedded Program Annotations for WCET Analysis

Bernhard Schommer

Saarland Informatics Campus
Saarland University Building E1.3, D-66123 Saarbrücken, Germany
schommer@absint.com

Christoph Cullmann

AbsInt Angewandte Informatik GmbH
Science Park 1, D-66123 Saarbrücken, Germany
cullmann@absint.com

Gernot Gebhard

AbsInt Angewandte Informatik GmbH
Science Park 1, D-66123 Saarbrücken, Germany
gebhard@absint.com

Xavier Leroy

INRIA Paris
2 rue Simone Iff, 75589 Paris, France
xavier.leroy@inria.fr

Michael Schmidt

AbsInt Angewandte Informatik GmbH
Science Park 1, D-66123 Saarbrücken, Germany
schmidt@absint.com

Simon Wegener

AbsInt Angewandte Informatik GmbH
Science Park 1, D-66123 Saarbrücken, Germany
wegener@absint.com

Abstract

We present `__builtin_ais_annot()`, a user-friendly, versatile way to transfer annotations (also known as flow facts) written on the source code level to the machine code level. To do so, we couple two tools often used during the development of safety-critical hard real-time systems, the formally verified C compiler CompCert and the static WCET analyzer aiT. CompCert stores the AIS annotations given via `__builtin_ais_annot()` in a special section of the ELF binary, which can later be extracted automatically by aiT.

2012 ACM Subject Classification Computer systems organization → Real-time systems, Hardware → Static timing analysis, Software and its engineering → Embedded software, Software and its engineering → Software verification, Software and its engineering → Automated static analysis, Software and its engineering → Compilers

Keywords and phrases Worst-Case Execution Time (WCET) Analysis, Annotation Support, CompCert, Tool Coupling, aiT

Digital Object Identifier 10.4230/OASIScs.WCET.2018.8

Acknowledgements The work presented in this paper has been conducted within the European ITEA3 project ASSUME (project number 14014), supported by the German Federal Ministry for Education and Research with the funding ID 01IS15031B and the French Ministry for the Economy and Finance. The responsibility for the content remains with the authors.



© Bernhard Schommer, Christoph Cullmann, Gernot Gebhard, Xavier Leroy, Michael Schmidt, and Simon Wegener;
licensed under Creative Commons License CC-BY

18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018).

Editor: Florian Brandner; Article No. 8; pp. 8:1–8:13

OpenAccess Series in Informatics



OASIS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

For safety-critical hard real-time systems, it is not only crucial that the software computes the correct result, but also that this happens in a timely manner. Examples for such software are the flight control systems of modern aircraft or the crankshaft-synchronized tasks of automotive motor control systems. One must therefore determine the worst-case execution time of critical parts of the software to get an embedded system certified.

The execution time of a program depends on three things: The input data of the program, the state of the CPU core on which the program is executed, and interference from the outside world (e.g. due to preemption or shared resources). It is thus not sufficient to just measure the execution time of a task once from start to end, as this measurement might underestimate the true WCET. Since the determination of the exact WCET is undecidable in general, WCET analysis tools instead compute estimates or safe upper bounds. There exist basically two mechanisms to do this [15, 14]:

- First, there are static analysis techniques that compute safe upper bounds with the help of a mathematical model of the execution platform. The precision of the results mostly depends on the predictability of the used hard- and software architecture [7, 5], but also on the availability and quality of the documentation concerning the execution time. If a feature is not described at all, or not well enough, the analysis model must incorporate the worst possible outcome to ensure soundness. AbsInt's aiT [2] is such a static WCET analysis tool.
- Second, there are measurement-based/hybrid techniques that obtain execution times from measurements on the real hardware. However, their soundness depends on whether it is possible to measure all possible executions of a program, which is usually infeasible due to the huge state space. AbsInt's TimeWeaver [4] is such a hybrid WCET analysis tool.

There exists a quasi-standard architecture for static WCET analysis tools. First, a binary reader disassembles a fully linked binary input executable into its individual instructions. Architecture specific patterns decide whether an instruction is a call, branch, return or just an ordinary instruction. This knowledge is used to form the basic blocks of the control flow graph (CFG). Then, the control flow between the basic blocks is reconstructed. In most cases, this is done completely automatically. However, if a target of a call or branch cannot be statically resolved, then the user needs to write some annotations to guide the control flow reconstruction. On this control flow graph, several static analyses take place to determine the values of registers and memory cells, addresses of memory accesses, bounds of loops and recursions, as well as infeasible code. Sometimes, loop bounds cannot be computed statically. Then, the user can guide the analysis via annotations. With this information, a microarchitectural analysis is started. There, a mathematical model of the target processor is used to derive timing bounds for each instruction, incorporating the intrinsic behavior of the pipeline and the memory hierarchy, in particular the caches. This abstract model does not cover all the features of the concrete processor, but only those that are relevant for timing analysis. Subsequently, the CFG together with the basic block timing information are used to construct an integer linear program (ILP). Solving this ILP gives then a path with the longest execution time (and consequently, an estimate of the worst-case execution time). For a measurement-based/hybrid WCET analysis tool, the microarchitectural analysis is replaced by an analysis step which extracts the basic block timing information from measurements taken on the real hardware.

The timing analysis tools of AbsInt can be guided via AIS annotations [1]. These annotations need to be specified on the machine code level, referring to e.g. code addresses

or memory cells, since the analysis works on the binary level. Usually, they are provided as extra files. The programmer, however, is normally more apt on the source code level. Thus, every tool support to bridge the gap between the (high-level) source code and the (low-level) machine code is welcome. We want to aid the programmer by offering a way to express annotations on the source level which are then automatically transferred to the binary level.

2 Embedded Program Annotations for aiT

In order to provide an automatic mechanism to transfer annotations from the source code level to the machine code level, we extended CompCert's already existing annotation mechanism via `__builtin_annot()` to (a) generate AIS compatible syntax for program points, registers, and memory cells and (b) store the generated annotations in a special section of the ELF executable. Consequently, we named the mechanism `__builtin_ais_annot()`. This mechanism is available in CompCert public version 3.3 and commercial version 18.04, and is supported in aiT version 18.04.

CompCert collects all annotations contained in a compilation unit and stores them in encoded form in a special section of the object file (`__compcert_ais_annotations`, see the assembler listing in the example below). While creating the final executable, the linker collects all annotation sections from the object files, concatenates them, and stores the result in the executable. Since we only use standard constructs of the assembler and linker, no changes neither to the linker nor to the assembler are necessary. To ensure that the final executable contains the special annotation section, the linker must be instructed to keep the section `__compcert_ais_annotations`, e.g. with a linker command file. Like debug sections, the annotation section is marked non-allocated/non-executable so that it is not loaded in the running program. aiT can automatically extract the annotations from section `__compcert_ais_annotations` and utilize them in analyses. Note that the order in which annotations are exported into the final executable is explicitly undefined. It is therefore not possible to rely on a specific order in which aiT will see the annotations.

For CompCert, annotations via `__builtin_ais_annot()` look like a call to a variadic function similar to “printf”: The first argument contains the AIS annotation and is also a format string. It can contain format specifiers like `%here` or `%e`, where the latter is tagged with an index number and refers to a specific argument independent of the order. It is also possible to refer to an argument more than once. `%here` is replaced with the absolute address of the annotation location in the final executable. Expressions, i.e., `%e1`, `%e2`, ... are replaced with an AIS expression for the value of the first, second, ... additional argument.

Furthermore, if the argument of the `__builtin_ais_annot()` is a constant pointer, the generated annotation contains the corresponding symbol name. This reference, then, is resolved by the linker to the address of the symbol, which allows to specify ambiguous symbols, for example static variables or functions.

Semantics

CompCert treats `__builtin_ais_annot()` as a call to an external function. No actual code is generated for the call, but the parameters of the builtin will be evaluated, as it is mandatory in C semantics. The execution of a `__builtin_ais_annot()` statement is modeled as producing an observable event that includes the text of the annotation and the values of the arguments. CompCert's formal correctness proof guarantees that, for a C source program that is deterministic and free of undefined behaviors, the compiled code performs the same observable events and in the same order as the source code [10, section 2]. This

■ **Listing 1** Source code annotations as part of dead code.

```
// assume that count is always zero
for (int i = 0; i < count; i++) {
    __builtin_ais_annot("try loop %here bound: %e1;", count);
    ...
}
```

gives strong guarantees that annotations are preserved throughout CompCert’s compilation and optimization passes.

If all additional arguments are non-volatile C variables or compile-time constant expressions, it is guaranteed that no additional code will be generated for `__builtin_ais_annot()`. Moreover, the location displayed as a replacement for the `%e` sequence is guaranteed to be the location where the corresponding variable resides.

Note that using local variables in parameter expressions to `__builtin_ais_annot()` may extend the liveness of those variables and hence prevent some optimizations. Furthermore, since `__builtin_ais_annot()` is considered a call to an external function it also acts as a barrier for many optimizations. In the current implementation, `__builtin_ais_annot()` can only be used at places where C statements are valid, i.e. within a function definition.

CompCert has no knowledge about the AIS annotation language and checks neither the syntax nor the semantics of the annotations. aiT can extract either all annotations embedded in an executable or none. Analyses that cover only a portion of the binary code – e.g., when doing a separate analysis for each task of the executable – may therefore issue warnings for annotations of unreachable program points. The `try` keyword of AIS can be used to suppress such warnings.

Robustness

In the context of optimizations and conditional compilation, the builtin is a *robust* mechanism to attach annotations to specific code locations. It does not rely on the rather ambiguous line information of the DWARF debug information, for example in macro usages, but rather utilizes the label mechanism of the assembler and linker to generate annotations with actual code addresses.

With the builtin-mechanism it is possible for CompCert to e.g. remove unreachable code together with the contained annotations or do code transformations like reordering code blocks without breaking the annotations. Consider the C code snippet in Listing 1. If constant propagation can prove that `count` is always zero, CompCert can remove the whole loop since it will never be executed. In such a situation the annotation will also be removed. In contrast to this, a conventional source code annotation via special formatted C comments (e.g. `/* ai: ...*/`) would remain visible and probably cause problems since aiT collects such annotations by scanning the source code without knowledge of any compiler optimizations. The same is true, when source code uses the C preprocessor for conditional compilation: CompCert can remove unused annotations while conventional source code annotations will remain visible for aiT. Section 4 discusses interactions with compiler optimizations in more details.

■ **Listing 2** Small C code example.

```
double func(double x)
{
    double data[10] = { ... };

    // x is known to be always >= 0.0 and < 10.0
    int i = x;

    // Refine the value in the location holding variable i
    __builtin_ais_annot("try instruction %here { enter with: %e1 =
        ↪ 0..9; };", i);
    return data[i];
}
```

Validation

In order to ensure that the linking was performed correctly, there exist the linker validation tool Valex. It takes as input a dump of the intermediate representation of the abstract assembly syntax as well as the linked binary and validates that the functions contained in the assembly are preserved, the addresses of the symbols are consistent and the initialization data of global variables is correct. In order to validate that all annotations are correctly translated and contained in the linked binary, we extended Valex to also check whether the AIS annotations are contained in the special AIS section `__compcert_ais_annotations` and that the addresses of the symbols used in the annotations are consistent.

3 Examples/Use Cases

The following, we present the different parts of the mechanism on an example. Moreover, we show how to use the annotation mechanism with aiT in, e.g., a software library that will be integrated in an embedded system.

Refinement/Assertion of Values

The first example is borrowed from [6]. In this example, shown in Listing 2, a double value with a known range is converted to an integer and used as an index, e.g. to access a look-up table. aiT has currently no knowledge of floating point values and assumes the full range of possible values for them. Thus, it needs help to restrict the range of `i` (and derived from it, a memory access) to a small range.

CompCert will emit the PowerPC assembly code shown in Listing 3 when compiling the example code. The assembler code at the labels `.L116`, `.L117` and `.L119` corresponds to the C code shown in the assembler comments below the labels. The assembler code at label `.L120` removes the stack frame, whereas the assembler code at label `.L121` performs the actual return. The format specifier `%here` has been replaced by CompCert with the label `.L118` which will later be replaced by the assembler/linker with an address. The format specifier `%e1` has been replaced with the register that was allocated to variable `i`. Finally, aiT extracts the annotation from the ELF executable, as shown in Listing 4.

8:6 Embedded Program Annotations for WCET Analysis

■ **Listing 3** Assembly code generated by CompCert for the example in Listing 2.

```
.L116:
; int i = x;
  fctiwz  f13, f1
  stfdu   f13, -8(r1)
  lwz     r5, 4(r1)
  addi    r1, r1, 8

.L117:
; __builtin_ais_annot("try instruction %here { enter with: %e1 =
  ↪ 0..9; };", i);

.L118: .L119:
; return data[i];
  addi    r3, r1, 16
  rlwinm  r4, r5, 3, 0, 28 ; 0xffffffff8
  lfdx    f1, r3, r4

.L120:
  addi    r1, r1, 96

.L121:
  blr

...

.section  "__compcert_ais_annotations",,n
.ascii  "# file:test.c line:25 function:func\ntry instruction "
.byte  7,4
.4byte .L118
.ascii  "\{ enter with: reg(\"r5\") = 0..9; \};"
.ascii  "\n"
```

■ **Listing 4** Annotation extracted by aiT.

```
# file:test.c line:25 function:func
try instruction 0x10013c { enter with: reg("r5") = 0..9; };
```

Besides its use for the refinement of values, the annotation mechanism can also be used to insert “assert” annotations about known value ranges of variables or function parameters (see Listing 5). aiT will then report if any assertion is violated.

Loop and Recursion Bounds

Loop or recursion bounds cannot always be automatically derived by aiT’s value analysis. A (probably overestimated) annotation can be specified in the source code of a common library routine to ensure that aiT can compute reasonable results without annotation effort or to increase analysis precision at specific code locations. If necessary, users of aiT can improve this annotation by giving more specific annotations for the actual analysis context. Listing 6 shows a data-dependent loop where the bound depends on the input parameter of the surrounding function. This fact can easily be expressed with `__builtin_ais_annot()`.

■ **Listing 5** Assertion of input values of a function to be validated by aiT.

```
int func(int a, int b, int c)
{
    __builtin_ais_annot("try instruction %here {\n"
                       "  assert always enter with: %e1 in (0..7);\n"
                       "  assert always enter with: %e2 in (2..4);\n"
                       "  assert always enter with: %e3 in (1..9);\n"
                       "};", a, b, c);
    ...
}
```

■ **Listing 6** Specifying a bound for a data-dependent loop.

```
int strncmp_x(char s[], char t[], int len)
{
    int i;
    for (i = 0; i < len && ...; i++) {
        __builtin_ais_annot("try loop %here bound: %e1;", len);
        ...
    }
    return 0;
}
```

Sometimes the analysis precision can be greatly improved if the first x iterations of a loop can be distinguished. aiT supports this via virtual unrolling¹. Note that this does not change the binary, nor does it affect the compilation process in any way. Instead, aiT uses analysis contexts to distinguish the first n loop iterations from all following ones. An annotation that enables the virtual unrolling of the first 49 iterations of a loop is shown in Listing 7.

In case of busy-waiting loops, no loop bound can be derived statically. However, it is also not easy to derive the maximal number of iterations manually, because this number depends on the execution time of the loop body. aiT supports a way to specify the loop bounds of busy-waiting loops depending on their worst-case waiting times, see Listing 8.

Finally, in automotive software, it is often the case that some implicit recursion happens during error handling. For these cases, we need to specify recursion bounds as shown in Listing 9.

Memory Areas

The properties and contents of memory areas can also be specified for aiT. For example, special care needs to be taken when accessing memory-mapped sensors and other devices which provide data via asynchronously updated buffers. We can specify that these buffers are read-only and volatile, see Listing 10.

¹ For historical reasons, aiT uses the name *virtual unrolling*, but *virtual loop peeling* might be a better name.

■ **Listing 7** Providing unrolling hints for loops for improved precision in aiT.

```
#define MAX_STR_LEN 50

void strcpy_x(char s[], char t[])
{
    int i = 0;
    while (( s[i] = t[i] ) != '\0') {
        __builtin_ais_annot("try loop %here mapping { default unroll:
            ↪ %e1; }", MAX_STR_LEN);
        ...
    }
}
```

■ **Listing 8** Specifying a time bound for a busy waiting loop.

```
void openCanSocket(volatile struct device_t* device)
{
    ...
    // Busy wait for ACK. Assume a worst-case timing of 23 us
    while(device->bus_data != 0x00) {
        __builtin_ais_annot("try loop %here takes: 23 us;");
    }
    ...
}
```

Another common pattern – shown in Listing 11 – is to copy calibration data from ROM to RAM once when the system boots. Without further annotations aiT usually cannot know which data is stored in the calibration vector. With the `copy area` annotation, the precision of the analysis can be improved.

4 Interactions with Compiler Optimizations

Compiler optimizations can complicate the transmission of source-level annotations to the compiled code: if done carelessly, optimizations can remove annotations, or render them inapplicable to the code after optimization.

Preservation guarantees for annotations

As mentioned in section 2, CompCert’s proof of semantic preservation guarantees that annotations are not erased during compilation, unless they occurred in parts of the code that are unreachable during execution, and that they not reordered or moved in the generated code, relative to each other and relative to other observable actions (such as external function calls and accesses to volatile variables). The proof does not rule out the possibility that optimizations would move annotations around pure, non-observable computations. However, this is not the case for CompCert’s optimizations, which are conservative and make no attempts to optimize around calls to unknown functions, which include annotation statements. Another reason why CompCert preserves the position of annotations relative to the code is that it currently performs no loop optimizations, as discussed below.

■ **Listing 9** Specifying a recursion bound and an incarnation bound for a recursive routine.

```
void errorHook(unsigned char err_code)
{
    __builtin_ais_annot("try routine %e1 {\n"
                       "    recursion bound: 1;\n"
                       "    incarnation limit: 1;\n"
                       "}", &errorHook);
    ...
}
```

■ **Listing 10** Memory areas that are used for external devices can be marked accordingly.

```
volatile char* device_buffer[128];

void init_device()
{
    __builtin_ais_annot("area %e1 width %e2 {\n"
                       "    readable: true;\n"
                       "    writable: false;\n"
                       "    volatile;\n"
                       "};", &device_buffer, sizeof(device_buffer));
    ...
}
```

Invariance of annotation texts

CompCert is agnostic with respect to the annotation language: it gives no specific meaning to the annotation strings and never modifies them during optimization. Consequently, an annotation that mentions functions or variables by their names can become pointless as a result of optimization.

Consider the example shown in Listing 12. After inlining, the annotation occurs within function `g` but still refers to an instruction in function `f`. The hardcoded reference to `f` in the annotation text must be replaced by a relative code position `%here`.

Similarly, if a static variable is mentioned by name in an annotation but unused anywhere else, CompCert will remove this variable and make the annotation meaningless. To avoid this risk, the variable should appear as an explicit argument of the annotation (see Listing 13).

Moreover, dead code elimination may remove annotations that have a global effect (see Listing 14).

Finally, some AIS annotations about function calls can become inapplicable if the call is turned into a jump by CompCert's tail call optimization. The workaround here is to turn tail call optimization off.

Towards loop optimizations

It is well known that program annotations that bound the number of iterations of a loop are difficult to maintain in the presence of loop optimizations [12]. CompCert does not address this problem since currently it does not perform any optimizations over loops. If classic loop optimizations were added in the future, they would combine poorly with loop count

8:10 Embedded Program Annotations for WCET Analysis

■ **Listing 11** A source level annotation to specify which data is copied from ROM into RAM.

```
volatile char calibration_data[__CALIBRATION_ROM_SIZE];

// setup at boot time
void init_calibration_data()
{
    __builtin_ais_annot("copy area %e1 width %e2 from %e3;}",
                       &calibration_data,
                       __CALIBRATION_ROM_SIZE,
                       (void *)__CALIBRATION_ROM_START);

    memcpy((void *)calibration_data, (void *)__CALIBRATION_ROM_START,
           ↪ __CALIBRATION_ROM_SIZE);
}
```

■ **Listing 12** A source level annotation that is not robust regarding inlining.

```
static inline void f(void)
{
    __builtin_ais_annot("try routine 'f' ...");
    ...
}

int g(int x)
{
    ...
    f();
    ...
}
```

annotations expressed with `__builtin_ais_annot()`. First, most optimizations over loop nests, such as loop interchange or loop blocking, change the order in which iterations are performed. Hence, they do not apply if the loop body can perform observable operations such as I/O. CompCert's annotations being observable operations, the presence of one or several `__builtin_ais_annot()` to give loop bounds would inhibit these optimizations.

Second, optimizations such as loop unrolling make upper bounds on the number of loop iterations inaccurate (unrolling by a factor of k divides the number of iterations by k). Yet, in the CompCert approach, such an optimization is not allowed to rewrite the annotation to adjust the loop count, because this would change the observable behavior of the annotation according to the formal semantics. This is a real conundrum with no easy workaround.

5 Related Work

CompCert [11, 3] already supports an annotation mechanism via `__builtin_annot()` [6]. There, the annotation string is printed as a comment in the generated assembly code. An additional tool can be used to parse these comments and to generate annotations, e.g. for aiT. Our work makes this extra annotation generator superfluous, as we print annotations that are (a) already in the right syntax to be understandable by aiT and (b) are directly stored in the executable binary.

■ **Listing 13** Explicit references to variables increase robustness.

```
__builtin_ais_annot("... static_var ...");           // risky
__builtin_ais_annot("... %e1 ...", &static_var);     // safe
```

■ **Listing 14** Dead code elimination may remove annotations.

```
int x = 5;
if (x == 7) {
    __builtin_ais_annot("# some global AIS annotation"); // removed
}
```

The ENTRA (Whole-Systems ENergy TRAnsparency) Deliverable D2.1 “Common Assertion Language” [8] describe a similar mechanism to transfer properties from the source to the machine level. Pragmas are used to specify properties which are translated to comments written as inline assembler statements. These comments need to be extracted from the assembler files, as they are not stored in the final binary.

WCC [9] also uses pragmas to specify properties on the source code level. During compilation, which also contains steps to optimize the worst-case timing behavior, the compiler framework translates these properties into annotations for aiT in order to steer the WCET estimation of intermediate binaries. WCC only covers a subset of the annotations possible with AIS – loop bounds and linear flow constraints – whereas our approach allows to exploit the full power of the AIS annotation language. On the other hand, WCC is able to transform the annotations when applying optimizations like loop unrolling.

Li, Puaut and Rohou [12] present a framework in which annotations on the source code are transformed into annotations on the binary level in the presence of compiler optimizations. They focus on loop count annotations and their preservation through classic loop optimizations. Our approach cannot deal with loop optimizations yet, but supports a more general annotation language and provides formal correctness guarantees.

aiT allows to extract AIS annotations from source code via special markers in C comments [1], for example: `/* ai: loop here bound: 10; */`. However, the special program point `here` might not be resolvable due to compiler optimizations. Moreover, whenever source code annotations are extracted from a source file, the whole file is scanned for AIS comments independent from any `#if`, `#ifdef`, or `#ifndef` preprocessor directives.

TuBound [13] uses pragmas to annotate additional knowledge for the timing analysis. In contrast to aiT, which operates on fully linked binaries, TuBound incorporates a compiler and takes C code as input.

6 Conclusions and Future Work

CompCert’s annotation mechanism via `__builtin_ais_annot()` enables programmers to *reliably* annotate flow facts on C source code level and reason about C variables instead of using code addresses or processor registers. Its versatility allows to exploit the full power of the AIS annotation language used by aiT. These annotations are automatically carried through the compilation chain and the linked executable into aiT without using external annotation files. Thus, version mismatch between executable and annotations is successfully prevented, which is especially useful for binary code libraries. Program points and other addresses survive recompilation, thus easing the maintenance effort needed for annotations.

There are two shortcomings of the current implementation of `__builtin_ais_annot()`: First, due to its treatment as a call to an external function, it cannot be placed at the top-level of a compilation unit, unlike, for example, a variable declaration. Second, since all annotations are merged in a single section, they cannot be extracted individually. We wish to address these shortcomings in future work.

References

- 1 AbsInt Angewandte Informatik GmbH. *a³* for PPC User Documentation (Version 18.04).
- 2 AbsInt Angewandte Informatik GmbH. aiT Worst-Case Execution Time Analyzer. URL: <http://www.absint.com/ait/>.
- 3 AbsInt Angewandte Informatik GmbH. CompCert: formally verified optimizing C compiler. URL: <http://www.absint.com/compcert/>.
- 4 AbsInt Angewandte Informatik GmbH. TimeWeaver: Hybrid Worst-Case Timing Analysis. URL: <http://www.absint.com/timeweaver/>.
- 5 Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard von Hanxleden, Reinhard Wilhelm, and Wang Yi. Building timing predictable embedded systems. *ACM Trans. Embedded Comput. Syst.*, 13(4):82:1–82:37, 2014. doi:10.1145/2560033.
- 6 Ricardo Bedin França, Sandrine Blazy, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Formally verified optimizing compilation in ACG-based flight control software. In *ERTS 2012: Embedded Real Time Software and Systems*, 2012.
- 7 Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza (Burguière), Jan Reineke, Benoît Triquet, Simon Wegener, and Reinhard Wilhelm. Predictability Considerations in the Design of Multi-Core Embedded Systems. *Ingenieurs de l'Automobile*, 807:26–42, 2010.
- 8 ENTRA Consortium. Deliverable D2.1 “Common Assertion Language”. URL: <http://entraproject.ruc.dk/deliverables>.
- 9 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.
- 10 Xavier Leroy. A formally verified compiler back-end. *J. Automated Reasoning*, 43(4):363–446, 2009.
- 11 Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert – a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems*, 2016.
- 12 Hanbing Li, Isabelle Puaut, and Erven Rohou. Traceability of flow information: Reconciling compiler optimizations and WCET estimation. In Mathieu Jan, Belgacem Ben Hedia, Joël Goossens, and Claire Maiza, editors, *22nd International Conference on Real-Time Networks and Systems, RTNS '14, Versailles, France, October 8-10, 2014*, page 97. ACM, 2014. doi:10.1145/2659787.2659805.
- 13 Adrian Prantl, Markus Schordan, and Jens Knoop. Tubound - A conceptually new tool for worst-case execution time analysis. In Raimund Kirner, editor, *8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Prague, Czech Republic, July 1, 2008*, volume 8 of *OASICS*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008. doi:10.4230/OASICS.WCET.2008.1661.

- 14 Simon Wegener. Towards Multicore WCET Analysis. In Jan Reineke, editor, *17th International Workshop on Worst-Case Execution Time Analysis, WCET 2017, June 27, 2017, Dubrovnik, Croatia*, volume 57 of *OASICS*, pages 7:1–7:12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/OASICS.WCET.2017.7.
- 15 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3):36:1–36:53, 2008. doi:10.1145/1347375.1347389.

Fine-Grain Iterative Compilation for WCET Estimation

Isabelle Puaut

Univ Rennes, Inria, CNRS, IRISA
Isabelle.Puaut@irisa.fr

Mickaël Dardaillon

Univ Rennes, Inria, CNRS, IRISA
Mickael.Dardaillon@irisa.fr

Christoph Cullmann

AbsInt Angewandte Informatik GmbH
Science Park 1, D-66123 Saarbrücken, Germany
Cullmann@absint.com

Gernot Gebhard

AbsInt Angewandte Informatik GmbH
Science Park 1, D-66123 Saarbrücken, Germany
Gebhard@absint.com

Steven Derrien

Univ Rennes, Inria, CNRS, IRISA
Steven.Derrien@irisa.fr

Abstract

Compiler optimizations, although reducing the execution times of programs, raise issues in static WCET estimation techniques and tools. Flow facts, such as loop bounds, may not be automatically found by static WCET analysis tools after aggressive code optimizations. In this paper, we explore the use of iterative compilation (WCET-directed program optimization to explore the optimization space), with the objective to (i) allow flow facts to be automatically found and (ii) select optimizations that result in the lowest WCET estimates. We also explore to which extent code outlining helps, by allowing the selection of different optimization options for different code snippets of the application.

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

Keywords and phrases Worst-Case Execution Time Estimation, Compiler optimizations, Iterative Compilation, Flow fact extraction, Outlining

Digital Object Identifier 10.4230/OASICS.WCET.2018.9

Acknowledgements This work was funded by European Union's Horizon 2020 research and innovation program under grant agreement No 688131, project Argo. The authors would like to warmly thank Benjamin Rouxel, Stefanos Skalistis and Imen Fassi and the anonymous reviewers, for their helpful comments on earlier drafts of this paper.



© I. Puaut, M. Dardaillon, C. Cullmann, G. Gebhard, and S. Derrien;
licensed under Creative Commons License CC-BY

18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018).

Editor: Florian Brandner; Article No. 9; pp. 9:1–9:12

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Real-time systems play an important role in our daily life. In hard real-time systems, computing correct results is not the only requirement. Results must be also produced within pre-determined timing constraints, typically deadlines. To obtain strong guarantees on the system temporal behavior, designers must compute upper bounds of the Worst-Case Execution Times (WCET) of the tasks composing the system, in order to finally guarantee that they meet their deadlines. Standard static WCET estimation techniques [18] compute such bounds from static analysis of the machine code. Their goal is to obtain a *safe* and *accurate* estimation of a task execution time on a given hardware platform. The *safety* criterion ensures that the WCET holds for any possible execution of the task on the target platform, whereas *accuracy* avoids resource over-provisioning.

WCET analysis is confronted with the challenges of extracting knowledge of the execution flow of an application from its machine code. In particular, loop bounds are mandatory to estimate WCETs. Extraction of flow information can be performed automatically by static WCET analysis tools, or guided by the designer through *flow facts* (loop bounds, unfeasible paths) expressed using source-level annotations.

Compiler optimizations are well known to significantly improve the (average-case) performance of programs, but raise issues regarding WCET estimation. On the one hand, automatic detection of loop bounds may not be feasible anymore because the generated code is more complex and less amenable to static analysis. On the other hand, manual annotations may not be valid anymore after the code optimizations (loops may have been unrolled, re-rolled, split, fused, or may simply have disappeared from the code).

To safely benefit from compiler optimizations, in this paper, we explore the use of *iterative compilation* (exploration of the optimization space) to minimize WCETs instead of average-case in the original use of iterative compilation [6, 2, 4]. More precisely, our contributions are the following:

- We propose and evaluate coarse-grain (application-level) WCET-oriented optimization exploration strategies. Each of the two proposed strategies selects a sequence of optimization passes that (i) allows static WCET analysis tools to automatically detect loop bounds (i.e. disregards optimizations making the WCET estimation fail because it is unable to detect some loop bounds without the use of annotations); (ii) results in the lowest estimated WCET. Two strategies are proposed, the former based on random selection of optimization sequences and the latter using a genetic algorithm.
- We detail and provide preliminary experimental data on a fine-grain (code snippet-level) WCET-oriented optimization exploration strategy, that allows different optimization per code snippet. Interesting code snippets (for the scope of this paper, loops) are outlined, to allow different optimization sequences within a same function. This enables selective application of optimizations: code snippets for which static WCET estimation tools can detect loop bounds with optimizations can be aggressively optimized, whereas the remaining parts can be left un-optimized and later fed with source-level flow fact annotations [13].

Experiments were conducted using the LLVM [12] compilation framework, that allows fine control over optimization passes, and aiT [1], the industry standard for static WCET analysis. The target architecture is the Leon3 core, used to build a predictable multi-core architecture in the framework of the Argo H2020 project¹.

¹ The work presented in this paper is part of ARGO (<http://www.argo-project.eu/>), funded by the European Commission under Horizon 2020 Research and Innovation Action, Grant Agreement Number 688131.

The rest of this paper is organized as follows. We first introduce in Section 2 the background on compilation optimization and static WCET estimation. Section 3 presents the coarse-grain and fine-grain strategies to explore the optimization space in order to both enable automatic flow fact derivation and minimize WCET estimates. Section 4 then describes the experimental setup used to evaluate their quality. Section 5 is devoted to an extensive experimental evaluation of the impact of optimizations on WCET estimates, with and without the proposed techniques. We conclude in Section 6.

2 Background & Related Work

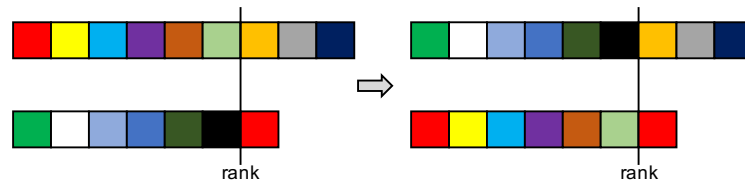
Turning on compiler optimizations impacts static WCET estimation from two respects. First, automatic detection of flow information, in particular loop bounds that are mandatory for WCET estimation, may not be feasible anymore because the transformed code is more complex and less amenable to static analysis. Second, manual source-level annotations may not be valid anymore after the code transformations, in particular the ones deeply modifying loops (loop unrolling, re-rolling, fusion, splitting, among others).

To address these issues, one solution would be to use some feedback provided by the compiler [3] to identify which optimizations were applied, in which ordering and with which parameter, and to transform manually-provided source-level flow information accordingly. However, the level of feedback provided by state-of-the-art compilers is still very limited. Another approach is to instrument the compiler such that it transforms source-level flow information jointly with code transformations, as done by several authors within *gcc* [11], *LLVM* [14, 15] or *WCC* [17]. This however imposes to stick to a given compiler version, or to maintain the flow-fact co-transformation framework along compiler versions. This is the approach followed by the WCC compiler infrastructure [9], containing WCET-oriented optimization and flow fact traceability features. Although a compiler designed specifically for WCET has many benefits, it lacks many optimizations available in standard compilation toolchains such as *gcc/LLVM*.

In this paper, we experiment a completely different approach based on the principles of *iterative compilation*, which is a now mature technology in compilers for optimizing (average-case) performance [6, 2]. The benefits of our approach are twofold. First, we rely on standard industrial strength compiler toolchain to benefit from their large number of available optimizations. Second, we consider the compiler as a black box and adapt the optimization sequences to the code under study to minimize the WCET. Our metrics for evaluating the quality of optimization sequences differ from standard iterative compilation: WCET is optimized instead of average-case performance, and optimizations sequences may be regarded as invalid if static WCET estimation fails at determining loop bounds. One of the hardest challenge in iterative compilation is dealing with the sensitivity of execution performance to input data. Interestingly, this issue does not manifest in our case, because WCET estimation is by definition insensitive to input data, making our approach even more relevant.

3 Proposed WCET-directed Optimization Strategies

Our approaches combine two techniques, with the variation of the optimized code granularity presented in Section 3.1, and the iterative strategies to explore the resulting optimization space in Section 3.2.



■ **Figure 1** Crossover operation in genetic exploration.

3.1 Optimization Granularity

The intuitive way to optimize an application is to compile all its functions with the same optimization options. We refer to this approach as *coarse-grain optimization* in the rest of the paper.

However, a single problem in a part of the application code, due to the application of an optimization, can forbid the use of that optimization on the whole application. To circumvent this difficulty, we propose to use different optimizations sequences on different parts of the code. In order to isolate a block of code to apply different optimizations we use *outlining* [16]. With this technique, the code snippet to isolate is replaced by a call to a new function that implement the same functionality. All variables used by the code snippet are passed as parameters to the generated function. The naive way to pass arguments is to pass everything by reference, which may significantly increase the average and the worst case execution time. Using liveness [5] properties of the used variables, we can filter which variable needs to be passed by reference or value. In our implementation all arrays are passed by reference; scalars are passed by reference if they are live-out, and by value otherwise; pointers which may be modified are similarly passed by reference.

In this work we systematically outline all outer loops, using to the GeCoS source-to-source code transformation framework [10]. Different optimizations sequences are generated, for the original functions, and also for each new function generated by loop outlining. We refer to this approach as *fine-grain optimization* in the rest of the paper.

3.2 Iterative Optimization Space Exploration Strategies

We designed two strategies to explore the optimization space:

- **Random exploration.** For this strategy, for each experiment the number of optimization passes to be applied is selected randomly. The sequence of passes to be applied is then constructed, each optimization in the sequence being selected randomly, with no attention paid to duplicated optimization passes. This random selection of optimizations is repeated a fixed number of times.
- **Genetic exploration.** A population is set-up, each individual representing a sequence of optimization passes to be applied. Individuals in the population are selected for breeding. At every generation, there is a probability of *mutation* of individuals (here change of one pass in the optimization sequence, selected randomly). Then, the population is doubled in size by randomly selecting N pairs of individuals for breeding. Each pair gives birth to two children by *crossover*. Figure 1 gives an example of *crossover*. A rank in the optimization sequence is selected randomly and the sequences of optimizations are swapped. Similarly to *random exploration*, to keep the implementation simple, no attention is paid to duplicated passes in an optimizations sequence. The optimizations sequences for the initial individuals are selected randomly (using the same techniques

as in the *random exploration* strategy). At each generation, the N best individuals (optimizations sequences for which aiT succeeds in estimating loop bounds, keeping the N lower WCET values) are kept.

4 Experimental Setup

This section presents our experimental setup. We first briefly discuss our choice of input benchmarks. We then describe the compiler and WCET tools used in our experiments, before detailing the parameters of our optimization space search strategies.

4.1 Corpus of Codes

Experiments were conducted on two image processing data benchmarks (Harris and PIPS, see description in Table 1) from the Mälardalen WCET benchmark suite² and from the PolyBench/C benchmark suite³. We restricted our study to the benchmarks analyzable by aiT with no additional information when compiled without optimization (-O0). This excludes the benchmarks that call library functions (*libmath*, *libc*) that need manual flow annotations. The complete list of benchmarks is given in Table 1 with a small description of each benchmark.

4.2 Compiler and WCET Estimation Tools

Programs are compiled using LLVM [12], version 4.0.0, targeting the Leon3 architecture (Sparc instruction set). Programs are first compiled into LLVM bitcode using the *clang* front-end, before using the *opt* LLVM optimizer to selectively apply optimization passes and then generating a Leon3 executable. *opt* takes as parameters an ordered list of optimization passes. *opt* automatically applies any analysis passes required when turning on a given optimization. The order of application of optimizations is respected unless there are dependencies between passes, in which case *opt* reorders the passes to respect the dependencies. At this stage of our work, we assume the combination of optimization passes to be correct. This will need to be verified for certification concerns, and is considered outside the scope of the paper.

Programs WCETs are estimated using aiT, the industry standard for static WCET analysis, version 17.04, for the Leon3 target [1], configured with no cache. No flow annotations are given to aiT, resulting in situations where the tool is not able to derive them automatically on the optimized code. The virtual unrolling factor of aiT used by its value analysis is set to 2.

Detection of loop bounds in aiT uses an interprocedural data-flow based analysis operating at assembly level. The analysis first searches for loop counters (registers or memory cells with known value when entering the loop). Potential loop counters are further examined by a data-flow analysis to derive *loop invariants* (expressions indicating how the loop counter is modified at each iteration). More details can be found in [7].

4.3 Parameters of Optimization Exploration Strategies

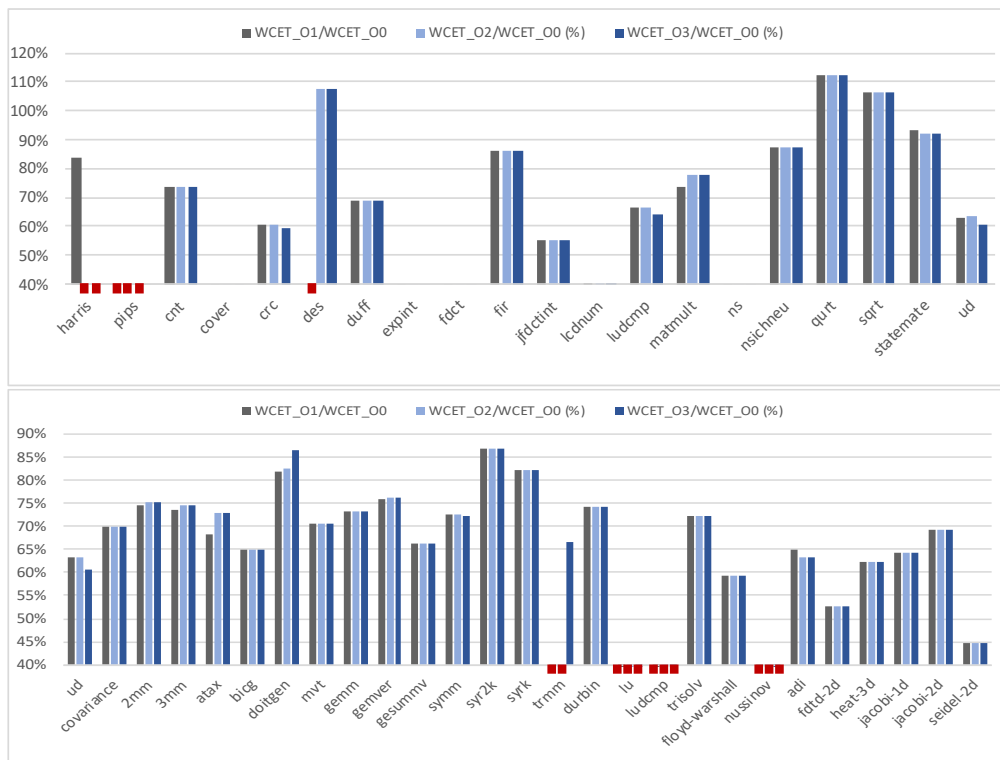
To provide a fair comparison, the same number of optimization sequences were experimented on each benchmark. For *random exploration*, 1000 random optimizations sequences were generated. By default, our *genetic exploration* generated optimizations sequences from 10

² <http://www.mrtc.mdh.se/projects/wcet/benchmarks.html>

³ <http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/>

■ **Table 1** Corpus of programs.

Harris	Classical Harris corner detection algorithm.
PIPS	Industrial use case from the ARGO project [8]. Image processing pipeline for post-processing raw data from a polarized image sensor.
cnt	Counts non-negative numbers in a matrix
cover	Program for testing many paths.
crc	Cyclic redundancy check computation on 40 bytes of data.
des	DES and Triple-DES encryption/decryption algorithm.
duff	Using “Duff’s device” from the Jargon file to copy 43 bytes array.
expint	Series expansion for computing an exponential integral function.
fdct	Fast Discrete Cosine Transform.
fir	Finite impulse response filter (signal processing algorithm) over a 700 items long sample.
jfdctint	Discrete-cosine transformation on a 8x8 pixel block.
lcdnum	Read ten values, output half to LCD.
ludcmp	LU decomposition algorithm.
matmult	Matrix multiplication of two 20x20 matrices.
ns	Search in a multi-dimensional array.
nsichneu	Simulate an extended Petri Net.
qurt	Root computation of quadratic equations.
sqrt	Square root function implemented by Taylor series.
statemate	Automatically generated code.
ud	Calculation of matrices.
covariance	Co-variance computation.
2mm	2 Matrix multiplications.
3mm	3 Matrix multiplications.
atax	Matrix transpose and vector multiplication.
bicg	BiCG sub kernel of BiCGStab linear solver.
doitgen	Multi-resolution analysis kernel (MADNESS).
mvt	Matrix vector product and transpose.
gemm	Matrix multiply.
gemver	Vector multiplication and matrix addition.
gesummv	Scalar, vector and matrix multiplication.
symm	Symmetric matrix-multiply.
syr2k	Symmetric rank-2k operations.
syrk	Symmetric rank-k operations.
trmm	Triangular matrix-multiply.
durbin	Algorithm for solving Yule-Walker equations.
lu	LU decomposition without pivoting.
ludcmp	Solving a system of linear equations using LU decomposition followed by forward and backward substitutions.
trisolv	Triangular solver.
floyd-warshall	Finds the shortest path in a graph.
nussinov	Algorithm for predicting RNA folding using dynamic programming.
adi	Alternating Direction Implicit solver.
fdtd-2d	2-D finite different time domain kernel.
heat-3d	Solving of heat equation over 3D space.
jacobi-1D	1-D Jacobi stencil computation.
jacobi-2D	2-D Jacobi stencil computation.
seidel-2D	2-D Seidel stencil computation.



■ **Figure 2** Ability to derive WCETs at -O1, -O2 and -O3.

generations of 100 individuals each (same number of distinct optimization sequences as *random exploration*). 15% of individuals are mutated at each generation.

In the *coarse-grain* case, all files are compiled using the same optimizations sequence, while *fine-grain* optimization uses different sequences for each file. Regarding the extension of the *genetic exploration* strategy to *fine-grain*, the implemented *mutation* operator mutates all files (with a different mutation per file). Similarly, we chose to apply the *crossover* operation to all files when breeding two individuals.

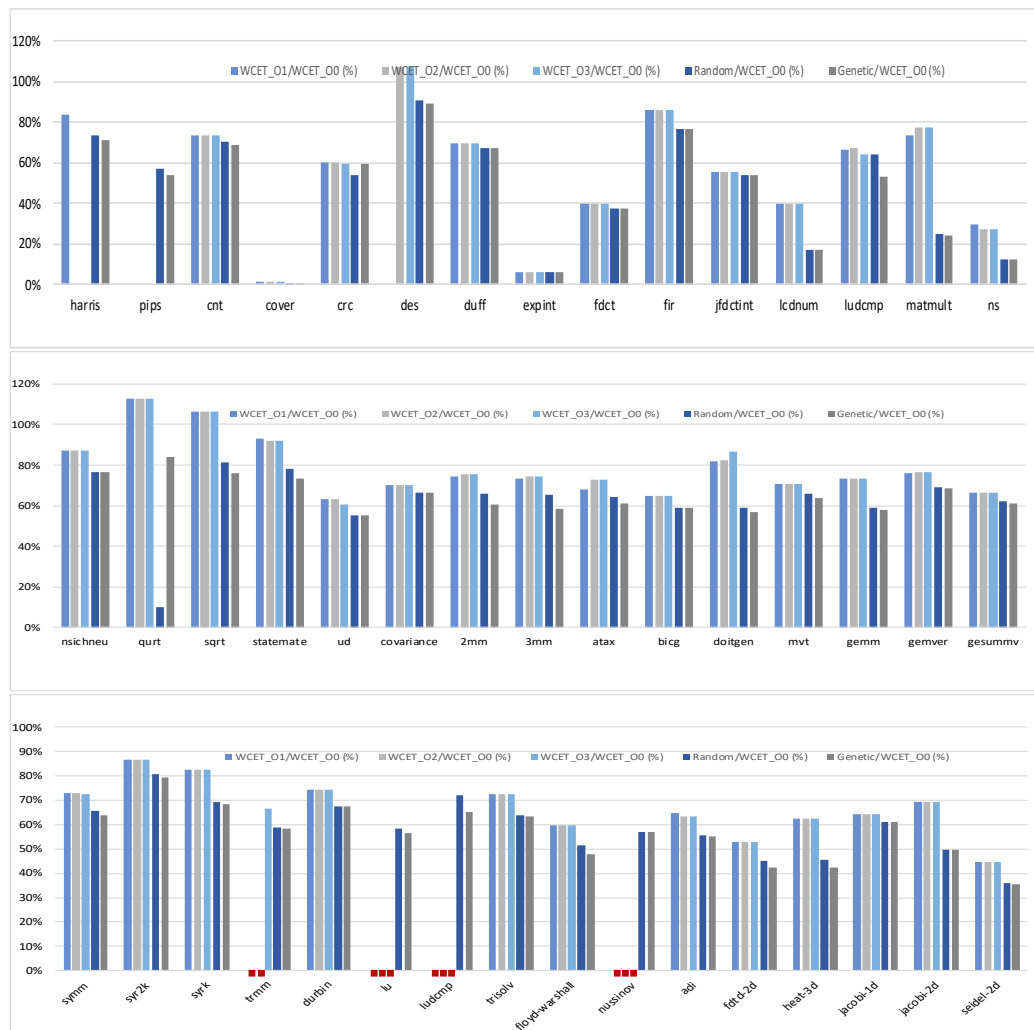
5 Experimental Evaluation

In this section, we look at the impact of standard optimizations levels on the WCET estimates, and then evaluate our *coarse-grain* and *fine-grain* optimization selection strategies.

5.1 Impact of Optimizations on the Ability to Derive Flow Information

LLVM comes with four optimization levels -O0 (no optimization applied) to -O3 (highly optimized code). Figure 2 gives for all optimizations levels beyond -O0 (-O1, -O2, -O3) the ratio $WCET_{-O_i} / WCET_{-O_0}$ expressed in percentage (the lower the better). No bar for a given optimization level means that aiT was not able to detect loop bounds automatically.

The results show that in most situations, turning on optimizations results in lower WCET estimates than when compiling with option -O0 (ratio $WCET_{-O_i} / WCET_{-O_0}$ lower than 100 %). However, in some cases (benchmarks *quart* and *sqrt* and *des*) optimized codes result in larger WCETs than non-optimized ones. For some benchmarks (*harris*, *pips*, *trmm*, *lu*, *ludcmp*, *nussinov*), aiT was not able to extract loop bounds when optimizations are turned on

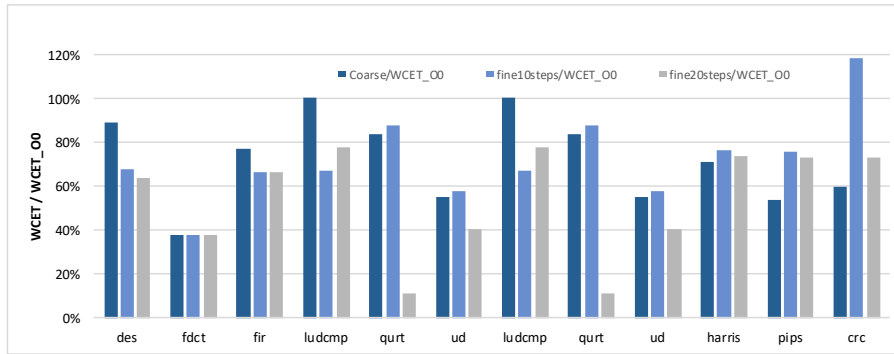


■ **Figure 4** Coarse grain exploration of optimization space.

generated. Understanding why the optimization passes other than *-jmp-threading* generate hard-to-analyze code is left for future work.

5.2 Evaluation of Coarse-Grain Optimization Selection Strategies

Figure 4 presents the experimental results for all benchmarks. The first important remark on the experimental results is that for all benchmarks aiT was able to derive loop bounds automatically, even in the situations where some optimization levels made it impossible before (benchmarks *harris*, *pips*, *trmm*, *lu*, *ludcmp*, *nussinov*). *On all benchmarks, exploring the optimization space using random exploration resulted in WCET estimates lower than the best WCET possible with -O1, -O2 and -O3. The gain is most of the time significant (21% on average as compared with the best optimization level). Finally, except for benchmarks *crc* and *qurt*, genetic exploration outperforms random exploration. Preliminary experiments with genetic exploration made us select large populations and low number of generations, that turned out to give better WCET estimates than lower population sizes and larger number of generations.*



■ **Figure 5** Fine grain exploration of optimization space.

5.3 Evaluation of Fine-Grain Optimization Selection Strategies

Due to time constraints, experiments of the proposed fine-grain optimization strategies were conducted on the image processing benchmarks *harris*, *pips* and the Mälardalen benchmarks only. We further restricted the benchmarks to the ones containing loops, and avoided those containing a single loop that covers the entire code. Results are given in Figure 5, showing three values for each benchmark: $WCET_{coarse}/WCET_{O0}$, $WCET_{fine10steps}/WCET_{O0}$ (10 generations) and $WCET_{fine20steps}/WCET_{O0}$ (20 generations instead of the default value of 10).

The first results obtained are encouraging (improvement of 37 % of the WCET estimates on average). On 6 out of the 9 benchmarks analyzed (all but the 3 ones at the right of the figure), the fine-grain *genetic exploration* outperforms the coarse-grain exploration. Moreover, for all benchmarks except one, having 20 generations instead of 10 significantly improves WCETs, at the cost of an analysis time twice longer. This result is expected, since the optimization space to be explored is much larger than for the coarse-grain strategy. We believe there is room left for improvements, by tuning the parameters of the genetic algorithm to better deal with the very large optimization space to be explored, or avoid the cost of outlining when not beneficial to the WCET.

6 Conclusion

Compiler optimizations are known to add challenges when estimating the WCET of applications. Hence it is quite common to disable them when dealing with critical systems. In this paper, we proposed an iterative compilation workflow to reconcile timing critical applications with compiler optimizations. Our methods, based on optimization space exploration, show a significant tightening of the estimated WCETs. Our first exploration of fine-grain application of optimizations demonstrated opportunities to further reduce WCET estimates (improvement of 37 % of the WCET estimates on average). Future work is still needed to take full benefit of fine-grain exploration of optimizations. A first direction is to better explore the very large optimization space, for example by concentrating the optimization effort of regions having the most impact on worst-case performance. Another direction is to develop techniques to better select the code snippets to be outlined: outlining has a cost (extra function call and parameter passing) that has to be avoided when outlining is not beneficial to the WCET. Symmetrically, we still need to explore which code sequences would benefit from being outlined and *not* optimized, such that manual source-level annotations can be given when more beneficial to WCET estimates than compiler optimizations and automatic flow fact extraction.

References

- 1 aiT:the industry standard for static timing analysis. <http://www.absint.com/ait>.
- 2 Felix V. Agakov, Edwin V. Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael F. P. O'Boyle, John Thomson, Marc Toussaint, and Christopher K. I. Williams. Using machine learning to focus iterative optimization. In *Fourth IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2006), 26-29 March 2006, New York, New York, USA*, pages 295–305, 2006.
- 3 Guillem Bernat and Niklas Holsti. Compiler support for WCET analysis: a wish list. In *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*, pages 65–69, 2003.
- 4 Yang Chen, Shuangde Fang, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Olivier Temam, and Chengyong Wu. Deconstructing iterative optimization. *TACO*, 9(3):21:1–21:30, 2012. doi:10.1145/2355585.2355594.
- 5 Keith Cooper and Linda Torczon. *Engineering a compiler*. Morgan Kaufmann, 2011.
- 6 Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. ACME: adaptive compilation made efficient. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05), Chicago, Illinois, USA, June 15-17, 2005*, pages 69–77, 2005.
- 7 Christoph Cullmann and Florian Martin. Data-Flow Based Detection of Loop Bounds. In Christine Rochange, editor, *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*, volume 6 of *OpenAccess Series in Informatics (OASISs)*, Dagstuhl, Germany, 2007. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASISs.WCET.2007.1193.
- 8 Steven Derrien, Isabelle Puaut, Panayiotis Alefragis, Marcus Bednara, Harald Bucher, Clement David, Yann Debray, Umut Durak, Imen Fassi, Christian Ferdinand, Damien Hardy, Angeliki Kritikakou, Gerard K. Rauwerda, Simon Reder, Martin Sicks, Timo Stripf, Kim Sunesen, Timon D. ter Braak, Nikolaos S. Voros, and Jürgen Becker. Wcet-aware parallelization of model-based applications for multi-cores: The ARGO approach. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, pages 286–289, 2017.
- 9 Heiko Falk, Peter Marwedel, and Paul Lokuciejewski. Reconciling compilation and timing analysis. In *Advances in Real-Time Systems (to Georg Färber on the occasion of his appointment as Professor Emeritus at TU München after leading the Lehrstuhl für Realzeit-Computersysteme for 34 illustrious years)*., pages 145–170, 2012.
- 10 A. Floc'h, T. Yuki, A. El-Moussawi, A. Morvan, K. Martin, M. Naullet, M. Alle, L. L'Hours, N. Simon, S. Derrien, F. Charot, C. Wolinski, and O. Sentieys. Gecos: A framework for prototyping custom hardware design flows. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 100–105, Sept 2013.
- 11 Raimund Kirner, Peter P. Puschner, and Adrian Prantl. Transforming flow information during code optimization for timing analysis. *Real-Time Systems*, 45(1-2):72–105, 2010. doi:10.1007/s11241-010-9091-8.
- 12 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization*, pages 75–88, San Jose, CA, USA, Mar 2004.
- 13 Thomas Lefeuvre, Imen Fassi, Christoph Cullmann, Gernot Gebhard, Emin Koray Kasnakli, Isabelle Puaut, and Steven Derrien. Using polyhedral techniques to tighten wcet estimates of optimized code: a case study with array contraction. In *Design, Automation*

8 Test in Europe Conference 8 Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018, 2018.

- 14 Hanbing Li, Isabelle Puaut, and Erven Rohou. Traceability of flow information: Reconciling compiler optimizations and wcet estimation. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems, RTNS '14*, pages 97:97–97:106, New York, NY, USA, 2014. ACM.
- 15 Hanbing Li, Isabelle Puaut, and Erven Rohou. Tracing flow information for tighter WCET estimation: Application to vectorization. In *21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2015, Hong Kong, China, August 19-21, 2015*, pages 217–226, 2015.
- 16 Chunhua Liao, Daniel J Quinlan, Richard Vuduc, and Thomas Panas. Effective source-to-source outlining to support whole program empirical optimization. In *Languages and Compilers for Parallel Computing*, pages 308–322. Springer, 2010.
- 17 Paul Lokuciejewski, Sascha Plazar, Heiko Falk, Peter Marwedel, and Lothar Thiele. Approximating pareto optimal compiler optimization sequences - a trade-off between wcet, ACET and code size. *Softw., Pract. Exper.*, 41(12):1437–1458, 2011. doi:10.1002/spe.1079.
- 18 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008.