12th International Workshop on Worst-Case Execution Time Analysis

WCET'12, July 10, 2012, Pisa, Italy

Edited by

Tullio Vardanega

OASICS

Editor

Tullio Vardanega
Department of Mathematics
University of Padova, Italy
tullio.vardanega@math.unipd.it

ACM Classification 1998
C.3 [Special-Purpose and Application-Based Systems]: Real-time and embedded systems, C.4 [Performance of systems]: Modelling techniques, Performance attributes, D.2.4 Software/Program Verification, D.2.5 Testing and Debugging, D.2.11 [Software Engineering]: Software Architectures – Domain-specific architectures, Patterns, D.4.1 [Operating System]: Process Management, F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages – Program Analysis

ISBN 978-3-939897-41-5

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/978-3-939897-41-5.

Publication date
July 2012

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.

Digital Object Identifier: 10.4230/OASIcs.WCET.2012.i

ISSN 2190-6807

www.dagstuhl.de/oasics

Contents

12th International Workshop on Worst-Case Execution Time Analysis (WCET 2012).
Editor: Tullio Vardanega
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

I take real pleasure in seeing the proceedings of the 12th International Workshop on Worst-Case Execution Time Analysis online already on the day of the workshop. This helps WCET'12 achieve its goal of facilitating discussion and interaction among participants as well as of returning value to the authors of the works that were accepted for presentation. I also feel personal satisfaction in having achieved the production of these proceedings as a tangible manifestation of the considerable effort that went in making WCET'12 happen, and successfully so, in fact.

The WCET workshop is a successful series indeed. The research community active in WCET analysis evidently cares for the event, values its venue and atmosphere, and the relevance of its proceedings. In that respect, it is a comparatively easy job to be the program chair for it (hopefully my predecessors would not feel diminished by me saying so!) in as far as the harvesting of valuable contributions goes. I was very pleased and reassured at seeing the whole program committee actively help me disseminate the call for papers, scout for good research projects that would be at the stage of maturity to present their ideas in the workshop, and turn in very thorough reviews.

We received 23 good-quality submissions, of which we selected 10 for the program and the proceedings. We had the luxury of being selective, and the opportunity of putting together a solid program that makes ample room for discussion and interaction, which is what the workshop is for in the first place.

I welcome all participants to WCET'12 in both the physical event, taking place as usual as a satellite event to ECRTS12, this year on 10 July at the beautiful venue of Scuola Sant'Anna in Pisa, Italy, and the online proceedings, which I hope will get the amount of citations that the authors need for their good work.

In closing, I extend my gratitude for the members of the Program Committee, which you see listed on the next page.

Padova, 25 June 2012
Tullio Vardanega

Workshop Organization

WCET'12 Program Committee

Iain Bate	University of York, UK
Guillem Bernat	Rapita Systems Ltd, UK
Francisco Cazorla	Barcelona Supercomputing Center, Spain
Heiko Falk	Ulm University, Germany
Jan Gustafsson	Maelardalen University, Sweden
Kevin Hammond	University of St Andrews, UK
Chris Healy	Furman University, USA
Vesa Hirvisalo	Aalto University, Finland
Niklas Holsti	Tidorum Ltd, Finland
Raimund Kirner	University of Hertfordshire, UK
Björn Lisper	Mälardalen University, Sweden
Stefan Petters	ISEP - IPP, Portugal
Isabelle Puaut	University of Rennes 1 / IRISA, France
Martin Schoeberl	Technical University of Denmark
Christine Rochange	IRIT, France
Reinhard Wilhelm	Universitaet des Saarlandes, Germany
Wang Yi	Uppsala University, Sweden

WCET Steering Committee

Guillem Bernat	Rapita Systems Ltd., UK
Jan Gustafsson	Mälardalen, Sweden
Peter Puschner	Vienna University of Technology, Austria

List of Authors

Alejandro Alonso
Real-Time Systems group (STRAST)
Universidad Politécnica de Madrid (UPM), Spain
aalonso@dit.upm.es

Andrea Baldovin
University of Padua, Department of Mathematics
via Trieste, 63 35121 Padua, Italy
baldovin@math.unipd.it

Johann Blieberger TU Vienna, Institute of
Computer-Aided Automation
Treitlstr. 1-3, 1040 Vienna, Austria
blieb@auto.tuwien.ac.at

Daniel Brosnan
Real-Time Systems group (STRAST)
Universidad Politécnica de Madrid (UPM), Spain
dbrosnan@datsi.fi.upm.es

Franck Cassez
National ICT Australia
Sydney, Australia
Franck.Cassez@nicta.com.au

Jorge Garrido
Real-Time Systems group (STRAST)
Universidad Politécnica de Madrid (UPM), Spain
jgarrido@datsi@fi.upm.es

Jan Gustafsson
School of Innovation Design and Engineering
Mälardalen University, Sweden
jan.gustafsson@mdh.se

Andreas Gustavsson
School of Innovation Design and Engineering
Mälardalen University, Sweden
andreas.sg.gustavsson@mdh.se

René Rydhof Hansen
Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300, DK-9220 Aalborg, Denmark
rrh@cs.aau.dk

Benedikt Huber
Institute of Computer Engineering
Vienna University of Technology, Austria
benedikt@vmars.tuwien.ac.at

Björn Lisper
School of Innovation Design and Engineering
Mälardalen University, Sweden
bjorn.lisper@mdh.se

Mohamed Abdel Maksoud
Saarland University
Saarbrücken, Germany
mohamed@cs.uni-saarland.de

Amine Marref
Department of Computer Science, Umm Al-Qura
University
Makkah, Saudi Arabia
ajmarref@uqu.edu.sa

Enrico Mezzetti
University of Padua, Department of Mathematics
via Trieste, 63 35121 Padua, Italy
emezzett@math.unipd.it

Robert Mittermayr, TU Vienna, Institute of
Computer-Aided Automation
Treitlstr. 1-3, 1040 Vienna, Austria
robert@auto.tuwien.ac.at

Mads Chr. Olesen
Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300, DK-9220 Aalborg, Denmark
mchro@cs.aau.dk

Daniel Prokesch
Institute of Computer Engineering
Vienna University of Technology, Austria
daniel@vmars.tuwien.ac.at

Juan A. de la Puente
Real-Time Systems group (STRAST)
Universidad Politécnica de Madrid (UPM), Spain
jpuente@dit.upm.es

Peter Puschner
Institute of Computer Engineering
Vienna University of Technology, Austria
peter@vmars.tuwien.ac.at

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

Tullio Vardanega
University of Padua, Department of Mathematics
via Trieste, 63 35121 Padua, Italy
tullio.vardanega@math.unipd.it

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

Juan Zamorano
Real-Time Systems group (STRAST)
Universidad Politécnica de Madrid (UPM), Spain
jzamora@datsi.fi.upm.es

What is a Timing Anomaly?

Franck Cassez[1], René Rydhof Hansen*[2], and Mads Chr. Olesen[2]

1 **National ICT Australia**
Sydney, Australia
`Franck.Cassez@nicta.com.au`

2 **Department of Computer Science, Aalborg University**
Selma Lagerlöfs Vej 300, DK-9220 Aalborg, Denmark
`{rrh,mchro}@cs.aau.dk`

Abstract

Timing anomalies make worst-case execution time analysis much harder, because the analysis will have to consider all local choices. It has been widely recognised that certain hardware features are timing anomalous, while others are not. However, defining formally what a timing anomaly is, has been difficult.

We examine previous definitions of timing anomalies, and identify examples where they do not align with common observations. We then provide a definition for *consistently slower hardware traces* that can be used to define timing anomalies and aligns with common observations.

1998 ACM Subject Classification C.4 [Performance of systems]: Modelling techniques, Performance attributes

Keywords and phrases Timing anomalies, worst case execution time (WCET), abstractions

Digital Object Identifier 10.4230/OASIcs.WCET.2012.1

1 Introduction

Developing reliable real-time systems requires that guarantees on the run-time of tasks can be given, that hold under all circumstances i.e. regardless of the input data and previous execution history of the system. Typically the Worst-Case Execution Time (WCET) is the most important guarantee as it can be used to ensure the system responds in a timely manner.

However, modern processors are not optimized for worst cases, but optimize for improving the average case performance instead. This often makes their worst-case behaviour much harder to predict, and thus makes it harder to give absolute guarantees. One often hoped for property is that local worst-case timing choices will lead to the global worst-case timing — when this is not the case it is dubbed a *timing anomaly*. The classic example of a timing anomaly [6] is shown in Figure 1, where a cache miss for instruction A (bottom) is locally slower but turns out not to be the globally slowest (the top trace is slower). The example will be treated in greater detail later.

If an execution platform can be proven to be free of timing anomalies, very efficient techniques exist for analysing the worst-case timing behaviour [11]. On the contrary, if the execution platform exhibits timing anomalies there is little hope for using the same efficient abstraction techniques [6].

* Author partially funded by the "Certifiable Java for Embedded Systems" (CJ4ES) project (Danish Research Council for Technology and Production, grant number 10-083159).

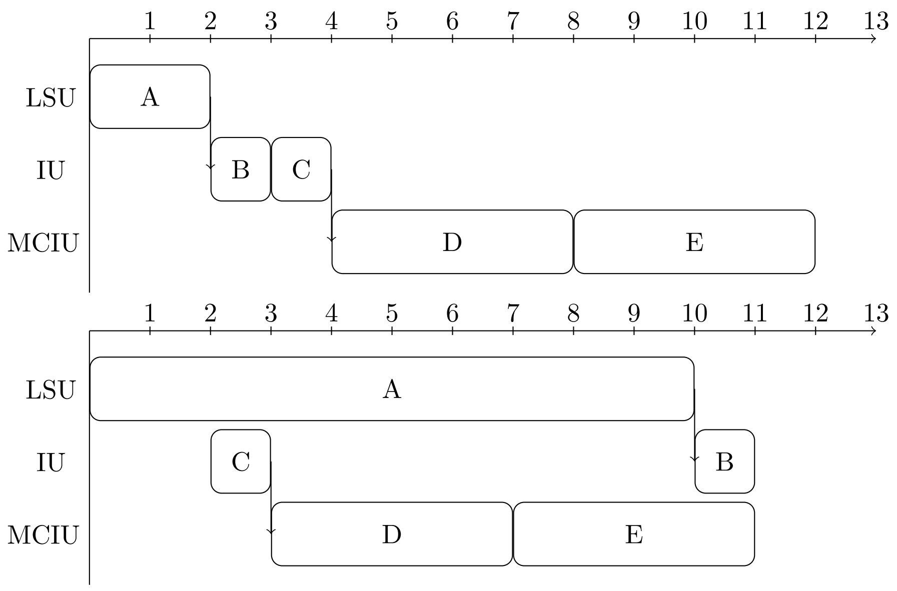


Figure 1 The canonical example of a timing anomaly from [6], where a cache miss (locally slower) leads to a scheduling that is globally faster. LSU, IU and MCIU are the three functional units that can execute out-of-order, but preference is given to older instructions.

Because of this, identifying timing anomalies has been an area of interest for some time, and some observations have been broadly recognised as being true:

- The LRU cache replacement policy is not timing anomalous.
- Other cache replacement policies such as FIFO and MRU exhibit timing anomalies [2, 4].
- In-order pipelines (without caches) are not timing anomalous.
- Resource allocation decisions (such as those presented by out-of-order execution or cache replacement) are a necessary condition for timing anomalies [10].

Using efficient abstraction techniques to compute the WCET is at the core of WCET analysis tools. However, the most powerful abstractions are sound only for timing anomaly free hardware. This explains why there have been some attempts to formally define timing anomalies [6, 9], but the various definitions have not been related to each other thus far.

In this work we will argue that the previous attempts are either too coarse or too precise to be used as universal definitions of timing anomalies. Each of the previous attempts definitely have their merits for application in connection with different analysis techniques (abstract interpretation, etc.), but a definition of timing anomalies should be as general as possible, while still retaining the property that the existence of timing anomalies forces the WCET analysis to consider more than one local choice.

Our Contribution

Our work is guided by the need for a definition of timing anomalies on the concrete model of the processor, instead of abstractions thereof. Consequently, in the following we propose a definition of timing anomalies that can be used in two different directions:

1. on hardware systems that are proven to be free of timing anomalies, the efficient abstraction techniques used in most WCET analysis tools are sound;

2. the definition we propose is based on the concrete reference hardware and only relates comparable hardware traces in order to avoid spurious timing anomalous diagnostics (see Section 4.2) resulting from abstraction of the hardware and/or of the hardware traces.

Without a definition of timing anomalies on the concrete reference hardware model, it is impossible to prove that abstraction is sound. Therefore we define timing anomalies as a property over different traces of the concrete hardware model.

But what traces should be comparable? We will argue that only traces resulting in the same instruction stream, i.e. the same program execution, should be comparable, in particular traces produced by different input data should not be comparable. It seems natural that different input data can result in different control flows, and therefore different instruction streams, where small changes in the input can result in much longer instruction streams, and therefore much longer execution times.

Another consideration is what elements of the hardware traces should be compared. Previous definitions have compared the timing of the first instruction with the timing of the last instruction in the stream [6], or made comparisons at points where the two traces have executed the same number of instructions [3]. We will argue that comparisons should be made on the *completion times* of each instruction.

Outline of the Paper

This work is divided into five sections: In Section 2 we define hardware systems and execution of programs on them. In Section 3 we define timing anomalies, and then examine related work in Section 4. We then compare the different definitions in Section 5, before concluding in Section 6.

2 Execution of Programs on Hardware

Before turning to timing anomalies, we first need to formalise our notion of hardware systems and how programs are executed on them. In order for our work to be applicable to a wide variety of systems, we aim to make as few assumptions about the hardware as possible. However, it usually consists of a processor and main memory (including caches). As usual, the hardware can process (machine code) instructions, taken from the set $\mathsf{Instructions}$, with each instruction located in memory at some address. We let $\mathsf{HardwareStates}$ be the (finite) set of possible hardware states and assume the hardware states contain the state of the memory.

The *semantics* of a hardware system is given by a transition system that specifies how the state of the hardware evolves in order to execute a program on given input data. We only model transitions between hardware states that take an observable amount of time and produce an observable result[1], e.g., finishing execution of a (set of) instruction(s).

The observable results, in the set $\mathsf{Observations}$, are not strictly necessary but are admitted as a convenience for later developments. In our work, the typical observations of interest in a given hardware system are the instructions that finish (in each cycle or time unit). We can now give the formal definition of a hardware system.

▶ **Definition 1** (Hardware System). A *hardware system* $\mathcal{H}$ is formalised as a *stutter-free* and *deterministic* labelled transition system $\mathcal{H} = (\mathsf{HardwareStates}, \mathsf{Time} \times \mathsf{Observations}, \rightarrow)$. The

[1] Bus latency, speculative execution, pipeline flushes, etc., are not visible and may generate extra cycles before an externally visible hardware state occurs.

transition relation $\rightarrow \subseteq \mathsf{HardwareStates} \times (\mathsf{Time} \times \mathsf{Observations}) \times \mathsf{HardwareStates}$ describes the *time* required to reach the next state and the externally visible *observations* produced by a transition.

As usual, a transition $(s, (t, o), s') \in\rightarrow$ is denoted $s \xrightarrow{(t,o)} s'$. The properties "stutter-free" and "deterministic" can then be formulated as follows: if $s \xrightarrow{(t,o)} s'$ then $s \neq s'$, and if $s \xrightarrow{(t,o)} s'$ and $s \xrightarrow{(t',o')} s''$ then $t = t', o = o'$ and $s' = s''$. A *run* in the hardware system $\mathcal{H}$ is defined to be a sequence $\sigma = s_0 \xrightarrow{(t_1,o_1)} s_1 \xrightarrow{(t_2,o_2)} \cdots \xrightarrow{(t_n,o_n)} s_n$ such that for all $1 \leq i \leq n-1$ it holds that $s_i \xrightarrow{(t_{i+1},o_{i+1})} s_{i+1}$ (in the $\mathcal{H}$ transition system) and the *length* of the run is defined as $\mathit{length}(\sigma) = n$. The *trace* of the run σ is $\mathit{trace}(\sigma) = (t_1, o_1) : (t_2, o_2) : \cdots : (t_n, o_n)$; the *time trace* of σ is $\mathit{time}(\sigma) = t_1 : \cdots : t_n$, and the *observation trace* of σ is $\mathit{obs}(\sigma) = o_1 : o_2 : \cdots : o_n$.

▶ **Example 2.** Observing the two traces shown in Figure 1 using "just finished instructions" as observations, we obtain the following run for the first (top) part of the example:

$$h_0 \xrightarrow{(2,\{A\})} h_1 \xrightarrow{(1,\{B\})} h_2 \xrightarrow{(1,\{C\})} h_3 \xrightarrow{(4,\{D\})} h_4 \xrightarrow{(4,\{E\})} h_5$$

and the run below for the second (lower) example in Figure 1:

$$h_0 \xrightarrow{(3,\{C\})} h_1 \xrightarrow{(4,\{D\})} h_2 \xrightarrow{(3,\{A\})} h_3 \xrightarrow{(1,\{B,E\})} h_4$$

Note that the observation on the final transition above shows that the two instructions labelled B and E finish simultaneously.

2.1 Execution of a Program on Hardware

We assume that all programs terminate. Given a program P and input data d in $\mathsf{Data}(P)$ (the set of admissible input data for P), the language semantics uniquely determine the *program trace*, i.e. the sequence of instructions to be performed to compute the result of program P on input d. In the following we need to be able to unambiguously identify specific occurrences of instructions in a program trace (the same instruction can be performed several times in the trace, for instance when loops are executed). Thus we formalise the program trace for (P, d) to be a mapping that assigns a unique index to each instruction in the program trace: $\mathsf{ProgramTrace}(P, d) : [1..k] \rightarrow \mathsf{Instructions}$ where k is the length of the trace and $\mathsf{ProgramTrace}(P, d)(j)$ gives the instruction executed at step j for each index $1 \leq j \leq k$.

▶ **Example 3.** The program trace for the example in Figure 1 is: $\mathsf{ProgramTrace}(P, d) = [1 \mapsto A, 2 \mapsto B, 3 \mapsto C, 4 \mapsto D, 5 \mapsto E]$.

Given a hardware system $\mathcal{H}$, a program P and input data $d \in \mathsf{Data}(P)$, we let $I(P, d) \subseteq \mathsf{HardwareStates}$ be the hardware states that contain program P and input data d in memory, and where the first instruction of P is about to start execution. For $h_0 \in I(P, d)$, executing P with input d on $\mathcal{H}$ yields a unique sequence[2] of transitions in the hardware: $h_0 \xrightarrow{(t_1,o_1)} h_1 \cdots h_{n-1} \xrightarrow{(t_n,o_n)} h_n$. As the hardware is deterministic, each observation o_i can be taken to be a set of indices in $\{1, \ldots, k\}$: the indices uniquely identify the occurrences of instructions of $\mathsf{ProgramTrace}(P, d)$ being completed at each step. We can now formalise what it means to execute a program on a hardware system:

[2] Which we assume to be correct with regard to the instruction semantics.

▶ **Definition 4** ((P, d, h_0)-run)**.** Let $\mathcal{H}$ be a hardware system, P a program, d input data, and $h_0 \in \mathsf{HardwareStates}$. Then the run (in $\mathcal{H}$) $h_0 \xrightarrow{(t_1,o_1)} h_1 \cdots h_{n-1} \xrightarrow{(t_n,o_n)} h_n$ is called a (P, d, h_0)*-run* (in $\mathcal{H}$) whenever $h_0 \in I(P, d)$ and o_i is the set of (indices of) instructions completed during the transition $h_{i-1} \xrightarrow{(t_i,o_i)} h_i$ for $1 \leq i \leq n$.

▶ **Definition 5** (Completion Time)**.** Let $\mathsf{ProgramTrace}(P, d) : [1..k] \to \mathsf{Instructions}$ be the program trace of (P, d) and let σ be the corresponding (P, d, h)-run starting in $h \in I(P, d)$ with $trace(\sigma) = (t_1, o_1) : \cdots : (t_n, o_n)$. By $Ctime(\mathsf{ProgramTrace}(P, d)[j], h)$ we denote the *completion time* of instruction $1 \leq j \leq k$ finishing after transition m (i.e., $j \in o_m$) and define it as follows $Ctime(\mathsf{ProgramTrace}(P, d)[j], h) = \sum_{i=1}^{m} t_i$.

We let $Ctime(\mathsf{ProgramTrace}(P, d), h) = \max_{1 \leq j \leq k} Ctime(\mathsf{ProgramTrace}(P, d)[j], h)$ denote the maximal completion time for all instructions in the program and thus for completing the entire program.

▶ **Example 6.** The first trace in the example in Figure 1 has the following completion times for the 5 instructions: $[2, 3, 4, 8, 12]$ and for the second trace: $[10, 11, 3, 7, 11]$.

2.2 Exemplary Hardware Models

To be able to exemplify different phenomena we will use three different hardware models:

M_1 is a single-stage pipeline with a data-cache. The instructions of interest are the memory accesses, and the hardware model will be used to demonstrate timing anomalies with different cache replacement policies such as LRU and FIFO. For this reason we will simply denote instructions by the memory address they access.

M_2 is the simplified PowerPC architecture described in [6]. It is an out-of-order processor with three functional units: a Load/Store unit (LSU) communicating with a data cache, a Multi-Cycle Integer Unit (MCIU) and an Integer Unit (IU). For a detailed description see [6]. It is used for the classic example in Figure 1.

M_3 is a single-stage pipeline with no caches, but with a multiplication instruction `MUL` that takes 1 cycle if one of the operands is 0, and 2 cycles otherwise. This is a simplified version of the processor model in [3]. We extend M_3 with conditional execution of all instructions as on the ARM architecture [1].

3 Formalising Timing Anomalies

Slightly simplified, our notion of timing anomaly is based on the idea that timing anomalies only occur when a program is executed on a hardware system where no initial state gives rise to worse (slower) execution time than all other initial hardware states (modulo "irrelevant" parts of the hardware state). This approach requires us to formalise what it means for one program execution to be slower than, or rather: consistently as slow as, another execution (of the same program on the same data):

▶ **Definition 7** (Consistently as slow)**.** Let P be a program with input data d and let $\mathsf{ProgramTrace}(P, d) : [1..k] \to \mathsf{Instructions}$ be the program trace of (P, d). Let $h, h' \in I(P, d)$ and σ (respectively σ') be a (P, d, h)-run (respectively (P, d, h')-run). Then h' is said to be *consistently as slow* as h, denoted $h \sqsubseteq_{time} h'$, if and only if

$$\forall 1 \leq j \leq k \colon Ctime(\mathsf{ProgramTrace}(P, d)[j], h) \leq Ctime(\mathsf{ProgramTrace}(P, d)[j], h')$$

Intuitively the above definition compares the execution time of all prefixes of a program trace and requires one to be consistently as slow as the other.

▶ **Example 8.** Consider the example in Figure 1, where h is the hardware state resulting in the top run, and h' the state resulting in the bottom run.

$$Ctime(\mathsf{ProgramTrace}(P,d)[1],h) = 2 \leq Ctime(\mathsf{ProgramTrace}(P,d)[1],h') = 10$$

meaning instruction A was slower in the bottom trace, but

$$Ctime(\mathsf{ProgramTrace}(P,d)[5],h) = 12 \nleq Ctime(\mathsf{ProgramTrace}(P,d)[5],h') = 11$$

meaning instruction E was not slower in the top trace, thus $h \not\sqsubseteq_{time} h'$. However instruction A in the bottom trace is still slower than in the top trace, so $h' \not\sqsubseteq_{time} h$.

The "consistently as slow" ordering is a pre-order (see below). However, it is not a partial order since two hardware states, which differ only in parts that are irrelevant to a given program, will still give rise to identical instruction completion times:

▶ **Lemma 9.** *For all programs P and input data d the relation $\sqsubseteq_{time}$ is a pre-order on $I(P,d)$.*

We can now propose a formal definition for timing anomalies:

▶ **Definition 10** (Timing Anomaly Free)**.** A hardware system, $\mathcal{H}$, is said to be *free of timing anomalies* with respect to program P and input d, if and only if there exists a maximal element, $\mathcal{W} \in I(P,d)$: $\forall h \in I(P,d)\colon h \sqsubseteq_{time} \mathcal{W}$.

Note that the maximal element is not necessarily unique: consider the case of LRU caches with no useful elements in them, hence all references resulting in cache misses.

The following lemma characterises (the absence of) timing anomalies in terms of upper bounds for arbitrary pairs of states. As shown in Section 5, this characterisation can be convenient when proving the presence of timing anomalies.

▶ **Lemma 11.** *A hardware system $\mathcal{H}$ is free of timing anomalies with respect to program P and input data d if and only if $\forall h, h' \in I(P,d)\colon \exists h'' \in I(P,d)\colon h \sqsubseteq_{time} h'' \wedge h' \sqsubseteq_{time} h''$.*

Proof. The "only if" direction is trivial and the "if" direction is proved by induction in the size of $I(P,d)$. ◀

Note that this does not require all hardware states to be ordered under $\sqsubseteq_{time}$, but only requires that for any pair of states a third state exists that gives rise to a consistently as slow run as both; i.e. it should be an upper bound for the pair of states.

Consider the example in Figure 2 where two runs of a LRU cache are not ordered either way, but we would still like to characterise LRU as not timing anomalous; there exists a consistently slower initial state than both, namely the empty cache.

Having defined timing anomaly free-ness for a given program and input data, it is straightforward to generalise the definition to cover entire hardware systems:

▶ **Definition 12** (Timing Anomaly Free Hardware System)**.** A hardware system, $\mathcal{H}$, is said to be *free of timing anomalies* for program P if and only if it is timing anomaly free for each $d \in \mathsf{Data}(P)$. Hardware $\mathcal{H}$ if free of timing anomalies if and only if it is timing anomaly free for all programs P (valid for $\mathcal{H}$).

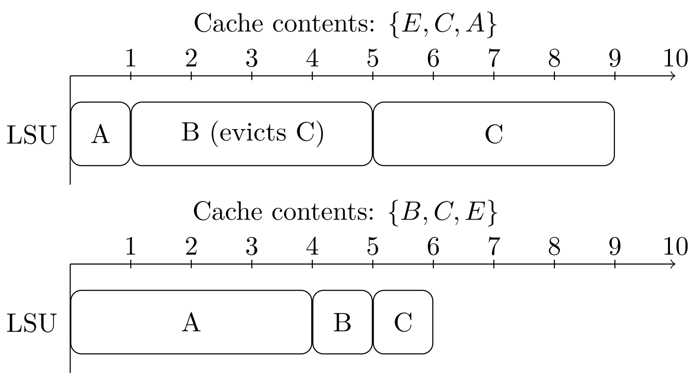


Figure 2 Two runs of the program `LD A; LD B; LD C` on hardware model M_1 with a LRU cache. The cache contents are ordered sets of data elements, from newest to oldest.

Finally we relate our definition of "consistently as slow as" to the definition of the WCET for a program P on hardware $\mathcal{H}$.

▶ **Definition 13** (Worst Case Execution Time (WCET)). The *worst case execution time* for a program P (on $\mathcal{H}$) is defined as follows:

$$WCET_{\mathcal{H}}(P) = \max_{h \in I(P,d), d \in \mathsf{Data}(P)} \{Ctime(\mathsf{ProgramTrace}(P, d), h)\}$$

If $\mathcal{H}$ is free of timing anomalies for P, only a maximal element in $I(P, d)$ need be considered. Indeed, if $h \sqsubseteq_{time} h'$, then $Ctime(\mathsf{ProgramTrace}(P, d), h) \leq Ctime(\mathsf{ProgramTrace}(P, d), h')$. Definition 13 is then reduced to computing $\max_{d \in \mathsf{Data}(P)}\{Ctime(\mathsf{ProgramTrace}(P, d), h) \mid h$ maximal in $I(P, d)\}$.

4 Related Work

4.1 Defining Timing Anomalies by Changes in Instruction Latency

Timing anomalies were first discovered by Lundqvist and Stenström in [6, 5]. Their definition is re-used in [10], and we formulate it here in our framework.

Assume a sequence of instructions $\pi = i_1 : \cdots : i_n$, with corresponding latencies $\tau_\pi(i_j)$, and total execution time C. Consider a situation where there exists a latency variation, Δt, such that the same sequence of instructions, but with a modified latency for the first instruction $\tau'_\pi(i_1) = \tau_\pi(i_1) + \Delta t$, results in a different sequence of instruction latencies $\tau_\pi(i_1) + \Delta t : \tau_\pi(i_2) : \cdots : \tau_\pi(i_n)$ and thus a possible different total execution time C', and thus a timing difference of $\Delta C = C' - C$.

▶ **Definition 14** (Timing Anomalies by Changes in Instruction Latency [10]). A timing anomaly is defined as a situation where according to the sign of Δt one of the following cases become true:
a) Increase of the latency: $\Delta t > 0 \implies (\Delta C > \Delta t) \vee (\Delta C < 0)$
b) Decrease of the latency: $\Delta t < 0 \implies (\Delta C < \Delta t) \vee (\Delta C > 0)$
This definition has some drawbacks:
- As pointed out in [9] there is an underlying assumption that the latency change of the first instruction does not influence the latencies of the subsequent instructions. This is not always the case.

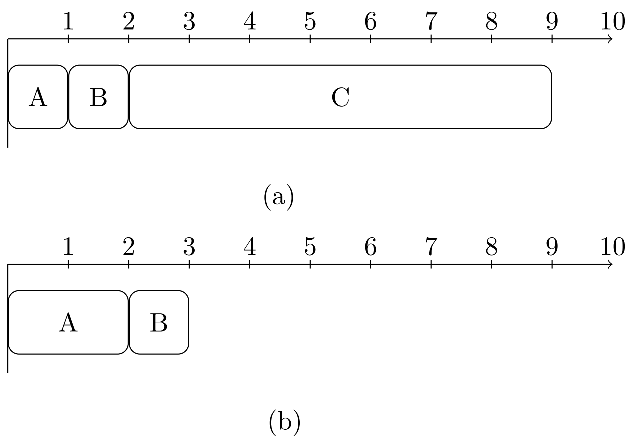


Figure 3 Example program run on M_3. The program is `A: MUL` R_0, R_0, R_1; `B: BRZ` R_0, C, where C is a linear, data-independent, subprogram summarised into one instruction. The instruction `BRZ` is interpreted as "branch to C if R_0 is zero". The two traces are (a) where $R_0 = 0$, and (b) where $R_0 = 1$.

- In [6] the change in latency can be unrelated to a change in hardware state, resulting in the definition deeming a hardware platform to suffer from timing anomalies, while the actual platform does not.

In [10] the second point is alleviated as the change in latency is assumed to be associated to two different initial hardware states, which are further assumed to be "almost identical". However without a formalisation of "almost identical" it could be argued that the two LRU caches in Figure 2 are almost identical, and thus would be deemed timing anomalous.

4.2 Defining Timing Anomalies by Abstract Models

In [9] a formal definition of timing anomalies is given. It however states that "Non-determinism – which is necessary for timing anomalies – is only introduced by abstraction", and goes on to define timing anomalies in terms of non-deterministic hardware models. We note that timing anomalies as demonstrated originally in [6] do not involve non-determinism, but instead involve concrete traces run on the same concrete (deterministic) hardware model.

We will argue that non-determinism is *not* necessary for timing anomalies to occur. Indeed, there is a strong link between the presence of timing anomalies and the existence of a sound deterministic over-approximating model of the timing of the hardware, but the causality is not both ways. In some cases timing anomalies can even occur as artefacts of the abstraction, even though they are not present in the concrete hardware.

As an example consider the two traces in Figure 3. Here different input data results in very different instruction streams, sharing similarities with timing anomalous traces. In our opinion this should not be characterised as a timing anomaly, as this would render practically all programs accepting input on all platforms to be timing anomalous. The definition in [9] requires that the two traces must have the same instruction streams, and thus Figure 3 is not a timing anomaly by that definition.

We however note that the same instruction stream can still result in two very different timing behaviours, when given different input data. As an example, the ARM architecture allows the conditional execution of most instructions. We can thus derive an example where the same instruction stream gives rise to timing anomalous behaviour on different input data,

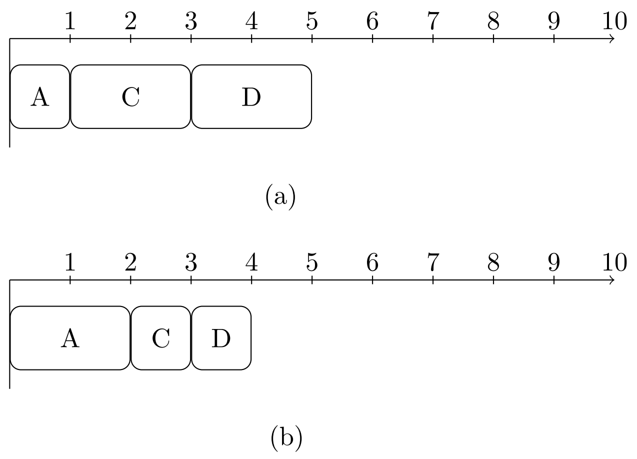


Figure 4 The example from Figure 3, but instead of branching it uses conditional execution. Therefore the two instruction streams are the same, but the processor treats C as a no-op in (b). The program is `A: MUL` R_0, R_0, R_1; `C: MULNZ` R_2, R_2, R_1; `D: MULNZ` R_2, R_2, R_1 , where we let A set the condition flags. Thus C and D only gets executed if R_0 is not 0.

as seen in Figure 4.

According to the definition in [9] this would be timing anomalous, as there exists a non-local worst-case path (the A instruction in (a)) resulting in a globally longer path, than all local worst-case paths (b).

In [8] a slightly relaxed definition from [9] is given, which is used to compute upperbounds on the the possible error in WCET estimation between two hardware states. However, with regards to the examples we consider there is no difference.

In the same line [3] describes a method to identify timing anomalies in a processor using bounded model checking. This is done by comparing the execution time of the same instruction stream on two different processors: the real processor, and an (abstract) “always-worst case” performing processor. If the “worst-case” processor can overtake the real processor, the processor is deemed to have timing anomalies.

However [3] uses abstraction of input data and thus ends up comparing execution traces which can only result from different input data: The trace given in [3] (a) cannot occur on the real processor with the same input data as the trace in (b). For trace (a) to occur one of the operands (R_4 and R_6 in this case) needs to be 0, that is $R_4 = 0 \vee R_6 = 0$. However for trace (b) to occur none of the operands can be 0, thus $R_4 \neq 0 \wedge R_6 \neq 0$. As these two conditions are the negation of one another, the two traces cannot occur with the same input data. Since the two traces cannot occur with the same input data, they will be incomparable, per our Definition 7. Of course, hardware can be viewed abstractly. For timing analysis it is very important that these abstractions are sound, i.e. overapproximating the timing.

▶ **Definition 15** (More Favorable Hardware). Hardware $\mathcal{H}$ with hardware states $\mathsf{HardwareStates}$ is more favorable than hardware $\mathcal{H}'$ with hardware states $\mathsf{HardwareStates}'$, written $\mathcal{H} \sqsubseteq \mathcal{H}'$, if there exists a mapping $\alpha : \mathsf{HardwareStates} \to \mathsf{HardwareStates}'$ $\forall P, \forall d \in \mathsf{Data}(P) : h \sqsubseteq_{time} \alpha(h)$, where $\sqsubseteq_{time}$ is extended across different hardware systems.

Typically α is an abstraction function. Clearly, if $\mathcal{H} \sqsubseteq \mathcal{H}'$, then for any program P, $WCET_{\mathcal{H}}(P) \leq WCET_{\mathcal{H}'}(P)$. $\mathcal{H}'$ is thus a sound abstraction for computing the WCET of any program. The technique presented in [3] is a very valuable tool in finding sound abstractions, however, the unsoundness of an abstraction cannot be translated into the real hardware exhibiting timing anomalies.

5 Results

We will now compare the different definitions of timing anomalies, and how they hold for and apply to different examples:

	Our Definitions 10, 12	Latency change [6, 10]	Abstraction [3, 9, 8]
Classic Ex. [6] (Fig. 1)	Yes, Lemma 16	Yes	Yes
LRU Cache	No, Lemma 19	Inapplicable[1)]	No
FIFO Cache	Yes, Lemma 20	Inapplicable[1)]	Yes[4)]
Branching (Fig. 3)	No, Lemma 17	Inapplicable[2)]	No[4)]
Conditional exec. (Fig. 4)	No, Lemma 18	Yes	Yes[4)]
MUL 0-speedup [3, Fig. 3]	No	Yes[3)]	Yes[4)]

1) Because the latency change can in general not be limited to, or contained within, the first instruction.
2) Because the sequence of instructions are not the same.
3) If we allow the latency change to occur on the second instruction.
4) Depends on the abstraction.

▶ **Lemma 16.** *The classic example in Figure 1 is timing anomalous by Definition 10.*

Proof. We will show that none of the two initial hardware states is consistently worse than the other, per Definition 7, and thus no upper bound can exist, per Definition 10. The only two initial hardware states that are relevant to consider is the cache where the data item referenced by A is in the cache, and a state where the data item referenced by A is not in the cache. Since there are only two initial hardware states, one or both of them would have to be consistently slower than the other. In Example 8 we already showed that none of the two traces is consistently slower than the other. Therefore, Definition 10 cannot be fulfilled. ◀

▶ **Lemma 17.** *The control-flow example in Figure 3 is not timing anomalous by Definition 10.*

Proof. Since M_3 has no cache, there is actually only one initial hardware state, h_0, where the first instruction is able to enter the processor in the first cycle: the empty pipeline. For every program P and data d there is therefore only one element in $I(P, d)$. By Definition 10 and the reflexivity of $\sqsubseteq_{\text{time}}$ the hardware system is timing anomaly free. ◀

▶ **Lemma 18.** *The "branching by conditional execution" example in Figure 4 is not timing anomalous by Definition 10.*

Proof. The argumentation is the same as for Lemma 17. ◀

▶ **Lemma 19.** *LRU caches are not timing anomalous by our Definition 12.*

Proof. A stronger statement can actually be proven: that the empty cache is always the worst initial hardware state for LRU caches [7]. By Definition 10 this satisfies Definition 12. ◀

▶ **Lemma 20.** *FIFO caches are timing anomalous by our Definition 12.*

Proof. Consider the two traces in Figure 5, none of which are consistently slower than the other. By Lemma 11 an upper bound should exist. An upper bound would have to have misses for all accesses. By enumeration of all distinct initial caches, none of them have misses for all accesses, and thus no upper bound for the two traces exist. ◀

	d e		b a	
a	a d	x	b a	
c	c a	x	c b	x
a	c a		a c	x
b	b c	x	b a	x
c	b c		c b	x
a	a b	x	a c	x
b	a b		b a	x
c	c a	x	c b	x

Figure 5 Example of a timing anomaly for the FIFO cache on M_1, adopted from [2]. The first line is the initial state of the cache for the two traces. The first column is the access sequence, and the x'es indicate cache misses.

6 Conclusion and Future Work

In this work we have looked at previous definitions of timing anomalies, and identified flaws in them. Specifically in their applicability to various types of known timing anomalies, but also in what examples they deem to be timing anomalies. We have proposed a definition of timing anomalies in terms of the existence of a consistently worst initial hardware state, in the concrete model of the hardware and shown that it coincides with common knowledge about timing anomalies.

The next step is to provide an operational definition of timing anomalies that enables us to effectively check whether some hardware is timing anomalous, and if it is, identify a set of initial hardware states, such that they are consistently worse than all other hardware states. This would enable a WCET analysis by simulating the execution of these initial states. The framework we have proposed can also be used to take advantage of the efficient abstraction techniques to over-approximate WCET on timing anomalous platforms: given $\mathcal{H}$ which is timing anomalous, define $\mathcal{H}'$ that soundly approximates $\mathcal{H}$ and show that $\mathcal{H}'$ is timing anomaly free.

References

1 *ARM920T Technical Reference Manual*, 1 edition, 2001.

2 C. Berg. PLRU Cache Domino Effects. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

3 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 *Design and Diagnostics of Electronic Circuits and systems, 2006 IEEE*, pages 13–18, 2006.

4 G. Gebhard. Timing Anomalies Reloaded. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15, pages 1–10, 2010.

5 T. Lundqvist. *A WCET analysis method for pipelined microprocessors with cache memories.* PhD thesis, Chalmers University of Technology, 2002.

6 T. Lundqvist and P. Stenstrom. Timing Anomalies in Dynamically Scheduled Microprocessors. In *IEEE Real-Time Systems Symposium*, pages 12–21, 1999.

7 J. Reineke and D. Grund. Sensitivity of Cache Replacement Policies. Technical Report 36, March 2008. ISSN: 1860-9821, http://www.avacs.org/.

8 J. Reineke and R. Sen. Sound and Efficient WCET Analysis in the Presence of Timing Anomalies. In *9TH INTERNATIONAL WORKSHOP ON WORST-CASE EXECUTION TIME ANALYSIS*, page 101, 2009.

9 J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A Definition and Classification of Timing Anomalies. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

10 I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Quality Software, 2005.(QSIC 2005). Fifth International Conference on*, pages 295–303, 2005.

11 R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.

An Empirical Evaluation of the Influence of the Load-Store Unit on WCET Analysis*

Mohamed Abdel Maksoud and Jan Reineke

Saarland University
Saarbrücken, Germany
`{mohamed,reineke}@cs.uni-saarland.de`

Abstract

Due to the complexity of today's micro-architectures, the micro-architectural analysis usually constitutes the most time-consuming step in worst-case execution time (WCET) analysis.

In this paper, we investigate the influence of the design of the load-store unit (LSU) in the PowerPC 7448 on WCET analysis. To this end, we introduce a simplified variant of the existing design of the LSU by reducing its queue sizes. Using AbsInt's `aiT` WCET analysis toolchain we determine the resulting WCET bounds and analysis times.

For the modified version of the LSU with reduced queue sizes, analysis time is reduced by more than 50% on a set of benchmarks from the Mälardalen suite, while there is little change in the WCET bound.

1998 ACM Subject Classification B.8.2 Performance Analysis and Design Aids

Keywords and phrases empirical evaluation, architecture complexity effect, WCET analysis precision, WCET analysis performance, PowerPC 7448, Load-Store Unit

Digital Object Identifier 10.4230/OASIcs.WCET.2012.13

1 Introduction

The increasing complexity of today's micro-architectures makes the construction of sound and precise timing models an increasingly time-consuming and error-prone task. Furthermore, the resulting, complex timing models lead to a state-explosion problem in the micro-architectural (also known as low-level) analysis, drastically increasing overall WCET analysis times.

Most micro-architectural innovations, causing this increase in complexity, like speculation and out-of-order execution, are undertaken to improve average-case performance. In the WCET community it is often argued that many of these innovations do not improve, or even harm, a processor's worst-case timing behavior. There is, however, little hard evidence supporting such claims. This paper intends to contribute some hard evidence by performing an empirical evaluation using AbsInt's `aiT` WCET analysis toolchain.

The PowerPC 7448 is a high-performance microprocessor used in safety-critical real-time scenarios featuring caches, pipelining, speculation and out-of-order execution. To support speculation and out-of-order execution, the PowerPC 7448 includes a load-store unit (LSU), maintaining queues of memory instructions in different execution states. We investigate the influence of the lengths of these queues on both WCET bounds and WCET analysis times.

* The research leading to these results was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

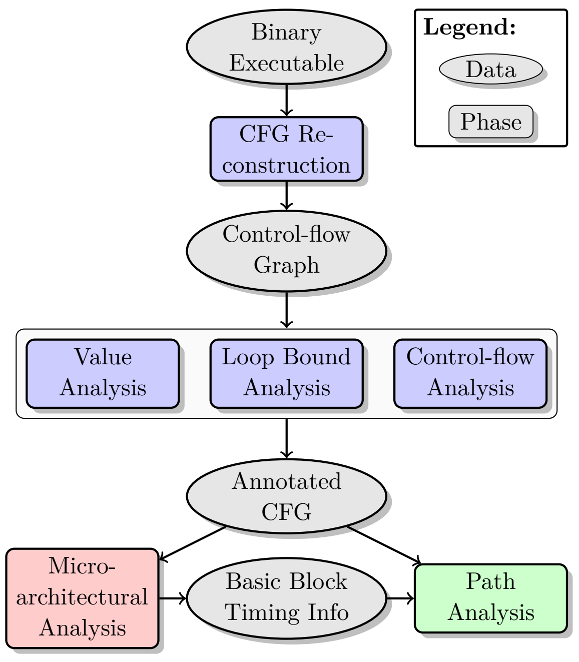


Figure 1 Main components of a timing-analysis framework and their interaction.

To this end, we introduce a simplified variant of the existing LSU, reducing its queue sizes to a minimum.

We compare the simplified design with the original design on various benchmarks from the Mälardalen benchmark suite. Surprisingly, we observe slightly decreased WCET bounds, and, as expected, strongly reduced analysis times.

2 Background

2.1 WCET Analysis Flow

Over roughly the last decade, a more or less standard architecture for timing-analysis tools has emerged. Figure 1 gives a general view of this architecture. The following list presents the individual phases and describes their objectives.

1. *Control-flow reconstruction* [21] takes a binary executable to be analyzed, reconstructs the program's control flow and transforms the program into a suitable intermediate representation. Problems encountered are dynamically computed control-flow successors, e.g. those stemming from switch statements, function pointers, etc.
2. *Value analysis* [4] computes an over-approximation of the set of possible values in registers and memory locations by an interval analysis and/or congruence analysis. The computed information is used for a precise data-cache analysis and in the subsequent control-flow analysis. Value analysis is the only one to use an abstraction of the processor's arithmetic. A subsequent pipeline analysis can therefore work with a simplified pipeline where the arithmetic units are removed. There, one is not interested in what is computed, but only in how long it will take.
3. *Loop bound analysis* [8, 14] identifies loops in the program and tries to determine bounds on the number of loop iterations; information indispensable to bound the execution time.

Problems are the analysis of arithmetic on loop counters and loop exit conditions, as well as dependencies in nested loops.

4. *Control-flow analysis* [8, 20] narrows down the set of possible paths through the program by eliminating infeasible paths or by determining correlations between the number of executions of different blocks using the results of value analysis. These constraints will tighten the obtained timing bounds.
5. *Micro-architectural analysis* [7, 23, 10, 5] determines bounds on the execution time of basic blocks by performing an abstract interpretation of the program, combining analyses of the processor's pipeline, caches, and speculation. Static cache analyses determine safe approximations to the contents of caches at each program point. Pipeline analysis analyzes how instructions pass through the pipeline accounting for occupancy of shared resources like queues, functional units, etc.
6. *Path Analysis* [17, 22] finally determines bounds on the execution times for the whole program by implicit path enumeration using an integer linear program (ILP). Bounds of the execution times of basic blocks are combined to compute longest paths through the program. The control flow is modeled by Kirchhoff's law. Loop bounds and infeasible paths are modeled by additional constraints. The target function weights each basic block with its time bound. A solution of the ILP maximizes the sum of those weights and corresponds to an upper bound on the execution times. In the following, we refer to the kind of path analysis described above as *traditional* ILP-based analysis.

The commercially available tool `aiT` by AbsInt, cf. `http://www.absint.de/wcet.htm`, implements this architecture. It is used in the aeronautics and automotive industries and has been successfully used to determine precise bounds on execution times of real-time programs [10, 9, 24, 15].

The ILP-based path analysis in `aiT` comes in two variants depending on how micro-architectural state graphs are constructed [2]:

1. *Traditional ILP-based analysis*, where an ILP is solved to find the worst-case path through the program, given worst-case timings of all basic blocks (possibly in various contexts). This approach may be imprecise, because the worst-case timings of some basic blocks may not occur simultaneously on a single architectural path through the program.
2. *Prediction-file-based ILP analysis*, where a global state graph consisting of micro-architectural states is constructed, and an ILP is solved to find the worst-case path. This results in a more precise WCET bound since architecturally-infeasible paths are excluded. However, it comes at the cost of a much larger ILP to be solved.

To illustrate the difference between the two path-analysis methods, consider the example analysis shown in Figure 2. An ILP-based path analysis computes a global WCET bound solely based

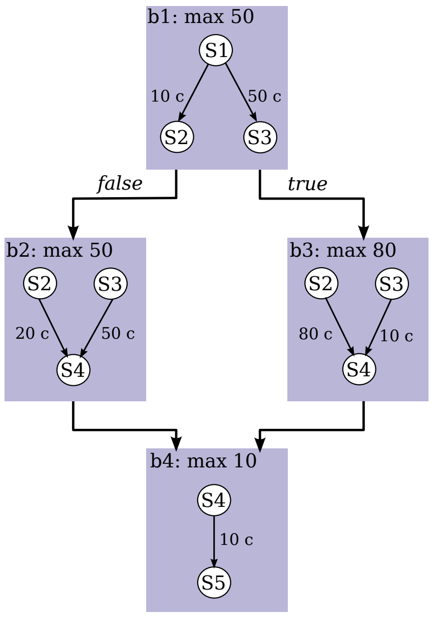


Figure 2 An example illustrating the differences between traditional ILP-based path analysis and prediction-file-based ILP path analysis.

on the maximum number of execution cycles for each basic block. The WCET is therefore 140 cycles in this case, and the worst-case exeuction path is `b1`→`b3`→`b4`. However, this result implies an architecturally-infeasible execution trace: `s1`→`s3`|||`s2`→`s4`→`s5`, where trace discontinuity is marked by |||.

On the other hand, the global state graph constructed in a prediction-file-based ILP path analysis excludes such paths and produces a WCET bound of 110 cycles, with the corresponding worst-case execution path: `b1`→`b2`→`b4`, and trace: `s1`→`s3`→`s4`→`s5`.

2.2 Motorola PowerPC 7448

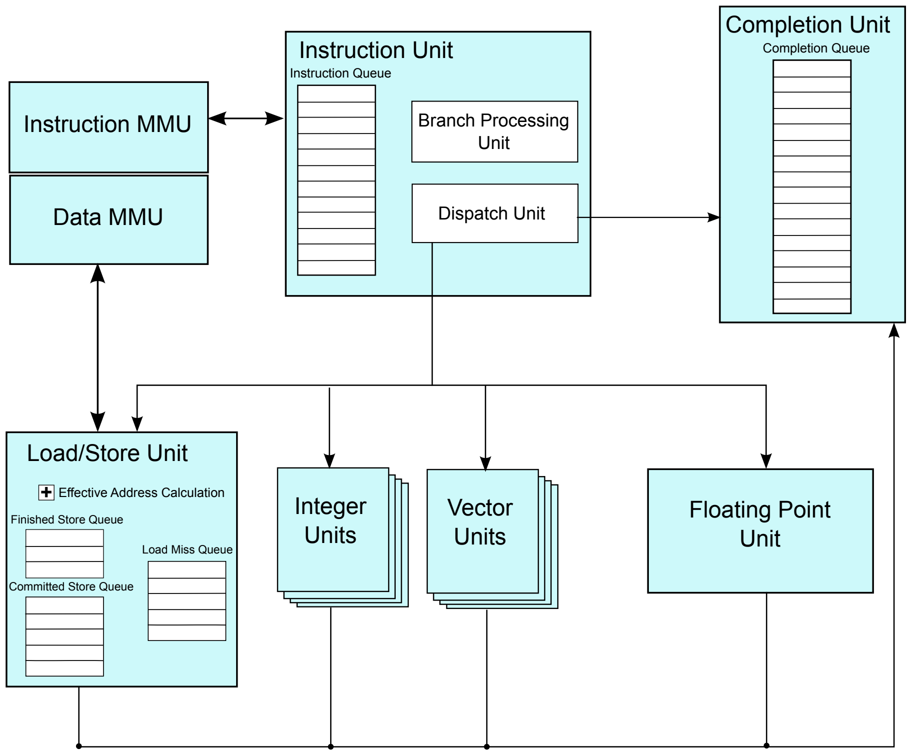


Figure 3 PowerPC 7448 Microprocessor Block Diagram.

The PowerPC 7448 is a reduced instruction set computer (RISC) superscalar processor that implements the 32-bit portion of the PowerPC architecture and the SIMD instruction set AltiVec architectural extension. It features a two-level memory hierarchy with separate L1 data and instruction caches (Harvard architecture), a unified L2 cache, four independent integer and four independent vector units for superscalar execution. It also features static and dynamic branch prediction, and a sophisticated load-store unit with long buffers.

"The PowerPC 7448 provides virtual memory support for up to 4 PB (2^{52}) of virtual memory and real memory support for up to 64 GB (2^{36}) of physical memory. It can dispatch and complete three instructions simultaneously" [11]. It consists of the following execution units, depicted in Figure 3:

- Instruction Unit (IU): the IU provides centralized control of instruction flow to the execution units. It contains an instruction queue (IQ), a dispatch unit (DU), and a branch processing unit (BPU). The IQ has 12 entries and loads up to 4 instructions from the instruction cache in one cycle. The DU checks register dependencies and the availability of a position in the *completion queue* (described below), and issues or inhibits subsequent instruction dispatching accordingly. The BPU receives branch instructions from the IQ and executes them early in the pipeline. If a branch has a dependency that has not yet been resolved, the branch path is predicted using either architecture-defined static branch prediction or PowerPC 7448 -specific dynamic branch prediction.

- Completion Unit (CU): The CU retires an instruction from the 16-entry completion queue (CQ) when all instructions ahead of it have been completed. The CU operates closely with the IU to ensure that the instructions are retired in program order.
- Integer, Vector, and Floating-Point Units: the PowerPC 7448 provides nine execution units to support the execution of integer, fixed point, and AltiVec instructions.
- Cache/Memory Subsystem: The PowerPC 7448 microprocessor contains two separate 32-Kbyte, eight-way set-associative level 1 (L1) instruction and data caches (Harvard architecture). The caches implement a pseudo least-recently-used (PLRU) replacement policy. In addition, the PowerPC 7448 features an integrated 1 MB level 2 (L2) cache.
- Load-Store Unit (LSU): The LSU executes all load and store instructions and provides the data transfer interface between registers and the cache/memory subsystem. The LSU also calculates effective address and aligns data. This unit is described in detail in the following section.

Load-Store Unit

The LSU provides all the logic required to calculate effective addresses, handles data alignment to and from the data cache, and provides sequencing for load-store string and load-store multiple operations [11]. The LSU contains a 5-entry load miss queue (LMQ) which maintains the load instructions that missed the L1 cache until they can be serviced. This allows the LSU to process subsequent loads. Unlike loads, stores cannot be executed speculatively: a store instruction is held in the 3-entry finished store queue (FSQ) until the completion unit signals that the store is committed, only then it moves to the 5-entry committed store queue (CSQ). In order to reduce the latency of loads dependent on stores, the LSU implements *data forwarding* from any entry in the CSQ before the data is actually written to the cache. When a load instruction misses, its address is compared to all entries in the CSQ. On a hit, the data is forwarded from the newest matching entry. If the address is also found in the FSQ, however, the LSU stalls since the newest data at this address could be updated should the store instruction in the FSQ be committed.

Analysis Model of the Load-Store Unit

During static analysis, crucial information on program execution such as register contents, cache contents and *bus clock offset* cannot be decided exactly. The bus clock offset is defined as the number of processor clock cycles until the next rising edge of the bus clock. When the analysis flow depends on such information, the analysis has to proceed in all possible paths to ensure a sound WCET bound in the presence of timing anomalies [18]. When the analysis is to proceed in more than one path, the analysis state has to be *split*, with the consequence of increasing the size of the state space during analysis and hence reducing analysis efficiency. The analysis model of the load-store unit reflects its structure in the concrete processor architecture, while accounting for non-determinism. In the load-store unit, the addresses of different memory accesses are represented in terms of intervals, rather than exact numbers. As we shall see in Section 4, the load-store unit is a significant source of splits in most cases, and this is attributed to the long queues in this unit. As described in the previous section, data forwarding involves a number of comparisons for each missed load instruction. This number is proportional to the sizes of the load miss queue, committed store queue, and finished store queue. These comparisons are performed on imprecise addresses, resulting in potential splits when it cannot be decided whether addresses do alias or not. Moreover, having long queues in the LSU indicates more pending memory accesses in the core waiting to be served. Serving more accesses increases the possibility of querying non-exact bus clock offset, hence representing another source of splits. These observations motivate our experimental setup described in the following section.

3 Experimental Setup

To the end of reducing the number of splits, and thus improving analysis time, we modified the PowerPC 7448 by cutting the queue sizes in the load-store unit. The LMQ, CSQ, and FSQ sizes were reduced to 1, 2, and 1, respectively[1]. The benchmarks were selected from the Mälardalen benchmark suite [13]. These are the benchmarks for which the WCET analysis terminated successfully for both architectures. The analyzed programs are briefly described in Table 1. The benchmarks marked with * were not found in the official documentation although they are included in the test-suite distribution.

Table 1 List of benchmarks analyzed in the experiment.

Benchmark	Description	LOC
bs	Binary search for the array of 15 integer elements.	114
cnt	Counts non-negative numbers in a matrix.	267
expint	Series expansion for computing an exponential integral function.	157
fac	Computes the sum of factorials of a set of integers.	28
fibcall	Simple iterative Fibonacci calculation, used to calculate fib(30).	72
janne_complex	Nested loop program.	64
lcdnum	Read ten values, output half to LCD.	64
loop3*	Several loop patterns.	240
minmax*	Simple program with infeasible paths.	58
qurt	Root computation of quadratic equations.	166
sqrt	Square root function implemented by Taylor series.	77

The `aiT` analyzer was configured to use traditional ILP-based path analysis (with the CLP solver [1]) on all benchmarks and prediction-file based ILP path analysis only on some of them. Although the latter produces more precise WCET bounds, it is more computationally demanding as will be seen in the following section, and we were not able to finish the analysis of all of the benchmarks in time for this submission.

The experiment was performed on a 64-bit AMD Opteron machine with 16 processor cores at 2500 MHz and 64 GB of RAM. As the WCET analysis is not parallelized, we ran multiple analyses concurrently on this machine. As performance metrics, we use the micro-architectural-analysis time and the path-analysis time. On the analyzed benchmarks, these two metrics constitute on the average about 80% and 75% of the whole analysis time for the standard and reduced architectures, respectively.

[1] This is the strongest simplification we could apply without having to make significant changes to the micro-architectural analysis.

4 Experimental Results and Analysis

The analysis results of selected benchmarks using prediction-file-based ILP path analysis are shown in Table 2. We compute both the average of the relative changes (<average>) and the relative change of the sum of the respective values (<weighted average>).

Looking first at the analysis performance metrics, we see that the state space in the reduced architecture is significantly smaller than that of the standard architecture. This is manifested in the consistently lower number of splits in the micro-architectural analysis and path analysis time, cf. the `janne_complex` benchmark. For less memory-demanding benchmarks, such as `fac` and `fibcall`, we do not see significant improvement in the analysis performance.

Comparing the WCET bounds in both architectures yields a surprise: in half of the cases, the reduced architecture achieves a WCET bound that is *lower* than that of the standard architecture. A closer look at one of the benchmarks featuring this anomaly, `minmax`, reveals the following:

- There are no ambiguous memory accesses in this simple benchmark (i.e. all addresses are exact), the effect of data-forwarding on the analysis precision is therefore ruled out.
- The benchmark starts with several store instructions followed by a branch instruction.
- The standard architecture with its long store queues accomodates more pending stores and execute further instructions, including the branch. This results in more pending memory requests, both data and instruction accesses, and hence it is more likely to query non-exact bus clock offset.
- On the other hand, the reduced architecture accomodates fewer pending stores, hence it stalls on encountering more store instructions. This stalling is beneficial in that it leads to fewer pending memory accesses and hence reduces the loss of precision caused by the non-deterministic bus clock offset.

Using the less precise, yet significantly more efficient traditional ILP-based path analysis, more benchmarks were analyzed. The analysis results and performance metrics are shown in Table 3.

We observe an increased average speed-up of the micro-architectural analysis for the reduced architecture compared with the results for the prediction-file-based analysis. The increased average speed-up is attributed to analyzing more memory intensive benchmarks such as `bs` and `cnt`. The micro-architectural analysis time varies slightly from that in Table 2 for some benchmarks, likely due to interference on shared resources between multiple analyses running concurrently on the machine.

The path-analysis contribution to the total analysis performance is insignificant, compared to that of the micro-architectural analysis. In the traditional ILP-based approach, path-analysis time is expected to be independent of the complexity of the underlying micro-architecture. Unsurprisingly, we observe little differences between the standard and the reduced architecture.

The "WCET anomaly", i.e. lower WCET bounds for the reduced architecture, is more pronounced using this path-analysis method. This is not surprising since a larger number of paths with different timings through basic blocks, as is the case for the standard architecture, makes it more likely for the path analysis to compute an architecturally infeasible worst-case execution path. This adds up to the precision loss observed in the standard architecture.

Table 2 WCET bounds and performance metrics using prediction-file-based ILP path analysis.

Benchmark	WCET bound *in cycles*			μarch. analysis splits			μarch. analysis time *in seconds*			Path-analysis time *in seconds*			Total analysis time *in seconds*	
	std.	red.	Δ %	std.	red.	Δ %	std.	red.	Δ %	std.	red.	Δ %	std.	red.
fac	3,321	3,343	0.7	1,610	1,586	-1.5	0.6	0.6	-0.8	0.1	0.1	-7.1	1.1	1.1
fibcall	3,346	3,325	-0.6	844	789	-6.5	0.5	0.5	-0.2	0.1	0.1	1.9	0.9	0.9
janne_complex	20,005	19,846	-0.8	107,222	4,599	-95.7	21.4	1.1	-94.8	3445.4	0.9	-100.0	3470.1	2.5
lcdnum	1,969	1,996	1.4	20,533	8,506	-58.6	3.2	1.4	-55.5	1.5	0.6	-62.0	5.1	2.4
loop3	39,329	41,199	4.8	8,380	5,942	-29.1	4.5	3.8	-15.9	0.7	0.4	-38.6	8.2	7.2
minmax	1,629	1,500	-7.9	2,925	1,106	-62.2	1.7	0.8	-52.7	1.0	0.1	-91.0	3.1	1.2
qurt	17,817	17,953	0.8	98,859	25,459	-74.2	62.7	14.0	-77.7	54.5	10.8	-80.2	120.8	26.4
sqrt	5,096	4,976	-2.4	26,415	7,768	-70.6	15.6	3.4	-77.9	13.0	2.3	-81.9	29.8	6.5
<average>			-0.5			-46.9			-49.8			-57.3		
<weighted avg.>			1.8			-76.7			-79.1			-99.6		

$$\mu\text{arch. analysis splits} := \text{the total number of splits in the micro-architectural analysis,}$$

$$\Delta\%(b, \text{<measure>}) := \frac{\text{<measure>}_{\text{red.}}(b) - \text{<measure>}_{\text{std.}}(b)}{\text{<measure>}_{\text{std.}}(b)} \times 100,$$

$$\text{<average>}(\text{<measure>}) := \frac{\sum\limits_{b \in \text{benchmarks}} \Delta\%(b, \text{<measure>})}{|\text{benchmarks}|}\text{, and}$$

$$\text{<weighted avg.>}(\text{<measure>}) := \frac{\sum\limits_{b \in \text{benchmarks}} \text{<measure>}_{\text{red.}}(b) - \sum\limits_{b \in \text{benchmarks}} \text{<measure>}_{\text{std.}}(b)}{\sum\limits_{b \in \text{benchmarks}} \text{<measure>}_{\text{std.}}(b)} \times 100.$$

5 Related Work

While there is an abundance of work proposing more predictable or analyzable micro-architectures, there is not a lot of work that empirically studies the impact of simplifications of micro-architectures on WCET analysis time. Exceptions include the work of Grund et al. [12] and Burguière and Rochange [3].

Grund et al. [12] investigate several modifications of the branch target instruction cache of the PowerPC 56x. They observe that using LRU in place of FIFO replacement reduces analysis time drastically, as more memory accesses can be classified as hits or misses, thereby reducing the number of splits.

Burguière and Rochange [3] investigate the modeling complexity of various dynamic branch prediction schemes. Here, the modeling complexity is measured by the number of constraints, the number of variables, and the sizes of constraints in an ILP formulation of the behavior of the respective branch prediction schemes. This analysis is based on the assumption that the modeling complexity is strongly-correlated with the resulting analysis complexity. However, the actual analysis times are not analyzed.

Heckmann et al. [15] focus on the difficulty in modeling various architectural components, including caches and pipelines, and their influence on the precision of the resulting analyses. From their experience in modeling various processors they derive several recommendations regarding the design of processors for real-time systems. Later, Wilhelm et al. [26] describe properties of memory hierarchies, pipelines, and buses, which make timing analysis more complex and/or reduce its precision. Neither Heckmann et al. nor Wilhelm et al. provide an empirical evaluation of their recommendations.

Approaches aiming at improving predictability or analyzability include the EU projects Predator, Merasa [25], the PRET project [6], and the Java-Optimized Processor JOP [19]. These projects present entirely new processor designs. This makes it difficult to evaluate the impact of individual design choices on WCET analysis times. In the context of the JOP project, Huber et al. [16] analyze the influence of different object cache configurations on worst-case execution time estimates, varying several cache parameters and the background memory. They do not, however, analyze the impact of the design choices on analysis times.

6 Conclusions and Future Work

In this paper, we have investigated the influence of the design of the load-store unit on WCET analysis, in terms of analysis times and WCET bounds. Reducing the complexity of the LSU results in significantly shorter analysis times, and, surprisingly, sometimes even in slightly lower WCET bounds. We plan to investigate the influence of further components to get a more complete view of how strongly various components and their configurations influence WCET analysis.

Regarding the “WCET anomaly” found in some benchmarks analyzed in this paper, we are uncertain whether it is a product of analysis imprecision, or whether it corresponds to actual behaviors of the respective architectures. It will be future work to shed more light on this question.

Table 3 WCET bounds and performance metrics using traditional ILP-based path analysis.

Benchmark	WCET bound *in cycles*			μarch. analysis splits			μarch. analysis time *in seconds*			Path-analysis time *in seconds*			Total analysis time *in seconds*	
	std.	red.	Δ %	std.	red.	Δ %	std.	red.	Δ %	std.	red.	Δ %	std.	red.
bs	11,082	9,807	-11.5	166,466	18,458	-88.9	201.6	15.8	-92.1	0.0	0.0	7.7	201.8	16.1
cnt	44,285	38,399	-13.3	20,489,562	830,784	-95.9	16842.5	489.4	-97.1	0.1	0.1	-12.3	16843.8	490.7
expint	13,610	13,536	-0.5	4,127	2,873	-30.4	2.5	2.0	-20.6	0.1	0.1	3.7	3.6	3.1
fac	4,173	4,015	-3.8	1,610	1,586	-1.5	0.6	0.6	-1.7	0.0	0.0	0.0	1.0	1.0
fibcall	3,685	3,530	-4.2	844	789	-6.5	0.5	0.5	5.1	0.0	0.0	0.0	0.8	0.9
janne_complex	28,172	21,034	-25.3	107,222	4,599	-95.7	19.8	1.1	-94.7	0.0	0.0	0.0	20.1	1.3
lcdnum	2,538	2,506	-1.3	20,533	8,506	-58.6	3.0	1.3	-55.3	0.0	0.0	-10.5	3.3	1.7
loop3	53,986	53,879	-0.2	8,380	5,942	-29.1	4.2	3.6	-15.5	0.1	0.1	0.0	7.2	6.5
minmax	1,987	1,898	-4.5	2,925	1,106	-62.2	1.7	0.8	-51.4	0.0	0.0	8.3	1.9	1.1
qurt	26,363	21,742	-17.5	98,859	25,459	-74.2	60.5	13.5	-77.6	0.1	0.1	27.5	61.5	14.5
sqrt	7,120	5,576	-21.7	26,415	7,768	-70.6	14.6	3.2	-77.9	0.0	0.0	0.0	15.1	3.7
<average>			-9.4			-52.6			-55.8			2.2		
<weighted avg.>			-10.7			-96.9			-95.7			2.2		

References

1 COIN-OR Linear Programming: `http://www.coin-or.org/Clp`.

2 AbsInt Angewandte Informatik GmbH. *AbsInt Advanced Analyzer for PowerPC MPC7448 (Simple Memory Model): User Documentation.*

3 Claire Burguière and Christine Rochange. On the complexity of modeling dynamic branch predictors when computing worst-case execution time. In *Proceedings of the ERCIM/DECOS Workshop On Dependable Embedded Systems*, August 2007.

4 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, New York, NY, USA, 1977. ACM Press.

5 Christoph Cullmann. Cache persistence analysis: a novel approach theory and practice. In Jan Vitek and Bjorn De Sutter, editors, *LCTES*, pages 121–130. ACM, 2011.

6 Stephen A. Edwards and Edward A. Lee. The case for the precision timed (PRET) machine. In *DAC*, pages 264–265. IEEE, 2007.

7 Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis.* PhD thesis, Dept. of Information Technology, Uppsala University, 2002.

8 Andreas Ermedahl and Jan Gustafsson. Deriving annotations for tight calculation of execution time. In *Euro-Par*, pages 1298–1307, 1997.

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 *International Conference on Embedded Software*, 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 Sys.*, 17(2-3):131–181, 1999.

11 Freescale Semiconductor. *MPC7450 RISC Microprocessor Family Reference Manual.*

12 Daniel Grund, Jan Reineke, and Gernot Gebhard. Branch target buffers: WCET analysis framework and timing predictability. *Journal of Systems Architecture*, 57(6):625–637, 2011.

13 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. pages 137–147, Brussels, Belgium, July 2010. OCG.

14 C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Sys.*, pages 129–156, 2000.

15 Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003.

16 Benedikt Huber, Wolfgang Puffitsch, and Martin Schoeberl. Worst-case execution time analysis-driven object cache design. *Concurrency and Computation: Practice and Experience*, 24(8):753–771, 2012.

17 Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 456–461, 1995.

18 Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *Proceedings of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2006.

19 Martin Schoeberl. A java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54(1-2):265 – 286, 2008.

20 Ingmar Stein and Florian Martin. Analysis of path exclusion at the machine code level. In *Proceedings of the 7th Intl. Workshop on Worst-Case Execution-Time Analysis*, 2007.

21 Henrik Theiling. *Control-Flow Graphs For Real-Time Systems Analysis*. PhD thesis, Saarland University, Saarbrücken, Germany, 2002.

22 Henrik Theiling. ILP-based interprocedural path analysis. In *International Conference on Embedded Software*, volume 2491 of *LNCS*, pages 349–363. Springer, 2002.

23 Stephan Thesing. *Safe and Precise WCET Determinations by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, Saarbrücken, Germany, 2004.

24 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 systems. In *Proceedings of the 2003 Intl. Conference on Dependable Systems and Networks*, pages 625–632. IEEE Computer Society, 2003.

25 Theo Ungerer, Francisco J. Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quiñones, Mike Gerdes, Marco Paolieri, Julian Wolf, Hugues Cassé, Sascha Uhrig, Irakli Guliashvili, Michael Houston, Florian Kluge, Stefan Metzlaff, and Jörg Mische. Merasa: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5):66–75, 2010.

26 Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(7):966–978, 2009.

Computing *Same Block* Relations for Relational Cache Analysis

Simon Wegener

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

Abstract

In contrast to the classical cache analysis of Ferdinand, the relational cache analysis does not rely on precise address information. Instead, it uses *same block* relations between memory accesses to predict cache hits. The relational data cache analysis can thus also predict cache hits if fully unrolling a loop is not feasible during analysis, for example due to high memory consumption or long computation time. This paper proposes a static analysis based on abstract interpretation which is able to compute *same block* relations for relational cache analysis.

1998 ACM Subject Classification C.4 Performance of Systems, D.2.4 Software/Program Verification

Keywords and phrases Cache Analysis, WCET Analysis, Real-time Systems, Static Program Analysis, Abstract Interpretation

Digital Object Identifier 10.4230/OASIcs.WCET.2012.25

1 Introduction

In his doctoral thesis [4], Ferdinand proposed an analysis based on abstract interpretation to predict the contents of set-associative caches with LRU replacement policy.

The analysis is split in two parts. One part, the so-called *must analysis* is used to predict definite cache hits ("always hit") by computing an under-approximation of the possible cache contents at any program point. The other part, called *may analysis* is used to predict definite cache misses ("always miss") by computing an over-approximation of the possible cache contents. For those memory accesses where neither the must analysis nor the may analysis are able to predict a definite result, "not classified" is returned.

One requirement of the cache analysis described above is the knowledge of precise address information for the targets of memory accesses. This information is typically available if instruction caches are analyzed since the control flow and thus the addresses of instructions have been computed beforehand. For data caches or unified instruction / data caches, this requirement cannot always be fulfilled.

Consider for example an array access depending on a loop counter as in listing 1. As a data-flow analysis computes invariants which hold for each and every execution of a program, it can only compute a set or an interval of possible addresses. If for example a is the array's base address and w the width of an array element, then the address interval would be

Listing 1 Array summation in C.

```
for (i = 0; i < 128; i++)
    sum += arr[i];
```

$[a, a + 127 \cdot w]$. In this case, all abstract cache sets which possibly would handle one of these addresses must be updated. For the may sets, this means that all the addresses included in the interval must be added. For the must sets, this means that none of these addresses are added, but all entries age by one. Ferdinand's cache analysis thus reacts very sensitively to imprecise address information.

One way to overcome this obstacle is to fully unroll the loop. Then, the different memory accesses can be distinguished by the analysis. However, fully unrolling may not be feasible, for example if the loop iterates several thousand or even million times. No information would reside in the abstract data cache sets inside such a loop. The memory accesses could then not be classified as either cache hits or misses.

Recent research [15, 9] studied how the dependency of the cache analysis on precise address information can be reduced. The result is the so-called relational cache analysis. There, symbolic names are used to identify memory accesses. Cache hits are predicted with the help of *same block* relations. These relations describe sufficient conditions whether two memory accesses target the same cache line. Ferdinand's cache analysis can be seen as an instance of the relational cache analysis framework, with cache address equality as *same block* relation.

The goal of this paper is to define additional analyses which can be used to compute *same block* relations. These can then be used in the relational cache analysis framework described in [9] to predict cache hits.

2 Cache Configuration

In the following (and if not stated otherwise), the cache configuration is assumed to be a write-through, write-allocate 2-way set associative LRU cache with a cache line size of 32 bytes. The whole memory is assumed to be cached. No cache locking takes place. Upon a miss, a whole line is loaded into the cache.

3 Predicting Cache Hits: Globally Precise Address Information vs. Locally Precise Address Information

A shortcoming of Ferdinand's cache analysis is its dependency on globally precise address information. This dependency on the exact position of a memory access in the whole address space is quite natural if we look at the definition of a cache hit in the concrete domain:

$$\mathrm{hit}(a) \overset{\mathrm{def}}{=} \exists i \in \{1, 2\} : \mathrm{cacheset}(a)[i].\mathrm{valid} \wedge \mathrm{cacheset}(a)[i].\mathrm{tag} = \mathrm{tag}(a)$$

A memory access is a cache hit if and only if there is an entry in the cache set which is valid and which stores the tag of the access address a (see also Figure 1). To compute the tag and the cache address, the exact memory address is needed. If this information is not available, we cannot evaluate the formula above.

Now recall the array summation example: There, some cache hits will occur after a cache miss, because a cache miss loads whole cache lines into the cache. The subsequent memory accesses will then lead to cache hits as long as they target the same cache line. How can we use this fact to predict cache hits if we do not have the exact address information?

Fortunately, there is a possibility to specify whether two memory accesses target the same cache line which does not depend on the exact addresses. Let a_{prev} be the address of the memory access previous to the ongoing access (with address a). Assume that the distance $x = a - a_{prev}$ between the two access addresses is known as well as the position

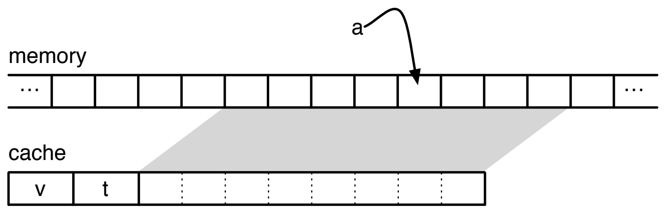


Figure 1 A cache consisting of only one cache line. The tag t maps the cache line to a specific interval of memory locations. The memory access to a is a hit if tag(a) $= t$ and the valid bit v is set. The value of t depends on the memory access before the one to a.

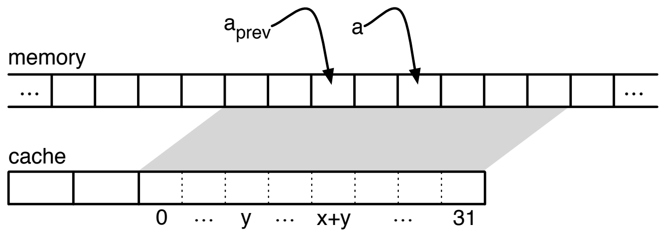


Figure 2 A cache consisting of only one cache line. a is the address of the ongoing memory access, a_{prev} is the address of the previous access, $x = a - a_{prev}$ is the distance between the two accesses and $y = a_{prev}$ mod 32 is the position of the previous access inside the cache line. The memory access to a is a hit if the cache line offset of a_{prev} plus the distance between a and a_{prev} is smaller than the cache line size.

of the previous access inside the cache line $y = a_{prev}$ mod 32. Then, the ongoing memory access is a cache hit if $0 \leq x + y < 32$ (see Figure 2).

The values of x and y can be computed without knowing the exact values of a and a_{prev} (see section 4). Locally precise address information is thus enough to predict cache hits.

4 Computing *Same Block* Relations

In this section, the findings from the previous section are formalized and used to build an analysis which is able to compute *same block* relations for the relational cache analysis.

4.1 Alignment Information

One information used to compute the *same block* relation is the relative position of a memory access inside a cache line. To compute the relative position, we use a static analysis [7] using arithmetical congruences as abstract domain. Values are identified by the congruence class to which they belong. The 32-bit address a for example is described by the pair $\langle a, 2^{32} \rangle$ because $a \equiv a \pmod{2^{32}}$ holds. The two addresses a and $a + 2$ are described either by the pair $\langle 0, 2 \rangle$ or by the pair $\langle 1, 2 \rangle$, depending whether a is even or odd.

Recall the array access example. If an element of the array has a size of four bytes, the analysis deduces that the address of each memory access inside the loop goes to the same congruence class (modulo 4). Formally speaking, the analysis computes the pair $\langle y, 4 \rangle$, that is, the equation $\exists y \in \{0, \ldots, 3\} : \forall i \in \{0, \ldots, 127\} : a_i \equiv y \pmod 4$. If the array is properly 32-bit aligned, the analysis can even deduce that $y = 0$.

However, this information is not precise enough for our needs, because not the whole cache line is covered by congruence classes, i.e. the modulus is to small. To improve the precision, some of the addresses must be kept apart (see section 4.2).

4.2 Loop Peeling and Loop Unrolling

The precision problems arise from the the fact that the computed invariants must hold for each and every loop iteration. Thus only the least common divisor survives as modulus.

Skillful application of loop peeling (listing 2) and loop unrolling (listing 3) can be used to keep some of the address apart. 4-fold unrolling of the loop in the running example can be used to improve the alignment information such that the four pairs $\langle 0, 16\rangle$, $\langle 4, 16\rangle$, $\langle 8, 16\rangle$ and $\langle 12, 16\rangle$ are computed for the four array accesses (assuming a properly aligned array). Loop peeling is used to control which alignment information pair is computed for what unrolled array access (see section 6 for more details).

The loop unrolling and loop peeling can be done either directly by using the corresponding loop transformations or virtually by using a specially tailored context mapping (e.g. VIVUM [15]).

Listing 2 Loop peeling.

```
sum += arr[0];
sum += arr[1];
sum += arr[2];
for (i = 3; i < 128; i++) {
    sum += arr[i];
}
```

Listing 3 Loop unrolling.

```
for (i = 0; i < 128; i += 4) {
    sum += arr[i];
    sum += arr[i + 1];
    sum += arr[i + 2];
    sum += arr[i + 3];
}
```

4.3 Distance Relations

Often, a variable is not invariant inside a loop, but changes its value. Thus, static analyses can only compute an abstraction of the possible values, for example an interval. This abstraction might be very imprecise. One possibility to improve the results is to check how the value evolves over time.

In the running example, the address of the memory access is increased by the size of one element in each iteration. More formally, the equation $\forall i \in \{1, \ldots, 127\} : a_i - a_{i-1} = x$ holds, where x is the size of one array element.

Such linear relations can be expressed with difference bound matrices (or short DBMs). DBMs have been introduced by Dill [3] to express clock zones for the model-checking of timed-automata. Miné [13] used them to build an abstract domain which efficiently handles a restricted form of linear relations, namely those of the form $x_i - x_j \leq c$.

In a DBM, the row / column positions identifies which variables are used in the relation and the value inside the DBM is the constraining factor (i.e. the c above). The first row / column is used for the special variable x_0 which is used to express the constant zero. Infinity is used as value if the difference cannot be bounded.

If again 4-fold unrolling is applied to the loop in the example, one gets the four DBM variables $x_1, \ldots, x_4$. The analysis computes the DBM in Figure 3, assuming each array element has a size of 4 bytes.

The DBM analysis is applied to the whole program. However, not every register is covered in the DBMs but only those that are used to compute the addresses of memory accesses. This set is identified for each loop independently.

$$\begin{pmatrix} 0 & \infty & \infty & \infty & \infty \\ \infty & 0 & 4 & 8 & 12 \\ \infty & -4 & 0 & 4 & 8 \\ \infty & -8 & -4 & 0 & 4 \\ \infty & -12 & -8 & -4 & 0 \end{pmatrix}$$

Figure 3 DBM for the array accesses in the loop in listing 3. The address of each access is exactly four bytes greater than the address of the previous access.

4.4 Hit Classification

We have now all information available to compute the *same block* relation $\overset{\text{sb}}{\sim}$, which in turn is used to predict cache hits.

Three data structures are needed for the classification: the abstract cache set $\hat{\mathcal{S}}$, the map $\hat{\mathcal{V}}$ containing the relative positions of memory accesses inside a cache line and the map $\hat{\mathcal{R}}$ containing the distances between memory accesses.

The abstract cache set $\hat{\mathcal{S}}$ is build as an array containing sets of reference memory accesses. A reference memory access is an access which forces a particular line to be loaded into the cache.[1] To identify such a reference memory access, symbolic names are used, e.g. the instruction which induced the access.

The map $\hat{\mathcal{V}}$ contains for each memory access the alignment information pairs from section 4.1.

The map $\hat{\mathcal{R}}$ contains for each memory access a set of tuples, where each tuple consists of another memory access and the minimal and maximal distance of this memory access to the one used as index. If the DBM analysis could not derive such information, the set is empty.

Additionally, the width of a memory access is given as w_i, as it depends only on the instruction inducing the memory access.

A reference memory access i_{ref} and an ongoing memory access i are *same block*-related if one can show that the addresses of the ongoing and the reference access target the same cache line. For this, adding the access width w_i and the maximum distance x_{max} to the relative position of the reference access inside the cache line must not exceed the length of the cache line. The same check has to be done for the lower cache line boundary, too. Formally speaking:

$$\begin{aligned}\overset{\text{sb}}{\sim} \overset{\text{def}}{=} \{\langle i, i_{ref}\rangle \mid \exists x_{min}, x_{max}, y, z : \langle i_{ref}, x_{min}, x_{max}\rangle \in \hat{\mathcal{R}}[i] \wedge \langle y, z\rangle = \hat{\mathcal{V}}[i_{ref}] \\ \wedge (y \bmod 32) + x_{max} + w_{instr} \leq \min(z, 32) \wedge (y \bmod 32) + x_{min} \geq 0\}\end{aligned}$$

The information $\langle y, z\rangle$ means that the address a_{ref} of the reference access satisfies $a_{ref} \equiv y \pmod{z}$. The remainder is then taken modulo the cache line width as we are only interested in the relative position inside the cache line. The modulus z is taken as basis of comparison if it is smaller than the cache line width.

[1] In the assumed cache setting, each memory access is also a reference memory access, as both reads and writes load a line when a cache miss happens. When a cache hit happens, the line is already loaded, thus the access is again a reference memory access.

Listing 4 Array summation in PowerPC assembler.

```
.INIT:
    lis r9, 0x18880000@h          // load base address of arr ...
    addi r9, r9, -736             // ... into r9
    lis r8, 0x18880000@h          // load first address after arr ...
    addi r8, r8, -224             // ... into r8
.LOOP:
    lwzx r0, +0(r9)               // load arr[i] into r0
    add r3, r3, r0                // add arr[i] to sum (r3)
    addi r9, r9, +4               // compute address of arr[i+1]

    lwzx r0, +0(r9)               // same for arr[i+1] ...
    add r3, r3, r0                // ...
    addi r9, r9, +4               // ... due to loop unrolling

    lwzx r0, +0(r9)               // same for arr[i+2] ...
    add r3, r3, r0                // ...
    addi r9, r9, +4               // ... due to loop unrolling

    lwzx r0, +0(r9)               // same for arr[i+3] ...
    add r3, r3, r0                // ...
    addi r9, r9, +4               // ... due to loop unrolling

    cmp cr0, 0, r9, r8            // check whether we are still ...
    blt cr0, 0x18002c8.t <LOOP>   // ... in the bounds of arr
```

If one of the entries of the abstract cache set $\hat{\mathcal{S}}$ and the ongoing memory access are *same block*-related, then the ongoing access is classified as a hit. Otherwise, the access is not classified.

$$\mathrm{hit}(i, \hat{\mathcal{S}}) \overset{\mathrm{def}}{=} \exists j \in \{1, 2\} : \exists\, i_{ref} \in \hat{\mathcal{S}}[j] : i \overset{\mathrm{sb}}{\sim} i_{ref}$$

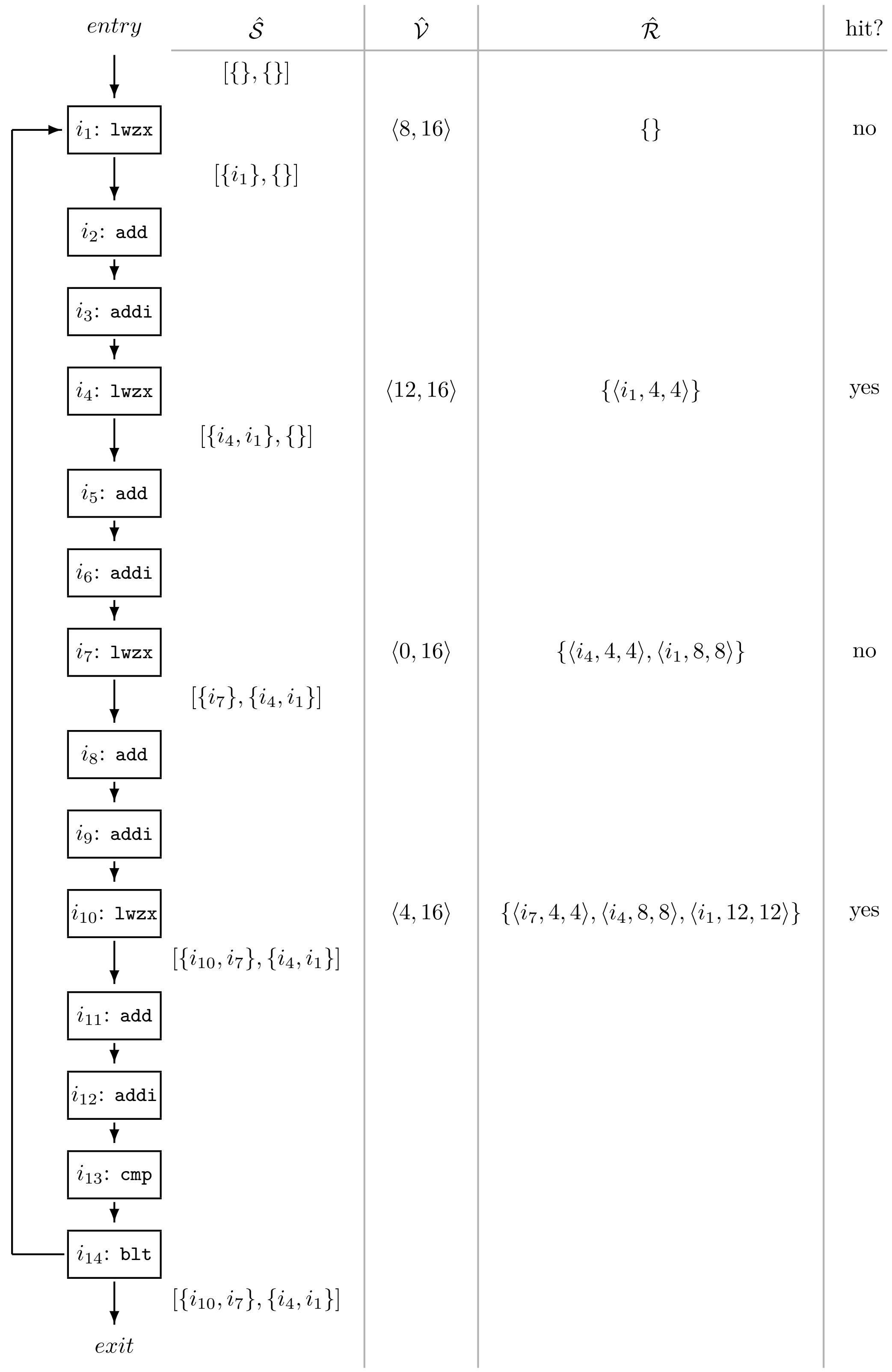
5 Example

To show the relational must analysis in action, we recall the little example used to show the shortcomings of the classical must analysis. The little snippet of C code in listing 3 is compiled to the PowerPC assembler code in listing 4. Note that loop unrolling has been applied.

The control-flow graph of the loop body together with the alignment information and the distance relations is given in Figure 4. The `lwzx` instructions are the only ones which accesses the data memory. The access width of them is four bytes. To enhance readability, only the ingoing and outgoing abstract cache sets of the `lwzx` instructions are shown, because all other instructions do not change the abstract cache sets.

The analysis starts with an empty abstract cache set $\hat{\mathcal{S}}$. Since no distance relations exist for i_1, no cache hit can be predicted. Thus every entry in $\hat{\mathcal{S}}$ ages by one and i_1 is added to the youngest one.

For i_4, there exists a distance relation: the distance between the memory access of i_4 and i_1 is four bytes. To check whether the access of i_4 is a hit, we have to check whether i_4 and i_1 are *same block*-related. To do so, we evaluate the constraint of relation $\overset{\mathrm{sb}}{\sim}$,



	$\hat{\mathcal{S}}$	$\hat{\mathcal{V}}$	$\hat{\mathcal{R}}$	hit?
entry	$[\{\}, \{\}]$			
i_1: lwzx		$\langle 8, 16\rangle$	$\{\}$	no
	$[\{i_1\}, \{\}]$			
i_2: add				
i_3: addi				
i_4: lwzx		$\langle 12, 16\rangle$	$\{\langle i_1, 4, 4\rangle\}$	yes
	$[\{i_4, i_1\}, \{\}]$			
i_5: add				
i_6: addi				
i_7: lwzx		$\langle 0, 16\rangle$	$\{\langle i_4, 4, 4\rangle, \langle i_1, 8, 8\rangle\}$	no
	$[\{i_7\}, \{i_4, i_1\}]$			
i_8: add				
i_9: addi				
i_{10}: lwzx		$\langle 4, 16\rangle$	$\{\langle i_7, 4, 4\rangle, \langle i_4, 8, 8\rangle, \langle i_1, 12, 12\rangle\}$	yes
	$[\{i_{10}, i_7\}, \{i_4, i_1\}]$			
i_{11}: add				
i_{12}: addi				
i_{13}: cmp				
i_{14}: blt				
	$[\{i_{10}, i_7\}, \{i_4, i_1\}]$			
exit				

Figure 4 Example for the relational must cache analysis.

namely $(y \bmod 32) + x_{max} + w \leq \min(z, 32) \wedge (y \bmod 32) + x_{min} \geq 0$. Instantiation gives $8 + 4 + 4 \leq 16 \wedge 8 + 4 \geq 0$. Thus the access of i_4 is a predicted cache hit. Therefore no entry ages, but i_4 is inserted into the youngest one.

For i_7 there exist also some distance relations. Instantiation of the constraint of relation $\overset{\text{sb}}{\sim}$ gives $12 + 4 + 4 \leq 16 \wedge 12 + 4 \geq 0$ and $8 + 8 + 4 \leq 16 \wedge 8 + 8 \geq 0$. Both evaluate to false. Therefore, no hit can be predicted for i_7. All entries age by one and i_7 is added to the youngest one.

At i_{10}, the memory access is *same block*-related to i_7. Thus the access is a predicted cache hit. Again, no entry ages. The youngest entry will consist of i_{10} and i_7 after the update. The other entry stays the same.

Then the second round of the fixed point iteration starts. This time, we must first join the two incoming abstract cache sets at i_1. Set intersection plus maximal age gives $\hat{\mathcal{S}} = [\{\}, \{\}]$. This is the same value as the input in the first round, thus the fixed point iteration stabilizes directly.

6 Precision

In the example above, two of four accesses are classified as cache hits. This is far below the theoretical maximum of $\frac{7}{8}$.

There are two reasons for that. On the one hand, no relations existed for i_1. This is due to the join of flow from the entry and the recursive loop edge. One possibility would be to add some kind of partitioning to keep the values apart.

The other possibility is to shift the first loop iteration with loop peeling such that the first access in the unrolled loop coincides with the one in which the first entry of a cache line is accessed. Then, the access in which naturally no cache hit is predictable and the one in which not enough information is available coincide.

The other reason is the length of the unrolled loop. Only half of a cache line is covered by one iteration of the loop. If the unroll factor is high enough, the congruence information would cover the whole cache line.

Thus, loop peeling and loop unrolling should be applied in such a way that (a) the peeled prefix is long enough to shift the start of the unrolled loop to match the cache line boundaries and (b) the loop is unrolled enough to cover whole cache lines. The former is usually hard to achieve without a preceding data-flow analysis to determine the right choice. The latter can easily be computed: the unroll factor must be a multiple of the cache line size divided by the access width.

For the example, this means a peeled prefix of three array accesses and eight array accesses inside the unrolled loop. Then, the number of predicted cache hits equals the theoretical maximum of $\frac{7}{8}$.

7 Correctness

The following section sketches the proof of correctness of the relational must analysis with *same block* relation $\overset{\text{sb}}{\sim}$. We assume that the used data-flow analyses and the relational cache analysis framework are correct.

▶ **Lemma 1** (Soundness of $\hat{\mathcal{V}}$, $\hat{\mathcal{R}}$). *$\hat{\mathcal{V}}$ contains only sound abstractions of the alignment of memory accesses. $\hat{\mathcal{R}}$ contains only sound abstractions of the linear relations between two memory accesses.*

Proof. Both statements follow from the correctness of the underlying data-flow analyses. ◀

▶ **Lemma 2** (Soundness of $\hat{\mathcal{S}}$). *For any instruction i in a given program, $\hat{\mathcal{S}}$ contains only reference memory accesses for cache lines that are in the cache at the point of execution of i.*

Proof. Follows from the correctness of the relational cache analysis framework. ◀

▶ **Lemma 3** (Soundness of $\overset{\text{sb}}{\sim}$). *Two instruction i, i_{ref} are* same block*-related only if the induced memory accesses of both instructions target the same cache line.*

Proof. The proof is carried out by showing that if two accesses target different cache lines, then they are not *same block*-related.

Let c be the size of the cache, b the size of one cache line. Let instruction i induce a memory access to address a with width $w > 0$. Let instruction i_{ref} induce a memory access to address a_{ref}. Let $a - a_{ref} = x$.

Assume furthermore that $\langle y, z\rangle \in \hat{\mathcal{V}}[i_{ref}]$ and $\langle i_{ref}, x_{min}, x_{max}\rangle \in \hat{\mathcal{R}}[i]$. The last two assumptions are safe as otherwise the constraint of $\overset{\text{sb}}{\sim}$ evaluates to false. From lemma 1, it follows that $y = a_{ref} \bmod z$ and $x_{min} \le x \le x_{max}$.

- Let a go into a cache line after a_{ref}, that is, $a \bmod c > a_{ref} \bmod c$. Then $(a_{ref} \bmod b)+x \geq b$ because the offsets of the original cache line lie in the interval $[0, b-1]$ and $(a_{ref} \bmod b)+x$ points to a later cache line.
  Two cases must be distinguished: (1) $b \le z$ and (2) $b > z$.
  If (1) holds, then $y \bmod b = a_{ref} \bmod b$. Thus $(y \bmod b) + x_{max} \geq b$. From $w > 0$, it follows directly that the constraint evaluates to false.
  If (2) holds, then $y \bmod b = y$. Moreover, $k' \cdot z + y + x \geq k \cdot z$ holds with $k = \frac{b}{z} > 1$ and $0 \le k' = \frac{(a_{ref} \bmod b) - y}{z} < k$. Thus $y + x \geq (k - k') \cdot z$ and $k - k' \geq 1$. Thus $(y \bmod b) + x_{max} \geq z$. From $w > 0$, it follows directly that the constraint evaluates to false.
- Let a go into the same cache line as a_{ref}, $a \bmod b > a_{ref} \bmod b$ and w such that the access crosses the cache line boundary. Then $(a_{ref} \bmod b) + x + w > b$.
  Two cases must be distinguished: (1) $b \le z$ and (2) $b > z$.
  If (1) holds, then $y \bmod b = a_{ref} \bmod b$. Thus $(y \bmod b) + x_{max} + w > b$. It follows directly that the constraint evaluates to false.
  If (2) holds, then $y \bmod b = y$. Moreover, $k' \cdot z + y + x + w > k \cdot z$ holds with $k = \frac{b}{z} > 1$ and $0 \le k' = \frac{(a_{ref} \bmod b) - y}{z} < k$. Thus $y + x + w > (k - k') \cdot z$ and $k - k' \geq 1$. Thus $(y \bmod b) + x_{max} + w > z$. It follows directly that the constraint evaluates to false.
- Let ma go into a cache line before a_{ref}, that is, $a \bmod c < a_{ref} \bmod c$. Then $(a_{ref} \bmod b)+x < 0$ because the offsets of the original cache line lie in the interval $[0, b-1]$ and $(a_{ref} \bmod b) + x$ points to an earlier cache line.
  Two cases must be distinguished: (1) $b \le z$ and (2) $b > z$.
  If (1) holds, then $y \bmod b = a_{ref} \bmod b$. Thus $(y \bmod b) + x_{min} < 0$. It follows directly that the constraint evaluates to false.
  If (2) holds, then $y \bmod b = y$ and $y \le (a_{ref} \bmod b)$. Thus $(y \bmod b) + x_{min} < 0$. It follows directly that the constraint evaluates to false.

◀

▶ **Theorem 4** (Soundness of the relational must cache analysis with *same block* relation $\overset{\text{sb}}{\sim}$). *For any instruction in a given program, a cache hit is only predicted if it would happen in every concrete run of the program.*

Proof. From lemma 2, it follows that at any instruction, $\hat{\mathcal{S}}$ contains only reference memory accesses for cache lines that are currently in the cache. A hit is only predicted, if the given instruction is *same block*-related to a reference memory access in $\hat{\mathcal{S}}$. From lemma 3 follows that two memory accesses are only *same block*-related if they target the same cache line. Thus a hit is only predicted if a memory access targets a cache line that is currently in the cache. ◀

8 Experimental Results

A prototype implementation of the relational cache analysis with *same block* relation $\overset{\text{sb}}{\sim}$ has been integrated into the `aiT` WCET analyzer [1]. With this prototype, the Mälardalen WCET Benchmark [8] has been analyzed. The results of both the classical cache analysis and the relational cache analysis are shown in table 1. For the classical cache analysis, the loops have been 1-fold peeled. For the relational cache analysis, the loops have been 1-fold peeled and 8-fold unrolled.

One interesting property is the data cache hit prediction ratio on the critical path. For most tests, the prediction ratio increased as expected.

A closer look has been taken for those tests where the relational cache analysis did not show the desired improvements. Three tests performed particularly bad: `insertsort`, `fir` and `cover`. They all have in common that no relations could be computed. This is because of the strange control flow generated by the compiler. In `insertsort` for example, two loops are merged into one. On one join point, the relation $r7 = r0$ comes from one edge and the relation $r7 = r5$ from another. As r0 and r5 have different values, no relation remains after the join. A similar thing happens in `fir`. In `cover`, some of the loops are split into two nested ones.

Another programming pattern that is bad for relational cache analysis is unveiled in `crc`. Here, the computation depends heavily on bit operations. These destroyed the relational information since the DBMs only handle linear relations.

One particular interesting test is `lcdnum`. The results of this test show a much higher data cache prediction ratio for the classical cache analysis than for the relational one. This seems counterintuitive at first glance, but there is no error. The reasons for this behaviour are the paths which could be proven infeasible due to the loop unrolling. On these paths are lots of loads which induce cache hits. Thus the data cache prediction ratio is higher than for the relational cache analysis.

Another point of interest is the performance of the relational cache analysis. For most tests, the additional runtime for performing the relational value analysis with difference bound matrices is only a few seconds. The biggest exception is `lms`, where the runtime increased from 3 to 127 seconds. However, the data cache prediction ratio increased from 56% to 78% and the WCET bound decreased by about a quarter. Thus the decreased performance is justified by the increased precision.

9 Related Work

Both Grewe [7] and Flexeder [5] propose to use alignment information to improve the prediction of cache hits. However, they do not work out the details of their ideas.

Hahn and Grund [9] use global value numbering [2] in combination with interval analysis to compute the *same block* relation. This technique, however, is not powerful enough to relate the changing addresses of the array accesses inside the loop.

Table 1 Results for the Mälardalen WCET benchmark. For both the relational cache analysis and the classical cache analysis, it is given the computed WCET bound in cycles, the predicted data cache hit ratio on the critical path and the analysis time in seconds.

	Relational			Classical		
program	WCET	ratio	time	WCET	ratio	time
adcpm	407360	.68	6	569195	.21	4
bs	1230	.08	1	1234	.08	1
bsort100	1827484	.86	14	3205665	.00	2
cnt	20007	.91	5	28668	.51	1
compress	4254976	.30	17	4281933	.30	2
cover	31196	.04	6	43102	.00	8
crc	175060	.17	3	228931	.01	2
duff	13039	.70	2	18425	.00	1
edn	541462	.42	12	836568	.04	3
expint	9999	.72	2	59291	.72	1
fac	824	.33	1	898	.33	1
fdct	11192	.93	2	22740	.15	1
fft1	57139	.93	14	66627	.87	5
fibcall	715	.00	1	715	.00	1
fir	32102	.04	6	36381	.01	1
insertsort	17887	.00	1	17887	.00	1
janne_complex[a]	12633	—	2	12633	—	1
jfdctint	17021	.91	2	32266	.08	1
lcdnum	2339	.57	1	5658	.78	1
lms	3575657	.78	127	4599643	.56	3
ludcmp	13129	.93	6	495695	.01	2
matmult	1570041	.43	44	2225727	.04	2
minver	13283	.70	2	36194	.10	3
ndes	644343	.57	17	759276	.54	3
ns	9744	.86	6	84442	.00	1
nsichneu	292027	.85	14	292027	.85	11
prime	27515	.76	2	27518	.76	1
qsort-exam	497534	.01	34	644161	.00	2
qurt	21101	.73	3	24002	.69	1
recursion	10972	.37	5	12308	.05	1
select	104532	.03	7	111912	.00	1
sqrt	9140	.50	2	20446	.50	1
st[b]	—	—	—	—	—	—
statemate	138338	.97	10	146960	.96	7
ud	9850	.93	3	104877	.01	2

[a] This program contains no loads.
[b] This program could not be compiled.

Lundqvist and Stenström [12] proposed a method to classify data structures as predictable or unpredictable. Unpredictable data structures should then be moved into some uncached memory areas to prevent the eviction of cache entries which buffer parts of the predictable data structures to reduce the overestimation caused by imprecise address information.

Vera and Xue [14] use cache miss equations [6]. They try to relate memory accesses, too, but instead of using an abstract interpretation based data-flow analysis, they compute linear equation systems to count the number of accesses between the reference and the ongoing access. If at least k such accesses exist, they assume a sure miss. Their analysis is restricted to precise address information.

Li, Malik and Wolfe [10, 11] expressed the cache behaviour through integer linear programs. In case of imprecise address information, they add a constraint to the ILP that only one of the addresses is taken. This allows the ILP solver to choose the worst case. This may lead to huge ILPs with unacceptable solving times. Moreover, they do not express mutually exclusive cache misses.

10 Conclusion and Future Work

This paper presented a method to compute *same block* relations for relational cache analysis. The relational cache analysis is a novel cache analysis which is able to predict cache hits even if precise address information is unavailable. It relates memory accesses and checks whether they go to the same cache line. Thus it forms a substantial improvement over Ferdinand's cache analysis which needs precise address information for its predictions.

Using the analyses presented in the work at hand, a *same block* relation $\overset{\text{sb}}{\sim}$ is computed which – in contrast to the one in [9] – is powerful enough to relate memory accesses with changing addresses, for example if array accesses happen inside a loop.

First experiments on the Mälardalen WCET Benchmark [8] show that the presented *same block* relation works well. In the mean, about 52% of the memory accesses have been predicted as cache hits by the relational cache analysis. The classical cache analysis predicts only about 27% cache hits if the loops are not unrolled, and about 64% cache hits if the loops have been fully unrolled to compute globally precise address information.

While loop peeling and loop unrolling are needed to increase the precision of $\overset{\text{sb}}{\sim}$, loop fusion and loop fission may hamper the precision as the experimental results show.

Further improvements are possible: At the moment, only cache hits are predicted by the relational cache analysis, i.e. it is a must analysis. Finding the right congruence analyses to support precise relational may analysis is still an open research question. Moreover, using only $\overset{\text{sb}}{\sim}$ as *same block* relation, the relational cache analysis cannot distinguish between the different cache sets, and all collapse into one, effectively decreasing the achievable precision. To overcome this obstacle, *same set*, *different set* relations need to be computed, too. Both will be targeted in future work.

Acknowledgements This work is based on the author's master's thesis [15], written under the supervision of Prof. Dr. Dr. h.c. Reinhard Wilhelm and Dr. Florian Martin. The author likes to thank Dr. Reinhold Heckmann, Dr. Daniel Grund and the anonymous reviewers for their valuable comments and suggestions.

References

1 AbsInt Angewandte Informatik GmbH. aiT Worst-Case Execution Time Analyzer. `http://www.absint.com/ait/`.
2 Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 1–11. ACM, 1988.
3 David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 197–212, London, UK, 1990. Springer-Verlag.
4 Christian Ferdinand. *Cache Behavior Prediction for Real-Time Systems.* PhD thesis, Saarland University, 1997.
5 Andrea Flexeder. *Interprocedural Analysis of Low-Level Code.* PhD thesis, Technische Universität München, 2011.
6 Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst.*, 21(4):703–746, July 1999.
7 Dominik Grewe. Static Congruence Analysis on Binaries. Bachelor's thesis, Saarland University, 2008.
8 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks - Past, Present and Future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, July 2010.
9 Sebastian Hahn and Daniel Grund. Relational Cache Analysis for Static Timing Analysis. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, Los Alamitos, CA, USA, July 2012. IEEE.
10 Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, RTSS '95, page 298, Washington, DC, USA, 1995. IEEE Computer Society.
11 Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Cache modeling for real-time software: beyond direct mapped instruction caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, RTSS '96, page 254, Washington, DC, USA, 1996. IEEE Computer Society.
12 Thomas Lundqvist and Per Stenström. A method to improve the estimated worst-case performance of data caching. In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, RTCSA '99, page 255, Washington, DC, USA, 1999. IEEE Computer Society.
13 Antoine Miné. A new numerical abstract domain based on difference-bound matrices. In *PADO '01: Proceedings of the Second Symposium on Programs as Data Objects*, pages 155–172, London, UK, 2001. Springer-Verlag.
14 Xavier Vera and Jingling Xue. Let's study whole-program cache behaviour analytically. In *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA '02)*, pages 175–186, 2002.
15 Simon Wegener. Improving Static Analysis of Loops. Master's thesis, Saarland University, 2011.

Toward Static Timing Analysis of Parallel Software*

Andreas Gustavsson, Jan Gustafsson, and Björn Lisper

School of Innovation Design and Engineering, Mälardalen University, Sweden
`{andreas.sg.gustavsson,jan.gustafsson,bjorn.lisper}@mdh.se`

Abstract

The current trend within computer, and even real-time, systems is to incorporate parallel hardware, e.g., multicore processors, and parallel software. Thus, the ability to safely analyse such parallel systems, e.g., regarding the timing behaviour, becomes necessary. Static timing analysis is an approach to mathematically derive *safe* bounds on the execution time of a program, when executed on a given hardware platform. This paper presents an algorithm that statically analyses the timing of parallel software, with threads communicating through shared memory, using abstract interpretation. It also gives an extensive example to clarify how the algorithm works.

1998 ACM Subject Classification F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages–*Program Analysis*

Keywords and phrases Parallelism, BCET, WCET, Static analysis, Abstract interpretation

Digital Object Identifier 10.4230/OASIcs.WCET.2012.38

1 Introduction

Many safety-critical embedded systems have hard real-time requirements. For these, safe bounds on the Best- and Worst-Case Execution Times (BCET/WCET) of the tasks in the system are key measures. Together, they define an interval in time within which the execution of the task is guaranteed to finish. In particular WCET bounds are needed by, e.g., schedulability analyses.

For reasons of energy consumption and performance, development in hardware today strives toward massively parallel architectures, like many-core, GPU and even special purpose, heterogeneous platforms. Thus, it is very likely that software tasks in future real-time systems will be parallel in order to utilise the provided computing power. Therefore, efforts must be made in providing WCET analyses for such systems.

This paper focuses on analysing the timing behaviour of *parallel software* with *dependent* sub-tasks, using a programming model with threads, shared memory, and locks. This kind of programming model is commonly used in parallel software today. It is assumed that an arbitrary underlying timing model, which can predict safe bounds on the BCET and WCET of individual instructions given a certain system state, is provided. An algorithm to statically derive the BCET and WCET of parallel software using abstract interpretation is presented. A technical report [8] covers the details and correctness proofs of the algorithm.

The rest of this paper is organised as follows. Section 2 presents related work on static timing analysis for parallel systems. Section 3 introduces a small model parallel language, with threads, thread-local and global memory, and locks. We also give a formal semantics for the language, including time, and we then present an analysis based on abstract interpretation. Section 4 clarifies how the analysis works by instantiating it for a given example program. Section 5 concludes the presentation with some discussion and directions for the future.

* This work was partly funded by the Swedish Research Council (VR) through project 2008-4650, "Worst-Case Execution Time Analysis of Parallel Systems".

2 Related Work

As far as we know, there have not been many attempts to statically analyse the execution time of explicitly parallel software. The parMERASA project provides a timing analysable multicore CPU with a system level software (c.f., operating system). In [11], a case study is performed in which the WCET of a parallel 3D multigrid solver, executing on the MERASA platform, is derived. In [7], model-checking is used to derive the WCET of a minimal parallel program. It is shown that, since model-checking is based on exhaustive exploration of concrete states, it is difficult to achieve scalability using only the presented approach. In [9], abstract interpretation is combined with model-checking to avoid the found scalability problems. This work does not focus on explicitly parallel (e.g., threaded) software, though.

In [3], an approach to directly calculate the BCET and WCET for sequential programs using abstract execution [6] is presented. Our work takes basically the same approach, but for explicitly parallel programs.

There is also some research on static low-level analysis of parallel systems. In [1] and [12], static methods for analysing multicores with a shared L2 instruction cache are presented. In [1], effects from timing anomaly influenced pipelines are also taken into account.

3 Timing Analysis

In this section, an algorithm for timing analysis of programs containing *dependent* parallel threads will be defined. It is assumed that the underlying architecture consists of both thread-private and global memory, referred to as registers, $r \in \mathbf{Reg}$, and variables, $x \in \mathbf{Var}$, respectively, and that arithmetical operations etc. can be performed using values of registers. It will also be assumed that shared resources that can be acquired in a mutually exclusive manner by the threads are provided, and that the operations provided by the instruction set (statements) may have variable execution times. (C.f., multicore CPU:s, where you have local and global memory, a shared memory bus and mutual exclusion operations.) No further assumptions on the underlying architecture, e.g., the number of CPU:s, the memory hierarchy or whether an operating system is used, are made. Timing effects from such features should not be considered in the software model but in the model of the underlying architecture.

3.1 Abstract Interpretation

In general, a timing analysis based on the concrete semantic of a program is infeasible due to the enormous number of states that must be explored. Abstract interpretation [2, 4, 10] is a method for *safely* approximating the concrete program semantics and can be used to obtain a set of possible abstract states for each point in a program. An abstract state describes, and sometimes over-approximates, the information given by a *set* of concrete semantic states. This means that an analysis based on abstractly interpreting the semantics of a program can become less complex and more efficient, but might suffer from imprecision, compared to an analysis based on the concrete semantics.

The concrete semantics of a programming language can be abstracted in many different ways. The choice of abstraction is done by defining an abstract domain. An abstract domain is essentially the set of all possible abstract states that fit the definition of the domain. It is often shown that the abstract domain is a safe over-approximation of the concrete domain by deriving a Galois connection (an abstraction function, α, and a concretisation function, γ) between the two domains [10]. An example of an abstract value domain is $\mathbf{Intv} = \{[z_1, z_2] \mid int_{min} \leq z_1 \leq z_2 \leq int_{max} \land z_1, z_2, int_{min}, int_{max} \in \mathbb{Z}\}$, i.e., the set of all intervals that "fit in" $[int_{min}, int_{max}]$. This domain can be used to over-approximate

$$
\begin{array}{llll}
P ::= T \mid P \parallel T & s ::= [\texttt{halt}]^l \mid [\texttt{skip}]^l \mid [r \texttt{ := } a]^l \mid [\texttt{if } b \texttt{ goto } l']^l \mid s_1 \, ; s_2 \mid \\
& \quad\;\; [\texttt{load } r \texttt{ from } x]^l \mid [\texttt{store } r \texttt{ to } x]^l \mid [\texttt{lock } lck]^l \mid [\texttt{unlock } lck]^l \\
T ::= (N, s) & a ::= n \mid r \mid a_1 \texttt{ + } a_2 \mid a_1 \texttt{ - } a_2 \mid a_1 \texttt{ * } a_2 \mid a_1 \texttt{ / } a_2 \\
& b ::= \texttt{true} \mid \texttt{false} \mid \texttt{!} b \mid b_1 \texttt{ \&\& } b_2 \mid a_1 \texttt{ == } a_2 \mid a_1 \texttt{ <= } a_2
\end{array}
$$

Figure 1 The parallel programming language.

$\textsc{stm}(T, pc)$	$\langle pc', \mathbb{r}', \mathbb{x}', \mathbb{l}' \rangle$	Condition
$[\texttt{halt}]^{pc}$	$\langle pc, \mathbb{r}, \mathbb{x}, \mathbb{l} \rangle$	–
$[\texttt{skip}]^{pc}$	$\langle pc + 1, \mathbb{r}, \mathbb{x}, \mathbb{l} \rangle$	–
$[r \texttt{ := } a]^{pc}$	$\langle pc + 1, \mathbb{r}[r \mapsto \mathcal{A}[\![a]\!]\mathbb{r}], \mathbb{x}, \mathbb{l} \rangle$	–
$[\texttt{load } r \texttt{ from } x]^{pc}$	$\langle pc + 1, \mathcal{R}(r, \mathbb{r}, x, \mathbb{x}), \mathbb{x}, \mathbb{l} \rangle$	–
$[\texttt{store } r \texttt{ to } x]^{pc}$	$\langle pc + 1, \mathbb{r}, \mathbb{x}[x \mapsto (\mathbb{x}\ x)[T \mapsto \{(\mathbb{r}\ r, t)\}]], \mathbb{l} \rangle$	–
$[\texttt{if } b \texttt{ goto } l]^{pc}$	$\langle pc + 1, \mathbb{r}, \mathbb{x}, \mathbb{l} \rangle$	$\neg\mathcal{B}[\![b]\!]\mathbb{r}$
$[\texttt{if } b \texttt{ goto } l]^{pc}$	$\langle l, \mathbb{r}, \mathbb{x}, \mathbb{l} \rangle$	$\mathcal{B}[\![b]\!]\mathbb{r}$
$[\texttt{lock } lck]^{pc}$	$\langle pc, \mathbb{r}, \mathbb{x}, \mathbb{l} \rangle$	$\textsc{own}(\mathbb{l}\ lck) \neq T$
$[\texttt{lock } lck]^{pc}$	$\langle pc + 1, \mathbb{r}, \mathbb{x}, \mathbb{l}[lck \mapsto (locked, T)] \rangle$	$\textsc{own}(\mathbb{l}\ lck) = T$
$[\texttt{unlock } lck]^{pc}$	$\langle pc + 1, \mathbb{r}, \mathbb{x}, \mathbb{l}[lck \mapsto (unlocked, \bot_{thrd})] \rangle$	–

where $\mathcal{R}(r, \mathbb{r}, x, \mathbb{x}) = \mathbb{r}[r \mapsto v]$ and $\{(v, t')\} = \bigcup_{T' \in \mathbf{Thrd}} ((\mathbb{x}\ x)\ \ T')$

Figure 2 Semantics of concrete axiom transitions: $\langle T, pc, \mathbb{r}, \mathbb{x}, \mathbb{l}, t \rangle \xrightarrow[ax]{} \langle pc', \mathbb{r}', \mathbb{x}', \mathbb{l}' \rangle$.

the concrete domain $\{z \mid int_{min} \leq z \leq int_{max} \wedge z, int_{min}, int_{max} \in \mathbb{Z}\}$, i.e., the set of all integers between (and including) int_{min} and int_{max}. It is easy to show that there exists a Galois connection between the domains **Intv** and $\mathcal{P}(\mathbb{Z})$ (see e.g., [4, 8, 10]), and thus the approximation is safe, given the abstraction function $\alpha_{int}(Z) = [\min(Z), \max(Z)]$ and the concretisation function $\gamma_{int}([z_1, z_2]) = \{z \in \mathbb{Z} \mid z_1 \leq z \leq z_2\}$.

3.2 A Parallel Programming Language

The analysis will be based on the parallel programming language defined in Fig. 1, which is a set of operations using the discussed architectural features. $P \in$ **Prg** denotes a program, which simply is a number of threads, denoted by $T \in$ **Thrd**. A thread is a pair of a statement, $s \in$ **Stm**, and a unique identifier, $N \in$ **ThrdID**. This makes every thread unique and distinguishable from other threads, even if several threads contain the same statement. To increase the readability of the semantics, it will be assumed that the axiom-statements (all statements except the sequentially composed statement, $s_1 \, ; s_2$) of each thread are uniquely labelled with consecutive labels, $l \in$ **Lbl**, and stored in an array-like fashion in ascending order of their labels. $a \in$ **Aexp** and $b \in$ **Bexp** denote an arithmetic and a boolean expression, respectively, $n \in$ **Val** is an integer value, and $lck \in$ **Lck** denotes a lock. Locks can be acquired in a mutually exclusive manner using `lock` and released using `unlock`. Values can be transferred between variables and registers using `load` and `store`. Conditional branching is performed using `if`, a register is assigned a value using `:=`, a no-operation is performed using `skip`, and `halt` stops the execution of the issuing thread. The arithmetical, boolean and relational operators are self-explanatory and will not be discussed further.

The semantics of the language is formally defined in Fig. 2 (individual axiom statements) and 3 (system of threads). $\mathbb{x} \in \mathbf{Var} \to \mathbf{Thrd} \to \mathcal{P}(\mathbf{Val} \times \mathbf{Time})$, $\mathbb{l} \in \mathbf{Lck} \to (\mathbf{Lck_{stt}} \times \mathbf{Thrd} \cup \{\bot_{thrd}\})$, where $\mathbf{Lck_{stt}} = \{unlocked, locked\}$, and $t \in \mathbf{Time}$ are the states for variables and locks, and the current time. For each thread, T, in the program, there is also $pc_T \in \mathbf{Lbl}_T$, $\mathbb{r}_T \in \mathbf{Reg}_T \to \mathbf{Val}$, $t^r_T \in \mathbf{Time}$ and $t^a_T \in \mathbf{Time}$, which are the states of the program counter and registers of T, the relative execution time of T's active statement, $\textsc{stm}(T, pc_T)$, and the accumulated execution time for T, respectively. The tuple collecting

$$\frac{\forall T \in \mathbf{Thrd}_{exe} : \langle T, pc_T, \mathbb{r}_T, \mathbb{x}, \mathbb{l}'', {t^a_T}' \rangle \xrightarrow[ax]{} \langle pc'_T, \mathbb{r}'_T, \mathbb{x}'_T, \mathbb{l}'_T \rangle}{\begin{array}{l}\langle \{(T, pc_T, \mathbb{r}_T, t^r_T, t^a_T) \mid T \in \mathbf{Thrd}\}, \mathbb{x}, \mathbb{l}, t \rangle \xrightarrow[prg]{} \\ \qquad \langle \{(T, pc'_T, \mathbb{r}'_T, {t^r_T}', {t^a_T}') \mid T \in \mathbf{Thrd}\}, \mathbb{x}', \mathbb{l}', t' \rangle\end{array}}$$

where

$${t^r_T}' = \begin{cases} \textsc{finTime}(\langle \{(T, pc_T, \mathbb{r}_T, t^r_T, t^a_T) \mid T \in \mathbf{Thrd}\}, \mathbb{x}, \mathbb{l}, t \rangle, T) & \textbf{if } t = t^a_T \\ t^r_T & \textbf{otherwise} \end{cases}$$

$$t' = \min(\{t^a_T + {t^r_T}' \mid T \in \mathbf{Thrd}\})$$

$${t^a_T}' = \begin{cases} t^a_T + {t^r_T}' & \textbf{if } t' = t^a_T + {t^r_T}' \\ t^a_T & \textbf{otherwise} \end{cases}$$

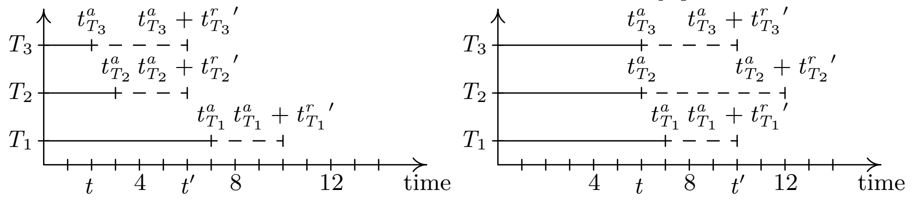
$$\mathbf{Thrd}_{exe} = \{T \in \mathbf{Thrd} \mid t' = {t^a_T}'\}$$

$$(\mathbb{x}'\ x)\ T = \begin{cases} \begin{cases} (\mathbb{x}'_T\ x)\ T & \text{for some } T \in \mathbf{Thrd}_{exe} : \exists r \in \mathbf{Reg}_T : \textsc{stm}(T, pc_T) = [\texttt{store}\ r\ \texttt{to}\ x]^{pc_T} \\ \emptyset & \text{for } T' \in \mathbf{Thrd} \setminus \{T\}, \textbf{ if} \text{ such a } T \text{ exists} \end{cases} \\ (\mathbb{x}\ x)\ T & \textbf{otherwise} \end{cases}$$

$$\mathbb{l}''\ lck = \begin{cases} (unlocked, T) & \text{for some } T \in \mathbf{Thrd}_{exe} : \textsc{stm}(T, pc_T) = [\texttt{lock}\ \ lck]^{pc_T}, \textbf{ if} \text{ such} \\ & T \text{ exists, } \textsc{stt}(\mathbb{l}\ lck) = unlocked \text{ and } \textsc{own}(\mathbb{l}\ lck) = \bot_{thrd} \\ \mathbb{l}\ lck & \textbf{otherwise} \end{cases}$$

$$\mathbb{l}'\ lck = \begin{cases} \mathbb{l}'_T\ lck & \text{for some } T \in \mathbf{Thrd}_{exe} : (\textsc{stm}(T, pc_T) = [\texttt{unlock}\ lck]^{pc_T} \vee \\ & (\textsc{own}(\mathbb{l}''\ lck) = T \wedge \textsc{stm}(T, pc_T) = [\texttt{lock}\ lck]^{pc_T})), \textbf{ if} \text{ such } T \text{ exists} \\ \mathbb{l}\ lck & \textbf{otherwise} \end{cases}$$

Figure 3 Semantics of concrete program transitions: $\langle \mathbf{Ts}, \mathbb{x}, \mathbb{l}, t \rangle \xrightarrow[prg]{} \langle \mathbf{Ts}', \mathbb{x}', \mathbb{l}', t' \rangle$.



(a) c_1: $\mathbf{Thrd}_{exe} = \{T_2, T_3\}$ **(b)** c_2: $\mathbf{Thrd}_{exe} = \{T_1, T_3\}$

Figure 4 Illustration of how $\mathbf{Thrd}_{exe}$ is determined ($c_1 \xrightarrow[prg]{} c_2$).

all these states will be referred to as a configuration, c, i.e., $c = \langle \{(T, pc_T, \mathbb{r}_T, t^r_T, t^a_T) \mid T \in \mathbf{Thrd}\}, \mathbb{x}, \mathbb{l}, t \rangle$. Note that states are updated on transitions, i.e., when pc is updated.

The state for locks keeps track of the state and owner of each lock. The owner is $\bot_{thrd}$ if no thread currently has the lock acquired. The state for registers of thread T simply keeps track of the current value of each register within T. The state for variables is not as intuitive. To be precise, the abstraction of the state for variables will need to save write history, i.e., what abstract writes (a pair of value and time) have been performed by each thread on each variable (see Section 3.3). Therefore, to derive a Galois connection (and hence implicitly get a safe approximation), the concrete state for variables has to be defined accordingly. In the concrete semantics, only one single write is saved for each variable, though. This write is non-deterministically chosen from one of the threads, if any, writing the variable at any given point in time (see Fig. 3). $\mathcal{R}$ is defined to return the value of the saved write (see Fig. 2).

$\mathcal{A} : \mathbf{Aexp} \to (\mathbf{Reg} \to \mathbf{Val}) \to \mathbf{Val}$ and $\mathcal{B} : \mathbf{Bexp} \to (\mathbf{Reg} \to \mathbf{Val}) \to \mathbf{Bool}$ evaluate arithmetic and boolean expressions, respectively, given a particular register state. The details of these functions are straightforward and can be found in [8]. $\textsc{finTime}$ is assumed to be provided by a timing-model of the underlying hardware. It should return a relative execution time for the statement of thread T, i.e., $\textsc{stm}(T, pc_T)$, based on the current system state. The set of threads to execute, $\mathbf{Thrd}_{exe}$, is determined based on t, $t^{r\prime}$ and t^a. It simply consists of the threads that will update their pc:s at the nearest point in time, t'. An illustration of how ${t^r_T}'$, t^a_T, t and t' are used to determine $\mathbf{Thrd}_{exe}$ is given in Fig. 4. For the arbitrary configuration c_1 in Fig. 4a, $t' = 6$ and hence $\mathbf{Thrd}_{exe} = \{T_2, T_3\}$. For c_2 (note that $c_1 \xrightarrow[prg]{} c_2$)

$\textsc{stm}(T, pc)$	$\langle pc', \tilde{\mathbb{r}}', \tilde{\mathbb{x}}', \mathbb{l}' \rangle$	Condition
$[\texttt{halt}]^{pc}$	$\langle pc, \tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \mathbb{l} \rangle$	–
$[\texttt{skip}]^{pc}$	$\langle pc+1, \tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \mathbb{l} \rangle$	–
$[r \texttt{ := } a]^{pc}$	$\langle pc+1, \tilde{\mathbb{r}}[r \mapsto \tilde{\mathcal{A}}[\![a]\!]\tilde{\mathbb{r}}], \tilde{\mathbb{x}}, \mathbb{l} \rangle$	–
$[\texttt{load } r \texttt{ from } x]^{pc}$	$\langle pc+1, \tilde{\mathbb{r}}[r \mapsto \textsc{read}(\tilde{\mathbb{x}}, x, T, \tilde{t})]\ \tilde{\mathbb{x}}, \mathbb{l} \rangle$	–
$[\texttt{store } r \texttt{ to } x]^{pc}$	$\langle pc+1, \tilde{\mathbb{r}}, \textsc{write}(T, \tilde{\mathbb{x}}, x, (\tilde{\mathbb{r}}\ r, \tilde{t})), \mathbb{l} \rangle$	–
$[\texttt{if } b \texttt{ goto } l]^{pc}$	$\langle pc+1, \tilde{\mathcal{BR}}[\![\texttt{!}b]\!]\tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \mathbb{l} \rangle$	$\tilde{\mathcal{BR}}[\![\texttt{!}b]\!]\tilde{\mathbb{r}} \neq \tilde{\bot}_{reg}$
$[\texttt{if } b \texttt{ goto } l]^{pc}$	$\langle l, \tilde{\mathcal{BR}}[\![b]\!]\tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \mathbb{l} \rangle$	$\tilde{\mathcal{BR}}[\![b]\!]\tilde{\mathbb{r}} \neq \tilde{\bot}_{reg}$
$[\texttt{lock } lck]^{pc}$	$\langle pc, \tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \mathbb{l} \rangle$	$\textsc{own}(\mathbb{l}\ lck) \neq T$
$[\texttt{lock } lck]^{pc}$	$\langle pc+1, \tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \mathbb{l}[lck \mapsto (locked, T)] \rangle$	$\textsc{own}(\mathbb{l}\ lck) = T$
$[\texttt{unlock } lck]^{pc}$	$\langle pc+1, \tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \mathbb{l}[lck \mapsto (unlocked, \bot_{thrd})] \rangle$	–

where $\tilde{\mathcal{BR}}[\![b]\!]\tilde{\mathbb{r}} = \alpha_{reg}(\{\mathbb{r} \in \gamma_{reg}(\tilde{\mathbb{r}}) \mid \mathcal{B}[\![b]\!]\mathbb{r}\})$

Figure 5 Semantics of abstract axiom transitions: $\langle T, pc, \tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t} \rangle \xrightarrow[ax]{\sim} \langle pc', \tilde{\mathbb{r}}', \tilde{\mathbb{x}}', \mathbb{l}' \rangle$.

in Fig. 4b, $t' = 10$ and hence $\mathbf{Thrd}_{exe} = \{T_1, T_3\}$.

The behaviour of locks needs to be explained. Assume that some threads in $\mathbf{Thrd}_{exe}$ execute a `lock`-statement on some lock, *lck*, and that *lck* is *unlocked* in the given configuration. In the resulting configuration, $\textsc{stt}(\mathbb{l}'\ lck) = locked$ and the owner will be one of the threads that tried to acquire *lck*. The chosen thread is given by $\textsc{own}(\mathbb{l}''\ lck)$; note that $\mathbb{l}''$ is only used to control the behaviour of the rules for `lock` in Fig. 2. This thread will have incremented its *pc* and thus moved on to executing its next statement. All other threads that tried to acquire *lck* will again try to acquire *lck* since their *pc*:s are not changed. Note that the latter would also be the case for *all* threads in $\mathbf{Thrd}_{exe}$ that try to acquire an already locked lock that is not owned by themselves. Also note that a thread who owns a lock is allowed to repeatedly acquire this lock any number of times.

3.3 Abstractly Interpreting the Language Semantics

First, it must be decided what parts of the system state to interpret in an abstract way. To allow for the hardware timing-model to be abstracted as well, **Time** will be approximated using the interval domain, i.e., $\tilde{\mathbf{Time}} = \mathbf{Intv}$. This approach is also taken by Chattopadhyay et al. [1] to approximate the execution time of pipeline stages in order to deal with timing anomalies in multicore platforms. **Val** will also be abstracted using intervals, i.e., $\tilde{\mathbf{Val}} = \mathbf{Intv}$, to allow for an efficient handling of data flow. Since **Thrd**, **Lbl**, **Var**, **Reg**, **Lck**, **Aexp** and **Bexp** are defined by the software, it does not make any sense to abstract them for the defined analysis (see Section 3.4). And, since $\mathbf{Lck}_{\mathbf{stt}}$ is comparable to **Bool**, an abstraction of it would not be very beneficial. The states implicitly affected by the abstractions of **Time** and **Val** are $\mathbb{r}$, $\mathbb{x}$, t^r, t^a, t, and thus c. The abstraction of these will be referred to as $\tilde{\mathbb{r}}$, $\tilde{\mathbb{x}}$, $\tilde{t}^r$, $\tilde{t}^a$, $\tilde{t}$ and $\tilde{c}$, respectively. In [8], it is shown that Galois connections (with the corresponding abstraction and concretisation functions) can be established between the concrete and abstract domains for these states, and thus, that the approximations are safe. It is also shown that the abstract axiom transition rules (including the abstract version of $\mathcal{A}$, i.e., $\tilde{\mathcal{A}}$) in Fig. 5 are safe approximations of the concrete rules in Fig. 2, and that the boolean restriction function, $\tilde{\mathcal{BR}}$, is safe. Note that the concretisation of $\tilde{\mathcal{BR}}[\![b]\!]\tilde{\mathbb{r}}$ will always contain (at least) the concrete stores, derived from $\tilde{\mathbb{r}}$, in which b evaluates to `true`.

$\tilde{\mathbb{x}} \in \mathbf{Var} \to \mathbf{Thrd} \to \mathcal{P}(\tilde{\mathbf{Val}} \times \tilde{\mathbf{Time}})$ can save any number (i.e., the history) of abstract writes, $\tilde{w} \in \tilde{\mathbf{Val}} \times \tilde{\mathbf{Time}}$, for each thread that occur on some variable. This is done to increase the precision in the analysis, since then, sequence (within each thread) and timing information (between threads) can be used to get a tight value when reading a variable. $\textsc{write}(T, \tilde{\mathbb{x}}, x, \tilde{w})$ is thus defined to simply add the write, $\tilde{w}$, to the set of write-history for

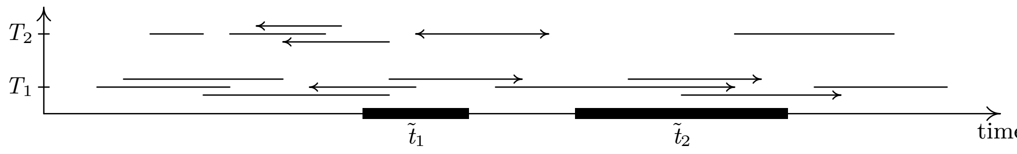


Figure 6 The time-stamps of the writes considered by $\textsc{read}(\tilde{\mathbb{x}}, x, T_1, \tilde{t}_1)$ and $\textsc{read}(\tilde{\mathbb{x}}, x, T_2, \tilde{t}_2)$.

$$\frac{\forall T \in \mathbf{Thrd}_{exe} : \langle T, pc_T, \tilde{\mathbb{r}}_T, \tilde{\mathbb{x}}, \mathbb{l}'', \tilde{t}^{a}_{T}{}' \rangle \xrightarrow[ax]{\sim} \langle pc'_T, \tilde{\mathbb{r}}'_T, \tilde{\mathbb{x}}'_T, \mathbb{l}'_T \rangle}{\langle \{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}^r_T, \tilde{t}^a_T) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t} \rangle \xrightarrow[prg]{\sim} \langle \{(T, pc'_T, \tilde{\mathbb{r}}'_T, \tilde{t}^r_T{}', \tilde{t}^a_T{}') \mid T \in \mathbf{Thrd}_{\tilde{c}}\}, \tilde{\mathbb{x}}', \mathbb{l}', \tilde{t}' \rangle}$$

where

$$\tilde{t}^r_T{}' = \begin{cases} \textsc{AbsFinTime}\ (\langle \{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}^r_T, \tilde{t}^a_T) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t} \rangle, T) & \textbf{if } \tilde{t} \,\tilde{\sqcap}_t\, \tilde{t}^a_T \neq \tilde{\bot}_t \\ \tilde{t}^r_T & \textbf{otherwise} \end{cases}$$

$$\tilde{t}' = \alpha_t(\{t_{min}, t_{max}\}) \qquad \textbf{where } t_{min} = \min\{\min(\gamma_t(\tilde{t}^a_T \tilde{+}_t \tilde{t}^r_T{}')) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}$$
$$t_{max} = \min\{\max(\gamma_t(\tilde{t}^a_T \tilde{+}_t \tilde{t}^r_T{}')) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}$$

$$\tilde{t}^a_T{}' = \begin{cases} \tilde{t}^a_T \tilde{+}_t \tilde{t}^r_T{}' & \textbf{if } \tilde{t}' \,\tilde{\sqcap}_t (\tilde{t}^a_T \tilde{+}_t \tilde{t}^r_T{}') \neq \tilde{\bot}_t \\ \tilde{t}^a_T & \textbf{otherwise} \end{cases}$$

$$\mathbf{Thrd}_{exe} = \{T \in \mathbf{Thrd}_{\tilde{c}} \mid \tilde{t}' \,\tilde{\sqcap}_t\, \tilde{t}^a_T{}' \neq \tilde{\bot}_t\}$$
$$(\tilde{\mathbb{x}}'\ x)\ T = (\tilde{\mathbb{x}}'_T\ x)\ T$$
$\mathbb{l}''\ lck = \ldots$ (same as in Fig. 3)
$\mathbb{l}'\ lck = \ldots$ (same as in Fig. 3)

Figure 7 Semantics of abstract program transitions: $\langle \tilde{\mathbf{T}}\mathbf{s}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t} \rangle \xrightarrow[prg]{} \langle \tilde{\mathbf{T}}\mathbf{s}', \tilde{\mathbb{x}}', \mathbb{l}', \tilde{t}' \rangle$.

thread T, i.e., to $((\tilde{\mathbb{x}}\ x)\ T)$. Using the sequence and timing information, $\textsc{read}(\tilde{\mathbb{x}}, x, T, \tilde{t})$ is defined to only take the writes that might be valid at $\tilde{t}$ (the point in time when T issues the $\textsc{read}$) into consideration for its returned value $\tilde{v} \in \mathbf{V\tilde{a}l}$. These writes, $\tilde{w} = (\tilde{v}', \tilde{t}')$, come from two categories. The first category covers the writes on x for threads $T' \neq T$ whose "time-stamps" overlap in time with $\tilde{t}$, i.e., $\tilde{t} \,\tilde{\sqcap}_t\, \tilde{t}' \neq \tilde{\bot}_t$. The second category covers the most recent write on x for all threads (including T) such that its time-stamp overlaps with the overall most recent write of any write, not belonging to the first category. Note that any write for thread T with a time-stamp that begins after the beginning of $\tilde{t}$ is discarded. So is any write for $T' \neq T$ such that its time-stamp completely succeeds $\tilde{t}$. This is because such writes can simply not have occurred at the time of the $\textsc{read}$ (and will thus usually not be included in $\tilde{\mathbb{x}}$ at all). An illustration of the time-stamps of the writes on x, by some threads T_1 and T_2, stored in $\tilde{\mathbb{x}}$, that must be considered by $\textsc{read}(\tilde{\mathbb{x}}, x, T_1, \tilde{t}_1)$ (lines with arrow heads pointing left) and $\textsc{read}(\tilde{\mathbb{x}}, x, T_2, \tilde{t}_2)$ (lines with arrow heads pointing right) is given in Fig. 6. The returned value, $\tilde{v}$, is the least upper bound of the values of the considered writes.

The abstract transition rule for program configurations in Fig. 7 is an approximation of the concrete rule in Fig. 3. The abstract rule now defines a window in time, $\tilde{t}'$, that determines which threads are included in $\mathbf{Thrd}_{exe}$. The window reaches from the earliest point in time when some thread might update its *pc*, to the earliest point in time when some *pc* must be updated. AbsFinTime is assumed to be a safe approximation of finTime.

The abstract rule in Fig. 7 is a safe approximation of the concrete rule in Fig. 3 only if some certain conditions are met. It is safe given that $|\mathbf{Thrd}_{\tilde{c}}| = 1$, or if a `load`-, `lock`- or `unlock`-statement is not executed by any thread in $\mathbf{Thrd}_{exe}$ [8]. This is easy to see since if these conditions are met, the threads in $\mathbf{Thrd}_{exe}$ execute independently from each other. If some thread in $\mathbf{Thrd}_{exe}$ would execute for example a `load`-statement, dependencies are introduced between the threads, and the $\textsc{read}$ function could return a value for which all possible writes have not been taken into account. Let's assume that $\mathbf{Thrd}_{exe} = \{T_1, T_2\}$, $\textsc{stm}(T_1, pc_{T_1}) = [\texttt{load}\ r\ \texttt{from}\ x]^{pc_{T_1}}$, $\textsc{stm}(T_2, pc_{T_2}) = [\texttt{skip}]^{pc_{T_2}}$

```
function ABSTRACTEXECUTION($\tilde{c}, \tilde{t}_{to}$)
    $workset \leftarrow \{\tilde{c}\}$,    $finalset \leftarrow \emptyset$
    repeat
        $\tilde{c}@\langle\{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}^r_T, \tilde{t}^a_T) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t}\rangle \leftarrow$ CHOOSE($workset$)
        $workset \leftarrow workset \setminus \{\tilde{c}\}$
        if ISTIMEOUT($\tilde{c}, \tilde{t}_{to}$) $\vee$ ISFINAL($\tilde{c}$) then
            $finalset \leftarrow finalset \cup \{\tilde{c}\}$
        else
            $\mathbf{Thrd}_{\texttt{load}} \leftarrow$ LOADTHRD($\tilde{c}$)
            if $\mathbf{Thrd}_{\texttt{load}} \neq \emptyset \wedge |\mathbf{Thrd}_{\tilde{c}}| > 1$ then
                for all $T' \in \mathbf{Thrd}_{\texttt{load}}$ do
                    $\tilde{t}^r_{T'}{}' \leftarrow$ ABSFINTIME($\tilde{c}, T'$)
                    $x \leftarrow$ GETVARLOAD(STM($T', pc_{T'}$)),    $r \leftarrow$ GETREGLOAD(STM($T', pc_{T'}$))
                    $\tilde{v} \leftarrow \tilde{\bot}_{val}$,    $\tilde{c}' \leftarrow \langle\{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}^r_T, \tilde{t}^a_T) \mid T \in \mathbf{Thrd}_{\tilde{c}} \setminus \{T'\}\}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t}\rangle$
                    $\tilde{C}^f_{T'} \leftarrow$ ABSTRACTEXECUTION($\tilde{c}', (\tilde{t}^a_{T'} +_t \tilde{t}^r_{T'}{}') \tilde{\sqcap}_t \tilde{t}_{to}$)
                    for all $\langle\tilde{\mathbf{T}}\mathbf{s}, \tilde{\mathbb{x}}', \mathbb{l}', \tilde{t}'\rangle \in \tilde{C}^f_{T'}$ do
                        $\tilde{v} \leftarrow \tilde{v} \,\tilde{\sqcup}_{val}$ READ($\tilde{\mathbb{x}}', x, T', \tilde{t}^a_{T'} +_t \tilde{t}^r_{T'}{}'$)
                    end for
                    $pc'_{T'} \leftarrow pc_{T'} + 1$,    $\tilde{\mathbb{r}}'_{T'}\ r' \leftarrow \begin{cases} \tilde{v} & \textbf{if } r = r' \\ \tilde{\mathbb{r}}_{T'}\ r' & \textbf{otherwise} \end{cases}$
                end for
                $\tilde{c}' \leftarrow \langle\{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}^r_T, \tilde{t}^a_T) \mid T \in \mathbf{Thrd}_{\tilde{c}} \setminus \mathbf{Thrd}_{\texttt{load}}\} \cup$
                          $\{(T, pc'_T, \tilde{\mathbb{r}}'_T, \tilde{t}^r_T{}', \tilde{t}^a_T +_t \tilde{t}^r_T{}') \mid T \in \mathbf{Thrd}_{\texttt{load}}\}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t}\rangle$
                $workset \leftarrow workset \cup \{\tilde{c}'\}$
            else
                $\tilde{C} \leftarrow \{\tilde{c}' \mid \tilde{c} \xrightarrow[prg]{\sim} \tilde{c}'\}$
                $\tilde{C}' \leftarrow \{\langle\tilde{\mathbf{T}}\mathbf{s}, \text{TRIM}(\tilde{\mathbb{x}}, \tilde{t}), \mathbb{l}, \tilde{t}\rangle \mid \langle\tilde{\mathbf{T}}\mathbf{s}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t}\rangle \in \tilde{C}\}$
                $workset \leftarrow workset \cup \tilde{C}'$
            end if
        end if
    until $workset = \emptyset$
    return $finalset$
end function
```

Figure 8 An algorithm for abstract execution.

and $\text{STM}(T_2, pc_{T_2} + 1) = [\texttt{store}\ r'\ \texttt{to}\ x]^{pc_{T_2}+1}$. When a transition occurs, the `load`- and `skip`-statements are considered. However, if the execution time of the `store`-statement (the abstract "point" in time when the thread's pc is updated) overlaps with the execution time of the `load`-statement, the resulting value of r in T_1 should be affected by the value of r' in T_2, but this will not be the case. A similar reasoning holds for `lock`- and `unlock`-statements.

3.4 Analysis by Abstract Execution

Since the abstract transition rule, $\xrightarrow[prg]{\sim}$, of Fig. 7 is not safe, one cannot simply use fixpoint-iterations [4, 10] on the abstract semantic rules to find a safe approximation to the concrete program semantics. Instead, a worklist algorithm will be defined that uses $\xrightarrow[prg]{\sim}$ in a safe way and handles the unsafe cases explicitly. The function ABSTRACTEXECUTION in Fig. 8 defines such an algorithm; the '@' symbol is used for denoting two ways of expressing the same thing (c.f., the "read as" operator in Haskell). Given a configuration, $\tilde{c}$, and a timeout, $\tilde{t}_{to}$, the function explores all the possible abstract transitions, until only final (all threads are standing on a `halt`-statement) and timed-out (all threads will update their pc:s at a point in time succeeding $\tilde{t}_{to}$) configurations remain. The function returns a set containing all the final and timed-out configurations. If a configuration is not final or timed-out, a transition will be performed. The threads executing `load`-statements are extracted and handled separately.

```
thread T_1:                thread T_2:                thread T_3:
[load r from x]1;[1,5]     [load r from y]1;[1,6]     [if r<=3 goto 3]1;[1,3]
[store r to y]2;[1,3]      [store r to z]2;[2,3]      [store r to x]2;[2,3]
[halt]3                    [halt]3                    [halt]3
```

Figure 9 Example program.

This is done by recursively using ABSTRACTEXECUTION for each such thread to simulate how the rest of the threads in the configuration can affect the read value. When the effects have been derived, they are merged and put in the target register for the thread that issues the `load`-statement. Next, a new configuration, in which the `load`:s have been performed, is added to the worklist. Note that TRIM is used to remove parts of the history from $\tilde{\mathtt{x}}$ that cannot affect a `load`-statement in any thread at time $\tilde{t}$. This is to lower the space complexity of ABSTRACTEXECUTION. Further details on the algorithm, definitions of the used functions and correctness proofs can be found in [8]. Note that this algorithm cannot safely analyse programs acting on locks. The algorithm will be extended with this ability (see Section 5).

Assuming that ABSTRACTEXECUTION has been applied to some $\tilde{c}$ and that $\tilde{t}_{to} = [0, \infty]$, safe bounds on the corresponding concrete BCET and WCET can be extracted from the resulting set of configurations (details can be found in [8]).

4 Example

In this section, the program in Fig. 9 is analysed (the results of ABSFINTIME are given after the non-`halt`-statements). Initially, let $\tilde{c} = \langle\{(T_1, 1, \tilde{\mathtt{r}}_{T_1}, [0,0], [0,0]), (T_2, 1, \tilde{\mathtt{r}}_{T_2}, [0,0], [0,0]), (T_3, 1, \tilde{\mathtt{r}}_{T_3}, [0,0], [0,0])\}, \tilde{\mathtt{x}}, \mathbb{I}, [0,0]\rangle$, where $\tilde{\mathtt{r}}_{T_3}\ \mathtt{r} = [2,4]$, $((\tilde{\mathtt{x}}\ \mathtt{x})\ T_2) = ((\tilde{\mathtt{x}}\ \mathtt{x})\ T_3) = \emptyset$ and $((\tilde{\mathtt{x}}\ \mathtt{x})\ T_1) = \{([1,1],[0,0])\}$, $((\tilde{\mathtt{x}}\ \mathtt{y})\ T_1) = ((\tilde{\mathtt{x}}\ \mathtt{y})\ T_2) = \emptyset$ and $((\tilde{\mathtt{x}}\ \mathtt{y})\ T_3) = \{([5,5],[0,0])\}$, and $((\tilde{\mathtt{x}}\ \mathtt{z})\ T_2) = \emptyset$, is analysed. ABSTRACTEXECUTION$(\tilde{c}, [0, \infty])$ is summarised in Fig. 10.

The tuples in the chart represent program points, defined as $\langle pc_{T_1}, pc_{T_2}, pc_{T_3}\rangle$. As can be seen, for $\langle 1,1,1\rangle$, T_1 and T_2 both execute a `load`-statement. This means that two new instances of ABSTRACTEXECUTION are created, one for each thread in $\mathbf{Thrd}_{\mathtt{load}}$. Within each of these instances, a new instance is created since one other thread also executes a `load`-statement. A '_' within the tuple indicates that the corresponding thread is removed from the configuration to evaluate the effects it might see. Next to each tuple and transition arrow, there is a comment stating what happens at the corresponding step. The found bounds on the BCET and WCET are 3 and 9, respectively. Note that ABSFINTIME is assumed to be defined somewhere outside the scope of this paper. Also note that programs containing loops can be analysed, but due to space reasons, this is not illustrated here.

5 Discussion & Future Work

The algorithm in Fig. 8 is based on synchronously advancing the threads of a program between their respective program points. This, together with the defined abstract domain for variables, has the advantage that the analysis result will be the same as for the sequential case [6], when $P = T$. Another advantage is that the complexity of the algorithm becomes more dependent on the number of program points than on the timing behaviour of the program. To further reduce the time complexity of the algorithm, merging of configurations could be performed. Using the control flow graph (CFG) of the program, suitable merge-points within each thread can be found [5]. Typically, such points have multiple incoming edges.

A drawback for the algorithm in Fig. 8 is that termination is not guaranteed if a program consists of *infinite* loops. This could be resolved by adjusting the initial timeout, though.

Our current focus is to extend the algorithm to support programs using locks and then to

```
ABSTRACTEXECUTION($\tilde{c}, [0, \infty]$)
$\langle 1,1,1\rangle$   $\mathbf{Thrd}_{\mathtt{load}} = \{T_1, T_2\}, \tilde{t}^r_{T_1} = [1,5], \tilde{t}^r_{T_2} = [1,6], \tilde{t}^r_{T_3} = [1,3]$
    ABSTRACTEXECUTION($\tilde{c}_1, [1,5]$)
    $\langle \_,1,1\rangle$   $\mathbf{Thrd}_{\mathtt{load}} = \{T_2\}, \tilde{t}^r_{T_2} = [1,6], \tilde{t}^r_{T_3} = [1,3]$
        ABSTRACTEXECUTION($\tilde{c}_{1.2}, [1,5]$)
        $\langle \_,\_,1\rangle$   $\mathbf{Thrd}_{\mathtt{load}} = \emptyset$
        $\swarrow \downarrow$   $\tilde{\mathbb{r}}_{T_3}\ \mathtt{r} \leftarrow [2,3]$   $(\widetilde{\mathcal{BR}}[\![\mathtt{r <= 3}]\!]\tilde{\mathbb{r}}_{T_3} \neq \tilde{\bot}_{reg})$
        $\downarrow \langle \_,\_,3\rangle$   final $(\tilde{t}^a_{T_3} = [1,3])$, no effects
        $\searrow$   $\tilde{\mathbb{r}}_{T_3}\ \mathtt{r} \leftarrow [4,4], \tilde{t}^a_{T_3} \leftarrow [1,3]$   $(\widetilde{\mathcal{BR}}[\![\mathtt{!(r <= 3)}]\!]\tilde{\mathbb{r}}_{T_3} \neq \tilde{\bot}_{reg})$
        $\langle \_,\_,2\rangle$   $\mathbf{Thrd}_{\mathtt{load}} = \emptyset, \tilde{t}^r_{T_3} = [2,3]$
        $\downarrow$   $((\tilde{\mathbb{x}}\ \mathtt{x})\ T_3) \leftarrow \{([4,4],[3,6])\}$
        $\langle \_,\_,3\rangle$   final $(\tilde{t}^a_{T_3} = [3,6])$, $\mathtt{x}$ affected
    $\downarrow$   $\tilde{\mathbb{r}}_{T_2}\ \mathtt{r} \leftarrow [5,5]$ (no effects on $\mathtt{y}$ from $T_3$), $\tilde{t}^a_{T_2} \leftarrow [1,6]$
    $\langle \_,2,1\rangle$   $\mathbf{Thrd}_{\mathtt{load}} = \emptyset, \tilde{t}^r_{T_2} = [2,3], \tilde{t}^r_{T_3} = [1,3]$
    $\swarrow \downarrow$   $\tilde{\mathbb{r}}_{T_3}\ \mathtt{r} \leftarrow [2,3], ((\tilde{\mathbb{x}}\ \mathtt{z})\ T_2) \leftarrow \{([5,5],[3,9])\}$,
    $\downarrow \langle \_,3,3\rangle$   final $(\tilde{t}^a_{T_2} = [3,9], \tilde{t}^a_{T_3} = [1,3])$, $\mathtt{z}$ affected
    $\searrow$   $\tilde{\mathbb{r}}_{T_3}\ \mathtt{r} \leftarrow [4,4], ((\tilde{\mathbb{x}}\ \mathtt{z})\ T_2) \leftarrow \{([5,5],[3,9])\}, \tilde{t}^a_{T_2} \leftarrow [3,9], \tilde{t}^a_{T_3} \leftarrow [1,3]$
    $\langle \_,3,2\rangle$   $\mathbf{Thrd}_{\mathtt{load}} = \emptyset, \tilde{t}^r_{T_3} = [2,3]$
    $\downarrow$   $((\tilde{\mathbb{x}}\ \mathtt{x})\ T_3) \leftarrow \{([4,4],[3,6])\}$
    $\langle \_,3,3\rangle$   final $(\tilde{t}^a_{T_2} = [3,9], \tilde{t}^a_{T_3} = [3,6])$, $\mathtt{x}$ and $\mathtt{z}$ affected
    ABSTRACTEXECUTION($\tilde{c}_2, [1,6]$)
    $\langle 1,\_,1\rangle$   $\mathbf{Thrd}_{\mathtt{load}} = \{T_1\}, \tilde{t}^r_{T_1} = [1,5], \tilde{t}^r_{T_3} = [1,3]$
        ABSTRACTEXECUTION($\tilde{c}_{2.1}, [1,5]$)
        $\langle \_,\_,1\rangle$   $\mathbf{Thrd}_{\mathtt{load}} = \emptyset$
        $\swarrow \downarrow$   $\tilde{\mathbb{r}}_{T_3}\ \mathtt{r} \leftarrow [2,3]$
        $\downarrow \langle \_,\_,3\rangle$   final $(\tilde{t}^a_{T_3} = [1,3])$, no effects
        $\searrow$   $\tilde{\mathbb{r}}_{T_3}\ \mathtt{r} \leftarrow [4,4], \tilde{t}^a_{T_3} \leftarrow [1,3]$
        $\langle \_,\_,2\rangle$   $\mathbf{Thrd}_{\mathtt{load}} = \emptyset, \tilde{t}^r_{T_3} = [2,3]$
        $\downarrow$   $((\tilde{\mathbb{x}}\ \mathtt{x})\ T_3) = \{([4,4],[3,6])\}$
        $\langle \_,\_,3\rangle$   final $(\tilde{t}^a_{T_3} = [3,6])$, $\mathtt{x}$ affected
    $\downarrow$   $\tilde{\mathbb{r}}_{T_1}\ \mathtt{r} \leftarrow [1,4]$ (effects on $\mathtt{x}$ from $T_3$), $\tilde{t}^a_{T_1} \leftarrow [1,5]$
    $\langle 2,\_,1\rangle$   $\mathbf{Thrd}_{\mathtt{load}} = \emptyset, \tilde{t}^r_{T_1} = [1,3], \tilde{t}^r_{T_3} = [1,3]$
    $\swarrow \downarrow$   $\tilde{\mathbb{r}}_{T_3}\ \mathtt{r} \leftarrow [2,3], ((\tilde{\mathbb{x}}\ \mathtt{y})\ T_1) \leftarrow \{([1,4],[2,8])\}$,
    $\downarrow \langle 3,\_,3\rangle$   final $(\tilde{t}^a_{T_1} = [2,8], \tilde{t}^a_{T_3} = [1,3])$, $\mathtt{y}$ affected
    $\searrow$   $\tilde{\mathbb{r}}_{T_3}\ \mathtt{r} \leftarrow [4,4], ((\tilde{\mathbb{x}}\ \mathtt{y})\ T_1) \leftarrow \{([1,4],[2,8])\}, \tilde{t}^a_{T_1} \leftarrow [2,8], \tilde{t}^a_{T_3} \leftarrow [1,3]$
    $\langle 3,\_,2\rangle$   $\mathbf{Thrd}_{\mathtt{load}} = \emptyset, \tilde{t}^r_{T_3} = [2,3]$
    $\downarrow$   $((\tilde{\mathbb{x}}\ \mathtt{x})\ T_3) \leftarrow \{([4,4],[3,6])\}$
    $\langle 3,\_,3\rangle$   final $(\tilde{t}^a_{T_1} = [2,8], \tilde{t}^a_{T_3} = [3,6])$, $\mathtt{x}$ and $\mathtt{y}$ affected
$\downarrow$   ($T_1$ sees effects on $\mathtt{x}$ and $\mathtt{z}$, and $T_2$ sees effects on $\mathtt{x}$ and $\mathtt{y}$.)
$\downarrow$   $\tilde{\mathbb{r}}_{T_1}\ \mathtt{r} \leftarrow [1,4], \tilde{\mathbb{r}}_{T_2}\ \mathtt{r} \leftarrow [1,5], \tilde{t}^a_{T_1} \leftarrow [1,5], \tilde{t}^a_{T_2} \leftarrow [1,6]$
$\langle 2,2,1\rangle$   $\mathbf{Thrd}_{\mathtt{load}} = \emptyset, \tilde{t}^r_{T_1} = [1,3], \tilde{t}^r_{T_2} = [2,3], \tilde{t}^r_{T_3} = [1,3]$
$\swarrow \downarrow$   $\tilde{\mathbb{r}}_{T_3}\ \mathtt{r} \leftarrow [2,3], ((\tilde{\mathbb{x}}\ \mathtt{y})\ T_1) \leftarrow \{([1,4],[2,8])\}, ((\tilde{\mathbb{x}}\ \mathtt{z})\ T_2) \leftarrow \{([1,5],[3,9])\}$
$\downarrow \langle 3,3,3\rangle$   final $(\tilde{t}^a_{T_1} = [2,8], \tilde{t}^a_{T_2} = [3,9], \tilde{t}^a_{T_3} = [1,3])$, $\mathtt{y}$ and $\mathtt{z}$ affected
$\searrow$   $\tilde{\mathbb{r}}_{T_3}\ \mathtt{r} \leftarrow [4,4], ((\tilde{\mathbb{x}}\ \mathtt{y})\ T_1) \leftarrow \{([1,4],[2,8])\}, ((\tilde{\mathbb{x}}\ \mathtt{z})\ T_2) \leftarrow \{([1,5],[3,9])\}$
$\langle 3,3,2\rangle$   $\mathbf{Thrd}_{\mathtt{load}} = \emptyset, \tilde{t}^r_{T_3} = [2,3]$
$\downarrow$   $((\tilde{\mathbb{x}}\ \mathtt{x})\ T_3) \leftarrow \{([4,4],[3,6])\}$
$\langle 3,3,3\rangle$   final $(\tilde{t}^a_{T_1} = [2,8], \tilde{t}^a_{T_2} = [3,9], \tilde{t}^a_{T_3} = [3,6])$, $\mathtt{x}$, $\mathtt{y}$ and $\mathtt{z}$ affected
```

Figure 10 The steps taken by ABSTRACTEXECUTION when analysing the program in Fig. 9.

implement and evaluate it. Allowing the use of locks introduces a risk for deadlocks (both in the analysed program and thus the algorithm). However, deadlocks could easily be detected and handled by the algorithm, because all threads, not standing on a `halt`-statement, would be waiting to acquire a lock that is locked and not owned by themselves. Thus, this detection allows termination of the analysis (with a resulting WCET of ∞) even if deadlocks occur.

References

1 Sudipta Chattopadhyay, C.-L. Kee, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. A unified WCET analysis framework for multi-core platforms. In *18th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS'12)*, Beijing, China, April 2012.

2 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, January 1977.

3 Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Deriving WCET bounds by abstract execution. In Chris Healy, editor, *Proc. 11th International Workshop on Worst-Case Execution Time Analysis (WCET'2011)*, Porto, Portugal, July 2011.

4 Jan Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation.* PhD thesis, Dept. of Information Technology, Uppsala University, Sweden, May 2000.

5 Jan Gustafsson and Andreas Ermedahl. Merging techniques for faster derivation of WCET flow information using abstract execution. In Raimund Kirner, editor, *Proc. 8th International Workshop on Worst-Case Execution Time Analysis (WCET'2008)*, Prague, Czech Republic, July 2008.

6 Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Proc. 27th IEEE Real-Time Systems Symposium (RTSS'06)*, pages 57–66, Rio de Janeiro, Brazil, December 2006. IEEE Computer Society.

7 Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET analysis of multicore architectures using UPPAAL. In Björn Lisper, editor, *Proc. 10th International Workshop on Worst-Case Execution Time Analysis (WCET'2010)*, pages 103–113, Brussels, Belgium, July 2010. OCG.

8 Andreas Gustavsson, Jan Gustafsson, and Björn Lisper. Toward static timing analysis of parallel systems – technical report. Technical Report 2796, Dept. of Computer Science and Engineering, Mälardalen University, April 2012.
URL: `http://www.mrtc.mdh.se/index.php?choice=publications&id=2796`.

9 Mingsong Lv, Nan Guan, Wang Yi, and Ge Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In Scott Brandt, editor, *Proc. 31th IEEE Real-Time Systems Symposium (RTSS'10)*, pages 339–349, San Diego, CA, December 2010. IEEE.

10 Flemming Nielson, Hanne Ries Nielson, and Chris Hankin. *Principles of Program Analysis, 2nd edition.* Springer, 2005. ISBN 3-540-65410-0.

11 Christine Rochange, Armelle Bonenfant, Pascal Sainrat, Mike Gerdes, Julian Wolf, Theo Ungerer, Zlatko Petrov, and Frantisek Mikulu. WCET analysis of a parallel 3D multigrid solver executed on the MERASA multi-core. In Björn Lisper, editor, *Proc. 10th International Workshop on Worst-Case Execution Time Analysis (WCET'2010)*, pages 90–100, Brussels, Belgium, July 2010. OCG.

12 Jun Yan and Wei Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In *Proc. 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'08)*, pages 80–89, June 2008.

Towards Parallel Programming Models for Predictability

Björn Lisper

School of Innovation, Design and Engineering, Mälardalen University
Box 883, S–721 23 Västerås, Sweden.
bjorn.lisper@mdh.se

Abstract

Future embedded systems for performance-demanding applications will be massively parallel. High performance tasks will be parallel programs, running on several cores, rather than single threads running on single cores. For hard real-time applications, WCETs for such tasks must be bounded. Low-level parallel programming models, based on concurrent threads, are notoriously hard to use due to their inherent nondeterminism. Therefore the parallel processing community has long considered high-level parallel programming models, which restrict the low-level models to regain determinism. In this position paper we argue that such parallel programming models are beneficial also for WCET analysis of parallel programs. We review some proposed models, and discuss their influence on timing predictability. In particular we identify data parallel programming as a suitable paradigm as it is deterministic and allows current methods for WCET analysis to be extended to parallel code. GPUs are increasingly used for high performance applications: we discuss a current GPU architecture, and we argue that it offers a parallel platform for compute-intensive applications for which it seems possible to construct precise timing models. Thus, a promising route for the future is to develop WCET analyses for data-parallel software running on GPUs.

1998 ACM Subject Classification C.3 Special-Purpose and Application-Based Systems

Keywords and phrases Real-Time System, WCET analysis, Parallel Program, Data Parallelism

Digital Object Identifier 10.4230/OASIcs.WCET.2012.48

1 Introduction

There is an ever growing need for advanced functionality in embedded systems. As hardware becomes more capable, new intelligent applications will emerge: some possible examples are real-time image processing and object recognition, multiple-sensor information fusion, and online spectral analysis for state-based maintenance. Some of these applications will have hard real-time constraints, and these constraints will then have to be verified. This requires a WCET analysis for the involved tasks.

High performance hardware is today invariably *parallel*, as this is the only way to meet the demands for computational power. A high performance task will also have to be a parallel program, thus utilizing the available computational resources. This raises the issue how to estimate the WCET of a parallel program, consisting of a number of cooperating threads synchronizing and exchanging data in some manner.

WCET analysis of parallel programs has hardly been studied up to now. Preliminary work on such a WCET analysis, for a simple language with parallel threads communicating through a shared memory and synchronizing with locks, is reported in [8]. A conclusion to be drawn from this work is that while it certainly is possible to define such an analysis, and prove its soundness, there will be issues to handle. One issue is that, since the functional semantics of the program now may be dependent on the timing, due to race conditions, it is no more possible to divide the analysis into a distinct program flow analysis, low-level analysis, and final calculation: the calculation has to be intertwined with the other phases to find the possible outcomes of race conditions, which may in turn affect the possible orders of writes, or acquirements of locks. Another issue is that the potentially many possible executions of such a program, especially in the presence of potential race conditions, may make the precision of the analysis sensitive to approximations. An overapproximated timing interval for a write, for instance, may introduce a "false" order of writes in the analysis that may in turn affect the estimated value of, say, a loop counter. This can in turn yield a very poor precision for the resulting WCET bound.

Thus, it is interesting to investigate programming models for parallel computers that avoid the nondeterminism of low-level, thread-based models, and that provide a more structured way to write parallel programs for high performance applications. Such models have been studied in the parallel computing community for the last 25 years, for two reasons: they provide easier and less error-prone programming, and they yield the possibility to attach abstract cost (performance) models to the parallel programming constructs. Both properties should be of interest also for embedded real-time software, in particular if the cost models can be sharpened to safe timing models.

In this position paper we argue that some of these parallel programming models provide a suitable parallel programming discipline for time-critical applications as well. In particular, we review the *data parallel* model, and we show how it restricts the patterns of parallelism in such a way that a WCET analysis can be done in a more or less traditional fashion if timing models are provided for the parallel constructs in this model. We identify for which constructs it may be problematic to find tight timing bounds – methods to find good bounds for them is a target for future research. We conclude by indicating how the data parallel constructs can be mapped to the CUDA Graphics Processing Unit (GPU) architecture.

The rest of the paper is organized as follows: in Section 2 we review the background: present and expected future development in embedded parallel computing, and the problems with unrestricted thread- and process based parallelism. Section 3 introduces the *Bulk Synchronous Parallel* model [21], which allows cost models that are valid over a wide range of parallel architectures, and in Section 4 we review the data parallel model and discuss timing models for the typical data parallel primitives. Section 5 introduces GPUs, and NVIDIA's CUDA model, and in Section 6 we discuss how the data parallel primitives can be mapped to CUDA. Section 7 provides a discussion of related work. In Section 8, finally, we conclude the paper and present future work.

2 Background

At this moment, multi-core processors are conquering the world. Current multi-cores have a traditional shared-memory architecture with a few cores sharing a common memory, connected via a shared bus. This architecture works reasonably well performance-wise for concurrent, independent tasks, or moderately parallel computing, but it does not scale since the shared bus and memory quickly become bottlenecks. As we go for higher performance, the number of cores will have to increase, distributed memory must be provided, and the bus must be replaced by a Network-on-Chip (NoC) that provides a wider communication channel. Tilera already offers general purpose many-cores with up to 100 cores, a distributed cache, and a mesh network[1].

[1] `http://www.tilera.com/products/processors`

Another strand of development is provided by GPUs. Originally designed to accelerate graphics, they are now rapidly turning into parallel general-purpose computational co-processors. We can expect them to be integrated on chip. Thus, a future processor architecture may have one part that is a conventional multi-core, with a few, relatively powerful processors, and a powerful computational co-processor with many relatively simple cores, connected through a NoC, and with local memories at different levels.

Now consider a time-critical, computationally intensive application, implemented by a parallel program, whose timing constraints must be verified. How can we obtain safe WCET estimates? Current practice in high performance computing it to write parallel software with explicit threads or processes, either using a shared memory, synchronizing through locks or semaphores (Pthreads [1], OpenMP [4]) or distributed memory, with synchronization and communication effectuated through message-passing (MPI [6]). Unfortunately such low-level parallel programming models are inherently nondeterministic, and race conditions create dependencies between timing and functional behaviour. This makes WCET analysis hard. Consider, for instance, the following example with parallel threads `T1` and `T2`, sharing the variable `n`:

```
T1: p; n = 100000000; halt
T2; q; n = 10; for i = 1 to n do r; halt
```

Here, there is a potential race condition for the assignment to `n`. If we know that $\mathit{WCET}(\mathtt{p}) = 150$ and $\mathit{BCET}(\mathtt{q}) = 200$, however, then `T2` will always “win” and the loop will always iterate ten times. But if an imprecise timing analysis yields a WCET bound for `p` that is greater than the BCET bound for `q`, then the analysis must assume that the loop can iterate 10^8 times. This example demonstrates that it is no more possible to perform a loop bounds analysis separately from the timing bounds calculation.

If we add synchronization through locks, then an imprecise analysis might even falsely detect deadlocks. If $\mathit{BCET}(\mathtt{p}) > \mathit{WCET}(\mathtt{q})$ below then no deadlock is possible, but again an imprecise timing analysis might fail to verify this and must then assume that a deadlock can occur.

```
T3: p; lock l; halt
T4; q; lock l; r; unlock l; halt
```

Also an imprecise program flow analysis might yield similar problems, like false deadlocks. Consider the two threads below, sharing a barrier `b`:

```
T5: m = expr1; for i = 1 to n do { a; barrier(b); c }
T6; m = expr2; for i = 1 to m do { x; barrier(b); y }
```

Assume that `expr1` amd `expr2` are such that they always evaluate to the same value. However, a value analysis may fail to detect this: a WCET analysis using the results from that value analysis for its loop bounds analysis will then have to assume that there is a possibility that one thread will attempt to execute its `barrier` statement one time more than the other one which then would cause it to deadlock. Thus, the analysis will have to assume that a deadlock might appear. (In fact, any loop bounds analysis that only detects *upper* loop bounds will not be strong enough to prove absence of deadlock for this example.)

These examples may be a bit contrived, but they are constructed to make a point. In reality, the kind of problems exemplified above may appear in much more subtle ways, in seemingly unproblematic code. The conclusion is that timing analysis of parallel software will be difficult as long as the programs are written on this level. To alleviate the problem, more disciplined parallel programming practices are needed that eliminate race conditions and the risk of deadlock.

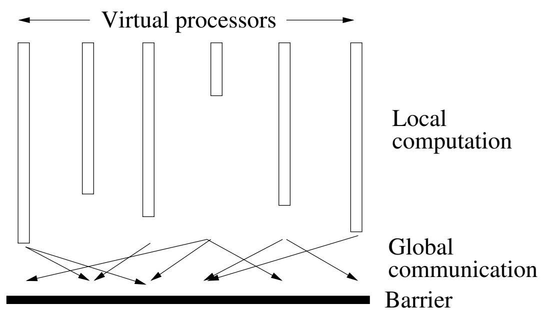


Figure 1 A BSP superstep (following [20]).

3 Bulk Synchronous Programming

Bulk Synchronous Programming (BSP) [20, 21] is a model for parallel computing allowing some performance analyzability over a wide range of target parallel architectures. The BSP model assumes a hardware model of *components* that are connected by a *router* (a network). The components can be processors or memories, and part of the router can connect single processors locally with memories: thus, the model covers both distributed memory architectures as well as shared (global) memory architectures, possibly with the memory structured into different memory banks.

BSP has a concept of *virtual processors*, where a physical processor in each computation step will execute the instructions of several virtual processors. In each computation step, the virtual processors execute independent tasks. To hide communication latencies it is assumed that there are considerably more virtual processors than actual processors, i.e., that the computation is massively parallel.

A BSP program executes in *supersteps*. In each superstep each component performs a task consisting of an initial phase of local computation steps, followed by a communication phase, followed by a final barrier synchronization. See Fig. 1.

The separation of computation, communication, and synchronization has some interesting consequences for the performance analyzability. For the computation phase, it is easy to estimate the execution time since the processors execute in an entirely local fashion. Similarly, the execution time for barrier synchronization should be easy to bound. Also note that the restriction to global barrier synchronization eliminates the risk of deadlock provided that all local supersteps terminate.

The communication phase is the hardest to analyze, since its performance is very dependent on the communication/access pattern, network topology, and distribution of processors and memories. The BSP model assumes that the router has a *permeability* g such that if each component (memory or processor) sends or receives at most h messages within a superstep, then the time for the communication phase is $O(gh)$. This assumption is easily violated in real parallel systems due to hotspots and similar. But assuming random (or hashed) distribution of data and computations such hotspots will be unlikely for a wide range of network topologies, and the permeability model can be applied to average case complexity calculations.

For WCET analysis it is clear that the separation into supersteps, with separate phases for computation, communication, and synchronization, should be beneficial. If the time for a superstep can be bounded, as well as the number of supersteps, then a coarse WCET estimate can be found by multiplying these bounds. The most serious remaining hurdle is

Elementwise application	`for all k in parallel do B[k] = f(A1[k],...,An[k])`
Get communication	`for all k in parallel do X[k] = Y[G[k]]`
Send communication	`for all k in parallel do X[G[k]] = Y[k]`
Replication	`for all k in parallel do X[k] = Y`
Masking	`for all k where B[k] in parallel do X[k] = ...k...`
Reduce	reduce(op, X) = X[0] op ⋯ op X[n − 1]
Scan	scan(op, X) = [X[0], X[0] op X[1], . . . , X[0] op ⋯ op X[n − 1]]

Figure 2 Data parallel operations.

the communication phase, which will require a much more detailed analysis for bounding the worst case tightly. Also, BSP does not preclude race conditions in this phase. Thus, parallel programming models that provide more predictable and deterministic communication primitives can be desirable from a timing analysis point of view.

4 The Data Parallel Programming Model

Data Parallel Programming is an instance of *Collection-Oriented Programming* [19], with roots in APL [10]. Collection-oriented programming emphasizes programming with homogenous data structures, such as arrays, lists, or sets. The core of the paradigm is a set of operations that work directly on such structures rather than on the elements one by one. This has two consequences: first, many loops can be replaced by such operations, rendering the programs more succinct and easier to understand, and second these primitives often have highly parallel implementations, with large numbers of small, independent computations. Thus, collection-oriented languages are interesting candidates for programming parallel hardware.

Data parallel languages take the collection-oriented approach one step further by associating each element in a data structure (often an array) with one virtual processor. The implementation will then map the virtual processors to physical processors, creating a partition of the data structure, and even effectuating a physical distribution in the case of a distributed memory machine.

The data parallel primitives can be classified in six groups: *elementwise application* applies a “scalar” function to each element in a data structure; *get communication* creates a new data structure by reading (pulling) each element from a source location in another data structure; *send communication* does likewise, but by *sending* (pushing) each element to a destination; *replication* creates a data structure whose elements are copies of a “scalar” value; *masking* will select part of a data structure for computation, using a boolean mask; *reduce*, and *scan*, finally, are generalized “sums” (or arrays of partial sums) where a binary, associative operator is successively applied to all elements in a data structure. See Fig. 2 for an informal definition, using a “`for all` . . . `in parallel do`” syntax ranging over all virtual processors of the data structures.

If the elementwise applied functions are side effect-free, then *all data parallel operations except send communication are deterministic.* (For send communication there are write conflicts if `G[k]` = `G[k']` for some `k` ≠ `k'`: however, if `G` specifies a permutation then this operation is also deterministic.) Thus, *a deterministic language that is extended with data parallel operations is still deterministic* (with the small proviso above).

It follows that a sequential language extended with data parallel primitives allows a conventional WCET analysis, with separate flow- and low-level analysis and calculation phases, provided that the execution times for the data parallel operations can be bounded.

The timing bounds for the data parallel operations will depend strongly on how well the pattern of communication for an operation matches the communication capabilities of the network. This pattern in turn depends strongly on how the memory is organized, and how data is allocated in memory. Assume there are n virtual processors being executed on k physical processors. If communication can be done in unit time, then the execution time for elementwise application is $O(n/k)$ since the individual applications can be done in any order, on any processor. This bound will hold in particular if the data accessed by each virtual processor already resides in local memory for its physical processor. Under similar conditions reduce and scan can be done in $O(n/k + \log k)$ time, given that the network provides efficient support for tree-balanced summation. Masking will not add more than a small constant cost, and can be implemented for instance with predicated instructions. The remaining operations all concern general communication, for which a more detailed analysis of the actual access and communication patterns will be needed to bound the execution time reasonably well. However data parallel languages tend to offer specialized communication primitives, like array shifts, which provide efficient and predictable communication on a wide range of parallel hardware. For such primitives it will be easier to find tight timing bounds, and a parallel program that can use them instead of the general communication operations will likely be more timing predictable.

As for BSP, the strict division into communication, computation, and synchronization helps formulating the timing models. The data parallel paradigm has the additional advantage that it provides a set of distinct deterministic parallel operations: this helps timing analysis since timing bounds for these operations can be used directly in a conventional WCET calculation.

5 GPUs: the CUDA Model

GPUs have rapidly emerged as computational co-processors that provide high performance computing capabilities at low cost. A GPU is typically designed to run many similar, independent threads at high speed, using massive parallelism (up to hundreds of cores). It will have several levels of local memory, distinct from the host memory. Data is moved from host to local memory, and between different levels of local memory, by explicit operations. The host will issue jobs, consisting of a number of threads to execute to completion, to the GPU. The jobs can either block the host until completion, or allow that the host continues to execute concurrently.

Two rather similar application programming interfaces have emerged for GPUs: OpenCL, and NVIDIA's CUDA [14]. Both also provide a computation model. We now review CUDA and we also discuss briefly the current underlying execution model for NVIDIA GPUs.

The basic units of execeution in CUDA are threads. The threads are grouped into *thread blocks*. A multithreaded CUDA program logically specifies a number of thread blocks: these are then scheduled in some unspecified, possibly parallel order on the GPU cores, where the execution of a thread block is done by executing its threads on a single core in some unspecified, possibly interleaved order. If there are no dependencies between the threads in a set of thread blocks, then the semantics will be independent of the order in which they are executed. This yields considerable flexibility in how thread blocks are executed, and the execution can easily adapt to the number of available cores in the GPU.

A thread within a block has a local unique ID within that block, and the block itself has a global unique ID: both can be accessed by the thread, and thus it can decide, e.g., which part of an array that it will work on.

There is no synchronization available between thread blocks. Threads within a block can, however, synchronize through local barriers.

CUDA has a memory hierarchy, with memory at different levels of locality. Each thread has its own private memory. Each thread block has shared memory, with the same lifetime as the block, which is visible to all the threads in the block: variables residing in this memory are explicitly declared. There is also global, shared memory visible in all thread blocks. Local shared memory can be expected to be much faster than global shared memory. Furthermore, the CUDA model assumes that the GPU executes as co-processor to a host, and that their memories are distinct. Transfers between host and co-processor memory are done explicitly, through calls to the CUDA runtime system. A thread executing on the co-processor can only access memory that is located on that processor: data to be processed thus has to be copied from the host to the co-processor, and results must be copied back.

CUDA C extends C with *kernels*. These are C functions for which, when called, N instances are executed in parallel in N different CUDA threads. A special call to the kernel will issue the execution of the threads on the GPU.

A CUDA C program will alternatingly execute in sequential and parallel phases. In a sequential phase, the C program is executing as usual on the host. A call to a kernel invokes a parallel phase where the kernel is executed in parallel, in a number of thread blocks, on the co-processor (the GPU). When the kernel call returns, the sequential execution will resume.

Under some circumstances the co-processor can execute concurrently with the host. Control is then returned to the host before the call has completed. This includes asynchronous kernel calls, and certain memory transfers.

NVIDIA's underlying hardware architecture is called SIMT (Single-Instruction, Multiple-Thread), and it can be seen as a hybrid of SIMD and MIMD with separate threads but strong coherence locally between their execution. Threads are executed on a scalable array of multithreaded *Streaming Multiprocessors* (SMs). The execution of threads is organized in groups of 32 threads called *warps*. A number of warps will make up a thread block. All threads in a warp will execute on the same SM: they will start at the same address, but they have their own program counters. Threads in a warp can therefore take different execution paths through the code. However, different threads in a warp can execute simultaneously only if they execute the same instruction: thus, diverging threads will force the warp to execute the instructions on each path separately, disabling threads that are not on that path. This can affect the performance very adversely. Instructions can be predicated, though.

The warps are scheduled by a *warp scheduler*. This scheduler will always pick a warp with an instruction ready to execute, if available. Since there typically will be considerably more warps than cores to execute them on, this will help mask the memory latencies as it allows memory fetches issued by instructions to complete in parallel with instructions for other warps being executed.

The current SM cores have a fairly simple architecture: instructions are pipelined, but they are executed in order and there is no branch prediction or speculative execution. Each SM has a set of registers, and local memory partitioned among the thread blocks: on some GPU models this is a partitioned cache, whereas for others it is explicitly addressed. A GPU also has global memory that is shared among the SMs.

The conclusion is that overall, the NVIDIA architecture with the CUDA model should be quite amenable to timing analysis. The processors are simple. Memory access times vary strongly with the degree of locality, but for the most part it is explicit which part of memory is being accessed. GPU models with local caches will need a cache analysis, though. Although threads are executed concurrently, it is done in a very controlled fashion. Also the interaction with the host is very explicit, with calls for submitting kernels and transferring memory contents. Overlaps in execution between GPU and host should be possible to estimate. Finally, since the GPU executes requests in sequence there will not be the same competition for shared resources within the GPU, between unrelated activities, as in a bus-based multi-core processor.

The largest sources of uncertainty seem to be (1) the fact that a warp can only execute threads executing the same instruction in parallel, and (2) uncertainty about exactly how the warps are scheduled. (1) will be alleviated if the executed kernels have a predictable, indata-independent program flow: if not, then an analysis may have to assume that all 32 instances of the kernel in a warp will diverge, and thus must have their respective instructions sequentially executed in each step. As for (2), uncertainties about warp scheduling means that it will be harder to estimate the overlaps between warp execution and memory fetches.

6 Mapping Data Parallel Primitives to GPUs

It should be evident from the preceding discussion that data parallel programming and GPUs is a good match. GPUs are designed to execute a large number of similar threads, executing in the same fashion with little or no synchronization, in parallel. The data parallel operations described in Section 4 can be implemented by such threads. For elementwise applied operations this is obvious. Parallel read operations, and replication, can also be implemented by threads reading data to local shared memory, and parallel writes will copy data in parallel from local memories to global shared memory. Masking can be implemented efficiently by predicated execution of instructions. A compiler can coalesce sequences of such data parallel operations into kernel code implementing them on the GPU: these kernels will execute efficiently and predictably since all threads will execute the same path. Operations like reduce and scan, finally, also have efficient GPU implementations [18]. Thus, data parallel languages implemented on GPUs should provide an interesting alternative for implementing high performance applications with stringent timing constraints.

7 Related Work

Most work regarding WCET analysis for parallel systems concern single-threaded programs running on single cores in multi-cores. Here, the challenge is to find tight WCET bounds when different activities can compete for shared resources like buses and memories [2, 3]. A common assumption is that the system bus uses TDMA to provide predicable bandwidth for the different cores [17]. Various cache analyses have also been considered for multi-cores [9, 11, 12].

Current multi-core architectures have poor timing predicability due to the presence of shared resources under loose control. Principles for designing timing-predictable multi-core hardware and software have been devised [22]. A multi-core architecture providing timing predictability for hard real-time tasks was suggested in [15].

WCET analysis of parallel programs, running on parallel hardware, has not been much studied. A case study using timed automata was reported in [7]. Another case study is found in [16]. In [8], a general WCET analysis for parallel programs with threads, shared memory, and locks is devised and its correctness is proved for the restricted class of programs that do not use locks. This analysis integrates flow analysis and WCET calculation using abstract execution in a manner similar to [5].

In [13], a simple cost model for a GPU program was estimated from measurements and used in a subsequent WCET calculation. No systematic WCET analysis was defined, though.

8 Conclusions and Further Research

WCET analysis of parallel software can easily become very difficult. Sources of unpredictability are shared hardware resources, providing channels for different activities to affect each others' timing, and low-level concurrent programming models that are inherently nondeterministic.

We discuss the use of restricted parallel programming models to make parallel software more amenable to timing analysis, with a focus on models for performance-demanding real-time applications. Such models have been proposed in the past, as means to make parallel programming easier and to allow general performance models. We review BSP and data parallel programming, and we conclude that especially the latter provides a promising model for writing timing-predictable parallel software.

We furthermore review a common current GPU architecture and computation model, and conclude that it seems to provide a reasonably timing-predictable platform for high performance computing applications. The combination of data parallel programming and GPU execution seems especially interesting, since the data parallel primitives can be translated into kernels that will make timing-predictable use of the streaming multiprocessors in the GPUs.

A possible development of future hardware architecture is that we will see heterogenous processors with a few general purpose multi-cores, and an on-chip massively parallel co-processor, building on GPU technology, for computationally demanding applications. Already now Broadcom offers the BCM2835 System-on-Chip with an ARM11 processor and a Videocore 4 GPU. Such future processors will surely find their way into various embedded real-time applications. This means that we will need timing analysis for these systems. In this paper, we have laid out a path how to get there. However, other directions are also possible: we may also see future many-core processors along the lines of the Tilera architecture, with a large number of general purpose cores equipped with distributed memory and connected by a NoC. The data parallel paradigm also provides good support for such architectures due to its deterministic primitives, and their separation into primitives for computation and communication. It seems likely that good timing models should be possible to develop for these primitives also when implemented on such architectures.

Future work will include designing suitable timing models and low-level analyses for future parallel architectures, and design WCET analyses that that can use these to provide tight and safe WCET bounds.

Acknowledgment

This research was supported by the Swedish Foundation for Strategic Research through grant IIS11-0060 (Ralf 3 – Software for Embedded High Performance Architecture).

References

1 David R. Butenhof. *Programming with POSIX Threads.* Addison-Wesley, 1997.

2 Sudipta Chattopadhyay, C.-L. Kee, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. A unified WCET analysis framework for multi-core platforms. In

18th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS'12), Beijing, China, April 2012.

3 Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In Todor Stefanov, editor, *Proc. 13th International Workshop on Software and Compilers for Embedded Systems (SCOPES'10)*, pages 6:1–6:10, St. Goar, Germany, June 2010. ACM.

4 Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *Computing in Science and Engineering*, 5(1):46–55, 1998.

5 Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Deriving WCET bounds by abstract execution. In Chris Healy, editor, *Proc. 11th International Workshop on Worst-Case Execution Time Analysis (WCET'2011)*, Porto, Portugal, July 2011.

6 Message Passing Interface Forum. MPI: A message passing interface standard. Technical report, University of Tennessee, Knoxville, Tennessee, May 1994.

7 Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET analysis of multicore architectures using UPPAAL. In Björn Lisper, editor, *Proc. 10th International Workshop on Worst-Case Execution Time Analysis (WCET'2010)*, pages 103–113, Brussels, Belgium, July 2010. OCG.

8 Andreas Gustavsson, Jan Gustafsson, and Björn Lisper. Toward static timing analysis of parallel software. Accepted for publication in 12th International Workshop on Worst-Case Execution Time Analysis (WCET'2012).

9 Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Proc. 30th IEEE Real-Time Systems Symposium (RTSS'09)*, pages 68–77, 2009.

10 K. E. Iverson. *A Programming Language*. Wiley, London, 1962.

11 Y. Li, Vivy Suhendra, Y. Liang, Tulika Mitra, and Abhik Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *Proc. 30th IEEE Real-Time Systems Symposium (RTSS'09)*, Washington, D.C., USA, December 2009.

12 Mingsong Lv, Nan Guan, Wang Yi, and Ge Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In Scott Brandt, editor, *Proc. 31th IEEE Real-Time Systems Symposium (RTSS'10)*, pages 339–349, San Diego, CA, December 2010. IEEE.

13 Rahul Mangharam and Aminreza Abrahimi Saba. Anytime algorithms for GPU architectures. In Luis Almeida, editor, *Proc. 32nd IEEE Real-Time Systems Symposium (RTSS'11)*, pages 47–56, Vienna, Austria, December 2011. IEEE Computer Society.

14 NVIDIA CUDA C programming guide, November 2011.

15 Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *Proc. 36th International Symposium on Computer Architecture (ISCA 2009)*, pages 57–68, 2009.

16 Christine Rochange, Armelle Bonenfant, Pascal Sainrat, Mike Gerdes, Julian Wolf, Theo Ungerer, Zlatko Petrov, and Frantisek Mikulu. WCET analysis of a parallel 3D multigrid solver executed on the MERASA multi-core. In Björn Lisper, editor, *Proc. 10th International Workshop on Worst-Case Execution Time Analysis (WCET'2010)*, pages 90–100, Brussels, Belgium, July 2010. OCG.

17 Jakob Rosen, Alexandru Andrei, Petru Eles, and Zebo Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proc. 28th IEEE Real-Time Systems Symposium (RTSS'07)*, pages 49–60, Washington, DC, USA, 2007. IEEE Computer Society.

18 Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *Proc. 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on*

Graphics hardware, GH ’07, pages 97–106, Aire-la-Ville, Switzerland, 2007. Eurographics Association.

19 Jay M. Sipelstein and Guy E. Blelloch. Collection-oriented languages. *Proc. IEEE*, 79(4):504–523, April 1991.

20 David B. Skillicorn, Jonathan M. D. Hill, and W. F. McColl. Questions and answers about BSP. *Scientific Programming*, 6:249–274, 1997.

21 Leslie G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33(8):103–111, August 1990.

22 Reinhard Wilhelm. The PROMPT design principles for predictable multi-core architectures. In *Proc. 12th International Workshop on Software and Compilers for Embedded Systems (SCOPES’09)*, pages 31–32, Nice, France, April 2009. ACM.

Timing Analysis of Concurrent Programs

Robert Mittermayr and Johann Blieberger

TU Vienna, Institute of Computer-Aided Automation
Treitlstr. 1–3, 1040 Vienna, Austria
{robert,blieb}@auto.tuwien.ac.at

Abstract

Worst-case execution time analysis of multi-threaded software is still a challenge. This comes mainly from the fact that the number of thread interleavings grows exponentially in the number of threads and that synchronization has to be taken into account. In particular, a suitable graph based model has been missing. The idea that thread interleavings can be studied with a matrix calculus is a novel approach in this research area. Our sparse matrix representations of the program are manipulated using Kronecker algebra. The resulting graph represents the multi-threaded program and plays a similar role for concurrent systems as control flow graphs do for sequential programs. Thus a suitable graph model for timing analysis of multi-threaded software has been set up. Due to synchronization it turns out that often only very small parts of the resulting graph are actually needed, whereas the rest is unreachable. A lazy implementation of the matrix operations ensures that the unreachable parts are never calculated. This speeds up processing significantly and shows that our approach is very promising.

1998 ACM Subject Classification D.2.4 Software/Program Verification, F.3.2 Program analysis

Keywords and phrases Worst-case execution time analysis (WCET), Concurrency, Thread Synchronization, Kronecker Algebra, Program Analysis

Digital Object Identifier 10.4230/OASIcs.WCET.2012.59

1 Introduction

Since concurrent programs may contain blocking because of synchronization between threads, the terms execution time and worst-case execution time (WCET) do not apply directly to concurrent systems. Anyway, we stick to the term WCET for concurrent systems. The reader, however, has to be aware of the fact that the WCET includes blocking time.

With the advent of multi-core processors scientific and industrial interest focuses on analysis and verification of multi-threaded applications. The scientific challenge comes from the fact that the number of thread interleavings grows exponentially in a program's number of threads. All state-of-the-art methods suffer from this so-called *state explosion problem*.

The idea that thread interleavings of concurrent programs can be studied with a matrix calculus is novel in this research area. Our sparse matrix representations of the program are manipulated using a lazy implementation of Kronecker algebra. Similar to [3] we describe synchronization by Kronecker products and thread interleavings by Kronecker sums. The first goal is the generation of a data structure called *Concurrent Program Graph* (CPG) which describes all possible interleavings and incorporates synchronization while preserving completeness. CPGs play a similar role for concurrent systems as control flow graphs (CFGs) do for sequential programs.

In this paper CPGs are used to calculate the WCET of the underlying concurrent system.

In [12] it is shown that CPGs in general can be represented by sparse adjacency matrices. Thus the number of entries in the matrices is linear in their number of lines. In the worst-case the number of lines increases exponentially in the number of threads. The CPG, however, contains many nodes and edges unreachable from the entry node. If the program contains a lot of synchronization, only a very small part of the CPG is reachable. Our lazy implementation of the matrix operations computes only this part. This optimization speeds up processing significantly and shows that our approach is very promising.

The outline of our paper is as follows. In Section 2 Refined CFGs and Kronecker algebra are introduced. Our model of concurrency, its properties, and our lazy approach are presented in Section 3. Section 4 is devoted to WCET analysis of multi-threaded programs. In Section 5 we survey related work. Finally, we draw our conclusion in Section 6.

2 Preliminaries

In this paper we refer to both, a processor and a core, as a processor. Our computational model can be described as follows. We model concurrent programs by threads which use semaphores for synchronization. We assume that on each processor exactly one thread is running and each thread immediately executes its next statement if the thread is not blocked. Blocking may occur only in succession of semaphore calls.

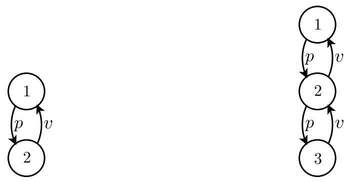
Threads and semaphores are represented by slightly adapted CFGs. *Edge Splitting* has to be applied to the edges containing semaphore calls. Each CFG is represented by an adjacency matrix. We assume that the edges of CFGs are labeled by elements of a semiring. Definitions and properties of the semiring can be found in [10, 12]. A prominent example for such semirings are regular expressions describing the behavior of finite state automata. Our semiring consists of a set of labels $\mathcal{L}$ which is defined by $\mathcal{L} = \mathcal{L}_\mathrm{V} \cup \mathcal{L}_\mathrm{S}$, where $\mathcal{L}_\mathrm{V}$ is the set of non-synchronization labels and $\mathcal{L}_\mathrm{S}$ is the set of labels representing semaphore calls ($\mathcal{L}_\mathrm{V}$ and $\mathcal{L}_\mathrm{S}$ are disjoint).

Usually two or more distinct thread CFGs refer to the same semaphore to model synchronization. The other labels are elements of $\mathcal{L}_\mathrm{V}$ and model ordinary program statements. The operations on the basic blocks are $\cdot, +$, and $*$ from a semiring (cf. [15]). Intuitively, $\cdot, +$, and $*$ model consecutive program parts, conditionals, and loops, respectively.

2.1 Refined Control Flow Graphs

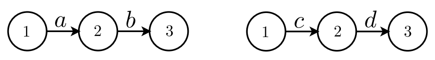
Usually CFG nodes represent basic blocks. Because our matrix calculus manipulates the edges we need to have basic blocks on the (incoming) edges. To keep things simple we refer to edges, their labels and the corresponding entries of the adjacency matrices synonymously. A basic block consists of multiple consecutive statements without jumps. For our purpose we need a finer granularity which we achieve by splitting edges. We apply it for semaphore calls (e.g. p_1 and v_1) and require that a semaphore call referred to as s_i has to be the only statement on the corresponding edge. Edge splitting maps a CFG edge e whose corresponding basic block contains k semaphore calls to a subgraph $\circ \xrightarrow{e_1} \circ \xrightarrow{s_1} \circ \xrightarrow{e_2} \circ \xrightarrow{s_2} \circ \cdots \circ \xrightarrow{e_k} \circ \xrightarrow{s_k} \circ \xrightarrow{e_{k+1}} \circ$, such that each s_i represents a single semaphore call, and e_i and e_{i+1} represent the consecutive parts before and after s_i, respectively ($1 \leq i \leq k$). Applying this procedure and edge splitting we result in a *Refined Control Flow Graph* (RCFG).

In Fig. 1a and 1b a binary and a counting semaphore are depicted. The latter allows two threads to enter at the same time. In a similar way it is possible to construct semaphores allowing n non-blocking p-calls ($n \in \mathbb{N}, n \geq 1$).



(a) Binary Semaphore **(b)** Counting Semaphore

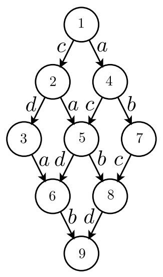
Figure 1 Semaphores.



(a) C **(b)** D

Interleavings
$a \cdot b \cdot c \cdot d$
$a \cdot c \cdot b \cdot d$
$a \cdot c \cdot d \cdot b$
$c \cdot a \cdot b \cdot d$
$c \cdot a \cdot d \cdot b$
$c \cdot d \cdot a \cdot b$

(c) Interleavings



(d) $C \oplus D$

Figure 2 A Simple Example.

2.2 Modeling Synchronization and Interleavings

Kronecker product and Kronecker sum form Kronecker algebra. In the following we define both operations. Proofs, additional properties, and examples can be found in [1, 5, 6, 12]. From now on we use matrices out of $\mathcal{M} = \{M = (m_{i,j}) \,|\, m_{i,j} \in \mathcal{L}\}$ only.

▶ **Definition 1** (Kronecker product)**.** Given a m-by-n matrix A and a p-by-q matrix B, their *Kronecker product* denoted by $A \otimes B$ is a mp-by-nq block matrix defined by

$$A \otimes B = \begin{pmatrix} a_{1,1} \cdot B & \cdots & a_{1,n} \cdot B \\ \vdots & \ddots & \vdots \\ a_{m,1} \cdot B & \cdots & a_{m,n} \cdot B \end{pmatrix}.$$

The Kronecker product is also being referred to as *Zehfuss product.* Kronecker product allows to model synchronization (cf. [3, 12, 13]).

▶ **Definition 2** (Kronecker sum)**.** Given a matrix A of order[1] m and matrix B of order n, their *Kronecker sum* denoted by $A \oplus B$ is a matrix of order mn defined by $A \oplus B = A \otimes I_n + I_m \otimes B$, where I_m and I_n denote identity matrices of order m and n, respectively.

Note that Kronecker sum calculates all possible interleavings (see e.g. [11] for a proof) even for general CFGs including conditionals and loops. The following example illustrates interleaving of threads and how Kronecker sum handles it.

▶ **Example 3.** Let the matrices $C = \begin{pmatrix} 0 & a & 0 \\ 0 & 0 & b \\ 0 & 0 & 0 \end{pmatrix}$ and $D = \begin{pmatrix} 0 & c & 0 \\ 0 & 0 & d \\ 0 & 0 & 0 \end{pmatrix}$. The CFGs of matrices C and D are shown in Fig. 2a and Fig. 2b, respectively. The regular expressions

[1] A k-by-k matrix is known as square matrix of order k.

associated with the CFGs are $a \cdot b$ and $c \cdot d$. All possible interleavings by executing C and D in an interleavings semantics are shown in Fig. 2c. In Fig. 2d the graph represented by the adjacency matrix $C \oplus D$ is depicted. It is easy to see that all possible interleavings are generated correctly. It is worth noting that $\oplus$ provides correct results even if the operands contain branches and loops.

3 Concurrent Program Graphs

Our system model consists of a finite number of threads and semaphores which are represented by RCFGs. The RCFGs are stored in form of adjacency matrices. The matrices have entries which are referred to as labels $l \in \mathcal{L}$ as defined in Sect. 2.

Formally, the system model consists of the tuple $\langle \mathcal{T}, \mathcal{S}, \mathcal{L} \rangle$, where $\mathcal{T}$ is the set of RCFG adjacency matrices describing threads, $\mathcal{S}$ refers to the set of RCFG adjacency matrices describing semaphores, and the labels in $T \in \mathcal{T}$ and $S \in \mathcal{S}$ are elements of $\mathcal{L}$ and $\mathcal{L}_{\mathrm{S}}$, respectively. The matrices are manipulated by using Kronecker algebra.

A *Concurrent Program Graph* (CPG) is a graph $C = \langle V, E, n_e \rangle$ with a set of nodes V, a set of directed edges $E \subseteq V \times V$, and a so-called *entry* node $n_e \in V$. The sets V and E are constructed out of the elements of $\langle \mathcal{T}, \mathcal{S}, \mathcal{L} \rangle$. Details on how we generate the sets V and E follow below. Similar to RCFGs the edges of CPGs are labeled by $l \in \mathcal{L}$.

3.1 Generating a Concurrent Program's Matrix

Let $T^{(i)} \in \mathcal{T}$ and $S^{(i)} \in \mathcal{S}$ refer to the matrices representing thread i and semaphore i, respectively. According to Fig. 1a we have for binary semaphore i the adjacency matrix $S^{(i)} = \begin{pmatrix} 0 & p_i \\ v_i & 0 \end{pmatrix}$ of order two. We obtain the matrix T representing k interleaved threads and the matrix S representing r interleaved semaphores by

$$T = \bigoplus_{i=1}^{k} T^{(i)}, \text{ where } T^{(i)} \in \mathcal{T} \text{ and } S = \bigoplus_{i=1}^{r} S^{(i)}, \text{ where } S^{(i)} \in \mathcal{S}.$$

Note that the associativity properties (cf. [12]) of the operations $\otimes$ and $\oplus$ imply that the corresponding n-fold versions are well defined. In the following we define the Selective Kronecker product which we denote by $\oslash_L$. This operator synchronizes only labels identical in the two input matrices.

▶ **Definition 4** (Selective Kronecker product). Given two matrices A and B we call $A \oslash_L B$ their Selective Kronecker product. For all $l \in L \subseteq \mathcal{L}$ let $A \oslash_L B = (a_{i,j}) \oslash_L (b_{p,q}) = (c_{i \cdot p, j \cdot q})$, where

$$c_{i \cdot p, j \cdot q} = \begin{cases} l & \text{if } a_{i,j} = b_{p,q} = l, \ l \in L, \\ 0 & \text{otherwise.} \end{cases}$$

▶ **Definition 5** (Filtered Matrix). We call M_L a *Filtered Matrix* and define it as a matrix of order $o(M)$ containing entries $l \in L \subseteq \mathcal{L}$ of $M = (m_{i,j})$ and zeros elsewhere:

$$M_L = (m_{L;i,j}), \text{ where } m_{L;i,j} = \begin{cases} l & \text{if } m_{i,j} = l, \ l \in L, \\ 0 & \text{otherwise.} \end{cases}$$

The adjacency matrix representing program $\mathcal{P}$ is referred to as P. In [12] it is proved that P can be computed efficiently by

$$P = T \oslash_{\mathcal{L}_{\mathrm{S}}} S + T_{\mathcal{L}_{\mathrm{V}}} \otimes I_{o(S)}.$$

In addition, it is shown in [12] that the resulting CPG has at most n^k nodes and at most $2k\,n^k$ edges, if k is the number of threads and each thread has n nodes in its RCFG. Hence CPGs have a sparse adjacency matrix, i.e., $|E| = O\big(|V|\big)$. Thus memory saving data structures and efficient algorithms suggest themselves. In the worst-case, however, the number of CPG nodes increases exponentially in k.

3.2 Lazy Implementation of Kronecker Algebra

In general, a CPG contains unreachable parts if a concurrent program contains synchronization (cf. [12]). If a program contains a lot of synchronization, the reachable parts may be very small. This observation motivates the lazy implementation described in this subsection. In the following we denote the subgraph of a CPG, whose nodes are reachable from the entry node, by RCPG. An empirical analysis of our approach showed that the runtime complexity of generating a RCPG is linear in the number of RCPG nodes [12].

The reasons why parts of the CPG are unreachable can be summarized as follows: Kronecker product limits the number of possible paths such that the p- and v-operations are present in correct p-v-pairs in the RCPG. In contrast $T = \bigoplus_{i=1}^{k} T_i$ contains all possible paths even those containing semantically wrong uses of the semaphore operations.

Choosing a lazy implementation (cf. [9]) for the matrix operations ensures that, when extracting the reachable parts of the underlying graph, the overall effort is reduced to exactly these parts. By starting from the RCPG's entry node and calculating all reachable successor nodes our lazy implementation [12] exactly does this. Thus, for example, if the resulting RCPG's size is linear in terms of the involved threads, only linear effort will be necessary to generate the RCPG.

Our implementation distinguishes between two kind of matrices: Sparse matrices are used for representing threads and semaphores. Lazy matrices are employed for representing all the other matrices, i.e., those resulting from the operations of Kronecker algebra. Besides the employed operation, a lazy matrix simply keeps track of its operands. Whenever an entry of a lazy matrix is retrieved, depending on the operation recorded in the lazy matrix, entries of the operands are retrieved and the recorded operation is performed on these entries to calculate the result. In the course of this computation, even the successors of nodes are calculated lazily. Retrieving entries of operands is done recursively if the operands are again lazy matrices, or is done by retrieving the entries from the sparse matrices, where the actual data resides. The lazy implementation has proven to be very space and time efficient.

4 Worst-Case Execution Time Analysis on RCPGs

In order to calculate the WCET of a concurrent program we apply a dataflow based approach introduced in [2]. Dataflow equations are set up and solved according to [14]. Details can be found in [2, 14].

Each node of the RCPG is assigned a dataflow variable and a dataflow equation is set up based on the predecessors of the RCPG node. A dataflow variable is represented by a vector. Each component of the vector reflects a processor and is used to calculate the WCET of the corresponding thread. Recall that only a single thread is allocated to a processor.

Synchronizing nodes, introduced below, are nodes where blocking occurs. These nodes have an incoming edge labeled by a semaphore v-operation, an outgoing edge labeled by a p-operation of the same semaphore, and these edges are part of different threads. In this case the thread with the p-operation has to wait until the other thread's v-operation is finished.

Let the vector $\mathfrak{X} = (X_1, \ldots, X_\ell, \ldots, X_p)^\intercal$. We write $\mathfrak{X}^{(\ell)} = X_\ell$ to denote the ℓth component of vector $\mathfrak{X}$.

▶ **Definition 1.** Let $\mathfrak{X} = (X_1, \ldots, X_p)^\intercal$ and $\mathfrak{Y} = (Y_1, \ldots, Y_p)^\intercal$. Then we define

$$\max(\mathfrak{X}, \mathfrak{Y}) := (\max(X_1, Y_1), \ldots, \max(X_p, Y_p))^\intercal.$$

▶ **Definition 2.** A *synchronizing node* is a RCPG node s such that

- there exists an edge $e_{in} = (i, s)$ with label v_k and
- there exists an edge $e_{out} = (s, j)$ with label p_k,

where k denotes the same semaphore and e_{in} and e_{out} are mapped to different processors, i.e., $\mathfrak{P}(e_{in}) \neq \mathfrak{P}(e_{out})$.

▶ **Definition 3** (Setting up dataflow equations)**.** If n is a non-synchronizing node, then

$$\mathfrak{X}_n = \max_{k \in \mathrm{Pred}(n)} (\mathfrak{X}_k + \mathfrak{t}(k \to n)),$$

where the ℓth component of vector $\mathfrak{t}(k \to n)$ is the time assigned to edge $k \to n$ and edge $k \to n$ is mapped to processor ℓ. The other components of $\mathfrak{t}(k \to n)$ are zero. The set of predecessor nodes of node n is referred to as $\mathrm{Pred}(n)$.

Let s be a synchronizing node. In addition, let π_i and π_j be the processors which the edges $i \to s$ and $s \to j$ are mapped to, i.e, $\pi_i = \mathfrak{P}(i \to s)$ and $\pi_j = \mathfrak{P}(s \to j)$. Then for $\ell \neq \pi_j$

$$\mathfrak{X}_s^{(\ell)} = \max_{k \in \mathrm{Pred}(s)} \left(\mathfrak{X}_k^{(\ell)} + \mathfrak{t}(k \to s)^{(\ell)} \right)$$

and

$$\mathfrak{X}_s^{(\pi_j)} = \max \left(\mathfrak{X}_i^{(\pi_i)} + \mathfrak{t}(i \to s)^{(\pi_i)}, \max_{k : \mathfrak{P}(k \to s) = \pi_j} \left(\mathfrak{X}_k^{(\pi_j)} + \mathfrak{t}(k \to s)^{(\pi_j)} \right) \right)$$

where the first term considers the incoming v-edge and the second term takes into account all incoming edges of the blocking thread running on processor π_j.

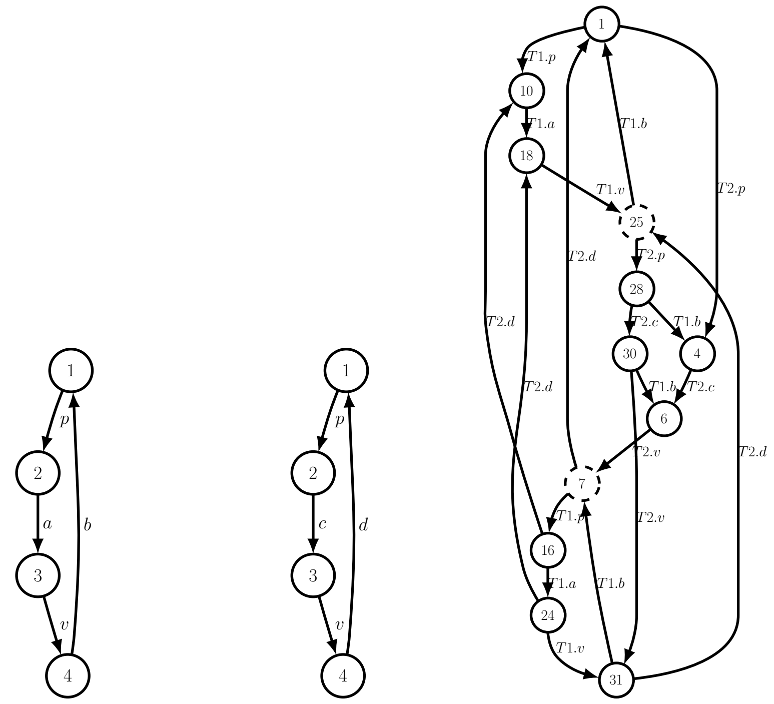
The system of dataflow equations can be solved by applying an algorithm presented in [14]. It relies on two operations: inserting one equation into another and solving recursions by so-called loop breaking. The order of these operations is completely determined by the DJ graph introduced in [14].

In contrast to [2] where CFGs are studied, RCPGs contain several copies of basic blocks in different places. Thus, during a loop breaking operation the number of loop iterations cannot be determined immediately. Instead we postpone the assigning of loop iterations and indicate this by "$*$" which denotes a number of loop iterations to be assigned later.

After the system of dataflow equations has been solved, we distribute the known number of loop iterations among all terms labeled by "$*$" such that the timing values achieve their maxima.

Example

We study an example consisting of two threads $T1$ and $T2$. Their RCFGs are depicted in Figures 3a and 3b, respectively. The behaviors of $T1$ and $T2$ are $(p \cdot a \cdot v \cdot b)^*$ and $(p \cdot c \cdot v \cdot d)^*$, respectively. The RCPG of the T1-T2-system together with a simple binary semaphore (depicted in Fig. 1a) is shown in Figure 3c. Note that the node numbers are generated by our implementation. Missing node numbers refer to unreachable nodes. The dashed nodes 7 and 25 are the only synchronizing nodes.



(a) RCFG of thread T1 **(b)** RCFG of thread T2 **(c)** RCPG of example program

Figure 3 RCFGs of threads T1 and T2 and the resulting RCPG.

According to Definition 3 the following equations are set up. For simplicity we do not distinguish between edge labels and the execution time of the corresponding basic blocks, i.e., the execution time of a basic block x is denoted by x.

$$\mathfrak{X}_1 = \max\left(\mathfrak{X}_7 + \begin{pmatrix}0\\d\end{pmatrix}, \mathfrak{X}_{25} + \begin{pmatrix}b\\0\end{pmatrix}\right), \qquad \mathfrak{X}_{10} = \max\left(\mathfrak{X}_1 + \begin{pmatrix}p\\0\end{pmatrix}, \mathfrak{X}_{16} + \begin{pmatrix}0\\d\end{pmatrix}\right)$$

$$\mathfrak{X}_{18} = \max\left(\mathfrak{X}_{10} + \begin{pmatrix}a\\0\end{pmatrix}, \mathfrak{X}_{24} + \begin{pmatrix}0\\d\end{pmatrix}\right), \qquad \mathfrak{X}_{25} = \begin{pmatrix}\mathfrak{X}_{18}^{(1)} + v\\ \max\left(\mathfrak{X}_{18}^{(1)} + v, \mathfrak{X}_{31}^{(2)} + d\right)\end{pmatrix}$$

$$\mathfrak{X}_{28} = \mathfrak{X}_{25} + \begin{pmatrix}0\\p\end{pmatrix}, \qquad \mathfrak{X}_{30} = \mathfrak{X}_{28} + \begin{pmatrix}0\\c\end{pmatrix}$$

$$\mathfrak{X}_4 = \max\left(\mathfrak{X}_1 + \begin{pmatrix}0\\p\end{pmatrix}, \mathfrak{X}_{28} + \begin{pmatrix}b\\0\end{pmatrix}\right), \qquad \mathfrak{X}_6 = \max\left(\mathfrak{X}_4 + \begin{pmatrix}0\\c\end{pmatrix}, \mathfrak{X}_{30} + \begin{pmatrix}b\\0\end{pmatrix}\right)$$

$$\mathfrak{X}_7 = \begin{pmatrix}\max\left(\mathfrak{X}_6^{(2)} + v, \mathfrak{X}_{31}^{(1)} + b\right)\\ \mathfrak{X}_6^{(2)} + v\end{pmatrix}, \qquad \mathfrak{X}_{16} = \mathfrak{X}_7 + \begin{pmatrix}p\\0\end{pmatrix}$$

$$\mathfrak{X}_{24} = \mathfrak{X}_{16} + \begin{pmatrix}a\\0\end{pmatrix}, \qquad \mathfrak{X}_{31} = \max\left(\mathfrak{X}_{24} + \begin{pmatrix}v\\0\end{pmatrix}, \mathfrak{X}_{30} + \begin{pmatrix}0\\v\end{pmatrix}\right)$$

We solve the above equations by using the DJ-graph [14] of our RCPG. For a concise presentation we use $\alpha = p + a + v$, $\gamma = p + c + v$, $T_1 = \alpha + b$, $T_2 = \gamma + d$, $M_2 = \alpha + \gamma + T_2^*$, and $M_1 = \max(T_1, M_2)$. We perform the following insertions and loop breaking operations:

$$24 \to 18 : \mathfrak{X}_{18} = \max\left(\mathfrak{X}_{10} + \begin{pmatrix}a\\0\end{pmatrix}, \mathfrak{X}_{16} + \begin{pmatrix}a\\d\end{pmatrix}\right)$$

$$24,30 \to 31 : \mathfrak{x}_{31} = \max\left(\mathfrak{x}_{16} + \begin{pmatrix} a+v \\ 0 \end{pmatrix}, \mathfrak{x}_{28} + \begin{pmatrix} 0 \\ v+c \end{pmatrix}\right)$$

$$30 \to 6 : \mathfrak{x}_{6} = \max\left(\mathfrak{x}_{4} + \begin{pmatrix} 0 \\ c \end{pmatrix}, \mathfrak{x}_{28} + \begin{pmatrix} b \\ c \end{pmatrix}\right)$$

$$16 \to 18 : \mathfrak{x}_{18} = \max\left(\mathfrak{x}_{10} + \begin{pmatrix} a \\ 0 \end{pmatrix}, \mathfrak{x}_{7} + \begin{pmatrix} a+p \\ d \end{pmatrix}\right)$$

$$16 \to 31 : \mathfrak{x}_{31} = \max\left(\mathfrak{x}_{7} + \begin{pmatrix} \alpha \\ 0 \end{pmatrix}, \mathfrak{x}_{28} + \begin{pmatrix} 0 \\ v+c \end{pmatrix}\right)$$

$$16 \to 10 : \mathfrak{x}_{10} = \max\left(\mathfrak{x}_{1} + \begin{pmatrix} p \\ 0 \end{pmatrix}, \mathfrak{x}_{7} + \begin{pmatrix} p \\ d \end{pmatrix}\right)$$

$$28 \to 31 : \mathfrak{x}_{31} = \max\left(\mathfrak{x}_{7} + \begin{pmatrix} \alpha \\ 0 \end{pmatrix}, \mathfrak{x}_{25} + \begin{pmatrix} 0 \\ \gamma \end{pmatrix}\right)$$

$$28 \to 6 : \mathfrak{x}_{6} = \max\left(\mathfrak{x}_{4} + \begin{pmatrix} 0 \\ c \end{pmatrix}, \mathfrak{x}_{25} + \begin{pmatrix} b \\ c+p \end{pmatrix}\right)$$

$$28 \to 4 : \mathfrak{x}_{4} = \max\left(\mathfrak{x}_{1} + \begin{pmatrix} 0 \\ p \end{pmatrix}, \mathfrak{x}_{25} + \begin{pmatrix} b \\ p \end{pmatrix}\right)$$

$$10 \to 18 : \mathfrak{x}_{18} = \max\left(\mathfrak{x}_{1} + \begin{pmatrix} p+a \\ 0 \end{pmatrix}, \mathfrak{x}_{7} + \begin{pmatrix} p+a \\ 0 \end{pmatrix}\right)$$

$$18 \to 25 : \mathfrak{x}_{25} = \begin{pmatrix} \max\left(\mathfrak{x}_1^{(1)}, \mathfrak{x}_7^{(1)}\right) + \alpha \\ \max\left(\max\left(\mathfrak{x}_1^{(1)}, \mathfrak{x}_7^{(1)}\right) + \alpha, \mathfrak{x}_{31}^{(2)} + d\right) \end{pmatrix}$$

$$25 \to 4 : \mathfrak{x}_{4} = \max\left(\mathfrak{x}_{1} + \begin{pmatrix} 0 \\ p \end{pmatrix}, \begin{pmatrix} \max\left(\mathfrak{x}_1^{(1)}, \mathfrak{x}_7^{(1)}\right) + T_1 \\ \max\left(\max\left(\mathfrak{x}_1^{(1)}, \mathfrak{x}_7^{(1)}\right) + \alpha, \mathfrak{x}_{31}^{(2)} + d\right) + p \end{pmatrix}\right)$$

$$25 \to 31 : \mathfrak{x}_{31} = \begin{pmatrix} \max\left(\mathfrak{x}_7^{(1)} + \alpha, \mathfrak{x}_1^{(1)} + \alpha\right) \\ \max\left(\mathfrak{x}_7^{(2)}, \mathfrak{x}_1^{(1)} + \alpha + \gamma, \mathfrak{x}_7^{(1)} + \alpha + \gamma, \mathfrak{x}_{31}^{(2)} + T_2\right) \end{pmatrix}$$

$$4,25 \to 6 : \mathfrak{x}_{6} = \begin{pmatrix} \max\left(\mathfrak{x}_1^{(1)}, \mathfrak{x}_7^{(1)}\right) + T_1 \\ \max\left(\mathfrak{x}_1^{(2)} + p + c, \mathfrak{x}_1^{(1)} + \alpha + \gamma, \mathfrak{x}_7^{(1)} + \alpha + \gamma, X_{31}^{(2)} + d + p + c\right) \end{pmatrix}$$

$$6 \to 7 : \mathfrak{x}_{7} = \begin{pmatrix} \max\left(\mathfrak{x}_1^{(2)} + \gamma, \mathfrak{x}_1^{(1)} + \alpha + \gamma, \mathfrak{x}_7^{(1)} + \alpha + \gamma, \mathfrak{x}_{31}^{(2)} + T_2, \mathfrak{x}_{31}^{(1)} + b\right) \\ \max\left(\mathfrak{x}_1^{(2)} + \gamma, \mathfrak{x}_1^{(1)} + \alpha + \gamma, \mathfrak{x}_7^{(1)} + \alpha + \gamma, \mathfrak{x}_{31}^{(2)} + T_2\right) \end{pmatrix}$$

$$31\ \emptyset : \mathfrak{x}_{31} = \begin{pmatrix} \max\left(\mathfrak{x}_7^{(1)} + \alpha, \mathfrak{x}_1^{(1)} + \alpha\right) \\ \max\left(\mathfrak{x}_7^{(2)}, \mathfrak{x}_1^{(1)} + \alpha + \gamma, \mathfrak{x}_7^{(1)} + \alpha + \gamma\right) + T_2^* \end{pmatrix}$$

$$31 \to 7 : \mathfrak{x}_{7} = \begin{pmatrix} \max\left(\mathfrak{x}_1^{(2)} + \gamma, \mathfrak{x}_1^{(1)} + M_2, \mathfrak{x}_1^{(1)} + T_1, \mathfrak{x}_7^{(2)} + T_2^*, \mathfrak{x}_7^{(1)} + M_2, \mathfrak{x}_7^{(1)} + T_1\right) \\ \max\left(\mathfrak{x}_1^{(2)} + \gamma, \mathfrak{x}_1^{(1)} + M_2, \mathfrak{x}_7^{(2)} + T_2^*, \mathfrak{x}_7^{(1)} + M_2\right) \end{pmatrix}$$

$$7\ \emptyset : \mathfrak{x}_{7} = \begin{pmatrix} \max\left(\mathfrak{x}_1^{(2)} + \gamma, \mathfrak{x}_1^{(1)} + M_2, \mathfrak{x}_1^{(1)} + T_1\right) + M_1^* \\ \max\left(\mathfrak{x}_1^{(2)} + \gamma, \mathfrak{x}_1^{(1)} + M_2\right) + M_1^* + M_2 \end{pmatrix}$$

$$7 \to 31 : \mathfrak{x}_{31} = \begin{pmatrix} \max\left(\mathfrak{x}_1^{(2)} + \gamma, \mathfrak{x}_1^{(1)} + M_2, \mathfrak{x}_1^{(1)} + T_1\right) + M_1^* + \alpha \\ \max\left(\mathfrak{x}_1^{(2)} + \gamma, \mathfrak{x}_1^{(1)} + M_2, \mathfrak{x}_1^{(1)} + T_1\right) + M_1^* + M_2 \end{pmatrix}$$

$$7,31 \to 25 : \mathfrak{X}_{25} = \begin{pmatrix} \max\left(\mathfrak{X}_1^{(2)} + \gamma + M_1^*, \mathfrak{X}_1^{(1)} + M_2 + M_1^*, \mathfrak{X}_1^{(1)} + T_1 + M_1^*\right) + \alpha \\ \max\left(\mathfrak{X}_1^{(2)} + T_2, \mathfrak{X}_1^{(1)} + \alpha + T_2^*, \mathfrak{X}_1^{(1)} + T_1 + d\right) + M_1^* + M_2 \end{pmatrix}$$

$$7,25 \to 1 : \mathfrak{X}_{1} = \begin{pmatrix} \max\left(\mathfrak{X}_1^{(2)} + \gamma + M_1^*, \mathfrak{X}_1^{(1)} + M_2 + M_1^*, \mathfrak{X}_1^{(1)} + T_1 + M_1^*\right) + T_1 \\ \max\left(\mathfrak{X}_1^{(2)} + T_2, \mathfrak{X}_1^{(1)} + \alpha + T_2^*, \mathfrak{X}_1^{(1)} + T_1 + d\right) + M_1^* + M_2 \end{pmatrix}$$

$$1\ \not{\circlearrowleft}: \mathfrak{X}_1 = \begin{pmatrix} M_1^* + T_1^* \\ M_1^* + T_1^* + T_2^* + \alpha \end{pmatrix}$$

Finally we obtain the following formula for the WCET of our T1-T2-system

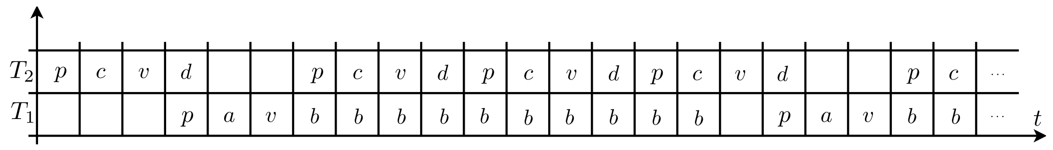
$$\text{WCET} = \max(\mathfrak{X}_1^{(1)}, \mathfrak{X}_1^{(2)}) = M_1^* + T_1^* + T_2^* + \alpha.$$

Assuming the execution times $p = v = a = c = d = 1$, $b = 10$, and the number of loop iterations of $T1$ and $T2$ to be r and s, respectively, we are left with distributing the number of loop iterations among the "$*$"-terms of our WCET formula. In this case we have $M_2(k) = \max(13, 6 + 4k)$ where the number 6 already includes one execution of a and c. Thus we write $M_2(k) = \max(13, 2 + 4k)$ and we have to choose k such that $M_2(k) = 2 + 4k$ and $M_2(i) = 13$ for $i < k$. This is a consequence of our system model presented in Section 2. We get $k = 3$, i.e., T_2 is executed three times while M_1 is executed once. Hence we obtain

$$\text{WCET} = \begin{cases} 14\left\lfloor\frac{s-1}{3}\right\rfloor + 13\left(r - \left\lfloor\frac{s-1}{3}\right\rfloor\right) + 3 & \text{if } r > \left\lfloor\frac{s-1}{3}\right\rfloor, \\ 14(r-1) + 4(s - 3(r-1)) + 3 & \text{if } r \le \left\lfloor\frac{s-1}{3}\right\rfloor. \end{cases}$$

A schedule of this case is shown in Fig. 4. It is easy to verify that the derived formula is a correct upper bound for the execution time of this $T1$-$T2$-system.

If we set $b = 1$ and $d = 10$, we get a similar result, but in this case M_2 is iterated once and M_1 three times where M_2 is started during the first iteration of M_1. During the second and third iteration of M_1, M_2 is still executing basic block d.



T_2 p c v d p c v d p c v d p c v d p c ⋯
T_1 p a v b b b b b b b b b b p a v b b ⋯
t

Figure 4 A Simple Schedule.

5 Related Work

Multiple data-flow-based WCET analysis frameworks are discussed in [2]. In this paper we adopt a dataflow approach and extend it in order to support concurrent programs.

Our Kronecker algebra based approach can be used for further analysis. In [12] we showed how to detect deadlocks.

In terms of how we generate a graph model for concurrent programs the closest work to ours was probably done by Buchholz and Kemper [3]. It differs from our work as follows. Our approach uses RCFGs and semaphores to model concurrent programs. Buchholz and Kemper worked on generating reachability sets in composed automata. In addition, we propose lazy calculation of matrix entries to optimize running time. Both approaches employ Kronecker algebra.

In [8] a method based on model checking of multi-core applications modeled as timed automata is investigated. The tool box UPPAAL is used and synchronization is modeled by using spinlock-like primitives.

Although not closely related we recognize the work done in the field of *stochastic automata networks* which is based on the work of Plateau [13] as related work. Basic operators are shared and some properties of Kronecker algebra were integrated into this paper.

6 Conclusion and Future Work

We established a framework for WCET analysis of concurrent systems based on Kronecker algebra. Thread synchronization is modeled by semaphores. Our graph representation of multi-threaded programs plays a similar role for concurrent systems as control flow graphs do for sequential programs. Thus a suitable graph model for timing analysis of multi-threaded software has been set up.

We consider optimizations like partial order reduction (cf. [4]) as future work. A standardized benchmark suite (similar to [7]) including concurrency would enable comparison of different approaches.

References

1 R. Bellman. *Introduction to Matrix Analysis.* Classics in Applied Mathematics. Society for Industrial and Applied Mathematics, 2nd edition, 1997.

2 J. Blieberger. Data-Flow Frameworks for Worst-Case Execution Time Analysis. *Real-Time Systems*, 22(3):183–227, 2002.

3 P. Buchholz and P. Kemper. Efficient Computation and Representation of Large Reachability Sets for Composed Automata. *Discrete Event Dyn. Systems*, 12(3):265–286, 2002.

4 E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.

5 M. Davio. Kronecker Products and Shuffle Algebra. *IEEE Trans. Computers*, 30(2):116–125, 1981.

6 A. Graham. *Kronecker Products and Matrix Calculus with Applications.* Ellis Horwood Ltd., New York, 1981.

7 J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In *Proc. 10th International Workshop on Worst-Case Execution Time Analysis*, pages 136–146, 2010.

8 A. Gustavsson, A. Ermedahl, B. Lisper, and P. Pettersson. Towards WCET Analysis of Multicore Architectures Using UPPAAL. In *Proc. 10th International Workshop on Worst-Case Execution Time Analysis*, pages 101–112, 2010.

9 P. Henderson and J. H. Morris, Jr. A Lazy Evaluator. In *3rd ACM Symposium on Principles of Programming Languages*, POPL '76, pages 95–103, January 1976.

10 W. Kuich and A. Salomaa. *Semirings, Automata, Languages.* Springer, 1986.

11 G. Küster. On the Hurwitz Product of Formal Power Series and Automata. *Theor. Comput. Sci.*, 83(2):261–273, 1991.

12 R. Mittermayr and J. Blieberger. Shared Memory Concurrent System Verification using Kronecker Algebra. Technical Report 183/1-155, Automation Systems Group, TU Vienna, http://arxiv.org/abs/1109.5522, Sept. 2011.

13 B. Plateau. On the Stochastic Structure of Parallelism and Synchronization Models for Distributed Algorithms. In *ACM SIGMETRICS*, volume 13, pages 147–154, 1985.

14 V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. A New Framework for Elimination-Based Data Flow Analysis Using DJ Graphs. *ACM Trans. Program. Lang. Syst.*, 20(2):388–435, 1998.

15 R. E. Tarjan. A Unified Approach to Path Problems. *J. ACM*, 28(3):577–593, 1981.

A Time-composable Operating System*

Andrea Baldovin, Enrico Mezzetti, and Tullio Vardanega

University of Padua, Department of Mathematics,
via Trieste, 63 35121 Padua, Italy
`{baldovin,emezzett,tullio.vardanega}@math.unipd.it`

Abstract

Time composability is a guiding principle to the development and certification process of real-time embedded systems. Considerable efforts have been devoted to studying the role of hardware architectures – and their modern accelerating features – in enabling the hierarchical composition of the timing behaviour of software programs considered in isolation. Much less attention has been devoted to the effect of real-time Operating Systems (OS) on time composability at the application level.

In fact, the very presence of the OS contributes to the variability of the execution time of the application directly and indirectly; by way of its own response time jitter and by its effect on the state retained by the processor hardware. We consider *zero disturbance* and *steady behaviour* as those characteristic properties that an operating system should exhibit, so as to be time-composable with the user applications. We assess those properties on the redesign of an ARINC compliant partitioned operating system, for use in avionics applications, and present some experimental results from a preliminary implementation of our approach within the scope of the EU FP7 PROARTIS project.

1998 ACM Subject Classification C.3 [Computer Systems Organization]: Special-purpose and application-based systems – Real-time and embedded systems, D.4.1 [Operating System]: Process Management, D.2.11 [Software Engineering]: Software Architectures – Domain-specific architectures, Patterns

Keywords and phrases Real-time Operating System, Timing composability, ARINC

Digital Object Identifier 10.4230/OASIcs.WCET.2012.69

1 Introduction

The increasing complexity in the design, development and validation of real-time embedded systems can be tackled only is best responded by compositional, incremental software development. The other side of the coin in a compositional approach is that the properties ascertained for individual components in isolation should allow reasoning on the properties of the system that results from their composition (*composability*). Whereas compositionality and composability are consolidated concepts when looking at a system from a purely functional perspective, they are much more difficult to understand and to guarantee when applied to extra-functional concerns and to timing in particular [10]. By the principle of composability, in fact, the timing behaviour of a system should be simply determined as a summation over the execution times of its building blocks; moreover, by composability, a software module should exhibit the same timing behaviour independently of the presence and operation of any other component in the system. Unfortunately, even guaranteeing just timing compositionality on

* This work has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the grant agreement n 249100.

current hardware and software architectures is difficult. Although timing analysis frameworks typically characterise the timing behaviour of a system compositionally, a truly composable timing behaviour is not generally provided at the lower levels of a system. The main obstacle to time composability is that modern hardware architectures include a score of advanced acceleration features (e.g., caches, complex pipelines, etc.) that bring an increase in performance at the cost of a highly variable timing behaviour. Since those hardware features typically exploit execution history to speed up average performance, the execution time of a software module is likely to (highly) depend on the state retained by history-dependent hardware, which in turn is affected by other modules. The incurred dependence wrecks composability in the timing dimension as the execution history becomes a characteristic of the whole system and not that of a single component.

For timing compositionality and composability to hold, stringent constraints are imposed on how the system should be conceived and built, in both hardware and software dimensions [9]. Whereas several studies focused on the importance of hardware architectures in enabling compositional timing analysis [6], less attention has been devoted to the role played by other layers in the execution stack. Timing composability is in fact a system property that originates from the underlying hardware and must be preserved across other layers, including the operating system. In this paper we address the role of the operating system layer in preserving timing composability in Integrated Modular Avionics (IMA) systems, where timing composability is a fundamental assumption behind temporal and spatial isolation among software partitions. In particular, we report on our attempt in redesigning part of POK [2], an open-source ARINC653-compliant real-time kernel with a view to timing composability.

The remainder of this paper is organised as follows: in Section 2 we address time composability as a property within the abstraction layers of a typical architecture and discuss a possible approach to enable composability between OS and application layers. Section 3 explains how our approach has been implemented in a partitioned real-time kernel, while Section 4 provides experimental evidence to our arguments. Finally, Section 5 draws some conclusions.

2 Timing composability: a layered approach

Seeking incrementality in the development life-cycle advocates the adoption of an incremental development approach from the system perspective, where incrementality should naturally emerge as a consequence of guaranteeing composability to the elementary constituents (i.e., software modules) of the system. However, in practice, real-time software development can only strive to adopt such discipline, as the supporting methodology and technology are still immature.

We were given the opportunity to study the issues above in the context of the EU FP7 PROARTIS [3] project: this initiative aims at defining a novel, probabilistic framework for timing analysis of critical real-time embedded systems. As an industrial trait, the main project focus is set on partitioned applications, commonly encountered in avionics systems, and particularly on the IMA architecture and its ARINC 653 [1] incarnation. These industrial standards encourage the development of partitioned applications, where the concept of composability is strictly related to the fundamental requirement of guaranteeing spatial and temporal segregation of applications sharing the same computational resources within a federated architecture.

Hardware acceleration features, speculative execution and complex software architectures prevent systems from achieving a composable timing behaviour. Dependence on the execution history, in particular, is one of the main hurdles to timing composability. Removing or at least minimising the effects of history dependence is thus a reasonable approach to achieve a time-composable architecture. Breaking down the execution platform into three classic layers – the hardware layer (`HW PLATFORM`), an operating system layer (`KERNEL`) in the middle, and, finally, a user application layer (`APPLICATION SW`) running on top – makes it possible to address history independence and thus timing composability as a *bottom-up* property that must be first accomplished by the underlying hardware, then preserved across the OS primitives and services, and finally exhibited by the user application.

In this paper, we do not focus on HW-related composability issues, although we are perfectly aware of their relevance, especially with respect to the possible occurrence of timing anomalies [12]. We will assume, instead, the availability of a `HW PLATFORM` where interference from execution history on the timing behaviour has been proactively countered. This is not an unrealistic assumption since it can be achieved by means of simplified hardware platforms [6] or novel probabilistic approaches, as suggested in PROARTIS [3]. Conversely, we consider the `APPLICATION SW` layer, the space within which user applications run, the only level at which (application-logic related) timing variability should be allowed. Ideally, time-composable hardware and kernel layers should be able to remove all history-dependent timing variability so that state-of-the-art timing analysis approach should be able to account for the residual variability at the application level.

The `KERNEL` layer, which is the main focus of our investigation, actually provides an abstraction layer for the operating system primitives and services. From the timing analysability standpoint, the role played by this layer is possibly as important as that played by the `HW PLATFORM`. As a fundamental enabler to compositional analysis approaches, in fact, this layer should preserve the independence property exhibited by the underlying hardware and should not introduce additional sources of timing variability in the execution stack.

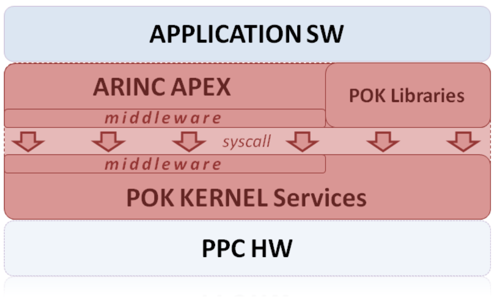


Figure 1 Structural decomposition of POK.

In the scope of our investigation, we focused on the PowerPC processor family, and the PPC 750 model [4] in particular, by reason of its widespread adoption in avionic platforms. We also selected POK [2] as our reference OS kernel because of its lightweight dimensions, its availability in open source and its embryonic implementation of the ARINC specification. We redesigned part of its services with a view to time-composability and analysability. Figure 1 shows a structural breakdown of the POK framework: the `KERNEL` layer provides an interface to a set of standard libraries (e.g., C standard library) and core OS services (e.g., scheduling primitives) to the `APPLICATION SW` layer. In addition, the POK kernel also provides an implementation of a subset of the ARINC Application Executive (APEX).

Enabling and preserving time-composability at the `KERNEL` layer poses two main requirements on the way an OS or ARINC service should be delivered:

- *Zero-disturbance*: in the presence of hardware features that exhibit history-dependent timing behaviour, the execution of an OS service should not have disturbing effects on the application. Some kind of separation is needed to isolate the hardware from the polluting effects of OS or ARINC services. The kind of hardware-level isolation that we seek can be provided by means of techniques similar to those adopted for cache partitioning [8]: a relatively small cache partition should be reserved for the OS so that the execution of OS services would still benefit from the cache acceleration but would not affect the cache state of the user code. However, implementing software cache partitioning (mapping of code to configure separate address spaces) in conjunction with a partitioned OS may result quite cumbersome in practice. An alternative (and easier to implement) approach consists in giving up any performance benefit and simply inhibiting all the history-dependent hardware at once when OS services are executed. This approach, however, comes at the cost of a relevant performance penalty that, though not being the main concern in critical real-time systems, could be still considered unacceptable. Also the execution frequency of a service is relevant with respect to disturbance: services triggered on timer expire (such as, for example, the PowerPC DEC interrupt handler) or an event basis can possibly have even more disturbing effects on the `APPLICATION SW` level, especially with respect to the soundness of timing analysis. The *deferred preemption* mechanism in combination with the selection of predetermined preemption points [13] could offer a reasonable solution for guaranteeing minimal uninterrupted executions while preserving feasibility.

- *Steady timing behaviour*: jittery timing behaviour of an OS service complicates its timing composition with the user-level application. Timing variability at the OS layer depends on a combination of multiple interacting factors: (i) the hardware state, as determined by history sensitive hardware features; (ii) the *software state*, as determined by the contents of its data structures and the algorithms used to access them; and, (iii) the input data. Whereas the first aspect can be treated similarly and contextually with the specular phenomenon of disturbance, the software state instead is actually determined by more or less complex data structures accessed by OS and ARINC services and by the algorithms implemented to access and manipulate them. The latter should thus be re-engineered to exhibit a constant-time – O(1) – and steady timing behaviour, like, for example, constant-time scheduling primitives (e.g., $O(1)$ Linux scheduler [7]). Besides the software state, the timing behaviour of an OS service may be influenced by the input parameters to the service call (so-called input data dependency). This is the case, for example, of ARINC IO services that read or write data of different size. This form of history dependence is much more difficult to attenuate as the algorithmic behaviour (e.g., application logic) cannot be completely removed, unless we do not force an overly pessimistic constant-time behaviour. We will get back to this issue in the next Section.

An OS layer that meets the above requirements is *time-composable* in that it can be seamlessly composed with the user-level `APPLICATION SW` without affecting its timing behaviour. In the following we present the implementation of a set of `KERNEL`-level services that exhibit a steady timing behaviour and do not disturb the timing behaviour of the user-level code. Our approach seeks for a general reduction in the effects of the OS layer on the application code and is expected to ease the analysis process, regardless of the timing analysis technique of choice.

3 Time-composable kernel layer

So far we reasoned on time composability between the OS and the user application layer. The original POK was not developed with time composability in mind, but rather aimed at the optimisation of the average-case performance. This section describes an alternative design and implementation aimed at injecting time composability in the POK framework. We start our discussion with the basic kernel design choices on time management and system scheduling an then proceed with considerations on some ARINC services we studied. In doing so, we refer to ARINC-specific concepts such as processes, partitions, scheduling slots, etc., whose detailed description is out of the scope of this paper: the interested reader is referred to [1]. Interestingly, similar ideas and solutions can be transposed to different execution platforms.

3.1 Time management

Time management, as one of the core OS services, is exploited by the operating system itself to perform back office activities, and by the application, which may have to program time-triggered actions. Most common time-management approaches adopted in real-time systems rely on either a tick counter or programmable one-shot timers. The original POK implementation provides a tick-based time management where a discrete counter is periodically incremented according to a frequency consistent with the real hardware clock rate[1]. Unfortunately, in tick-based approaches the operations involved in time management are periodically executed, regardless of the application logic; this is likely to incur timing interference on user applications, commensurate to the tick frequency.

For this reason we implemented a less intrusive time management mechanism based on interval timers, where clock interrupts are not necessarily periodic and can be programmed according to the specific application needs. Intuitively, a timer-based implementation can be designed to incur less interference in the timing behaviour of the user application as it guarantees that the execution of a user application is interrupted only when strictly required (i.e., partition switch, process activation, etc.). Making a step further, interval timers also enable to control and possibly postpone timing events at desired points in time and possibly in a way such that user applications are not interrupted. In particular, in an ARINC context we can program timers to expire only at partition switches, so that no overhead is introduced during application execution. Within each scheduling slot we enforce a variant of the fixed-priority deferred scheduling policy [13], in which preemption is enabled only at the end of a job (i.e., *run-to-completion* semantics).

3.2 Scheduling primitives

We implemented a lightweight constant-time – O(1) – fixed-priority scheduler exploiting an extremely compact representation of task states, that can be quickly updated through fixed-latency bitwise operations. In our implementation we assume all processes[2] defined in the same partition to have distinct priorities, to overcome the variability from linear-time insertion in FIFO priority queues. Since hard real-time operating systems typically define 255

[1] POK in its architectural-dependent implementation for PowerPC exploits the decrementer register and the TBU to periodically increment the tick counter.

[2] It should be noted that *process* is the ARINC equivalent of a *task* in classic real-time theory.

distinct priority levels, requiring distinct priorities poses no restriction on ARINC applications which are required to support up to 128 processes per partition [1].

Basically, we exploit a set of bit masks $MASK^{state}$, one for each state a process can assume (i.e., *dormant*, *waiting*, *ready* and *running* in ARINC speak), which collectively describe the current state of all application processes. A similar set of bit masks $MASK^{state}_{slot}$ is associated to each scheduling slot in a major frame, to describe process state changes. State updates are performed by bitwise OR-ing those masks. A simple priority-driven thread selection is done in a similar way by exploiting an ordered bitmask to represent priorities: selecting the runnable process with higher priority thus requires to identify the most significant bit in such mask. Such operation can be performed in constant time with built-in processor instructions (e.g., *count-trailing zeros* on PowerPC) or using perfect hashing with De Bruijn sequences [5].

Process activation events, however, can be dynamically programmed by the user application to occur within a scheduling slot, and thus outside of partition switches. This is the case, for example, when a synchronous kernel service requires a scheduling event to be triggered as a consequence of a timeout[3]. This kind of timeout can be used to enforce, for example, a timed self-suspension (i.e., with "delay until" semantics) or a phased execution of a process. Since we want to ensure that every process is run to completion, preemption is necessarily deferred at the end of process execution, which therefore becomes the next serviceable dispatching point, as shown in Figure 2; dispatching is performed using the same method presented above. A similar mechanism is used for aperiodic processes (i.e., sporadic tasks): in this case, the deferred scheduling event is triggered by a synchronous activation request, which does not involve the use of timers.

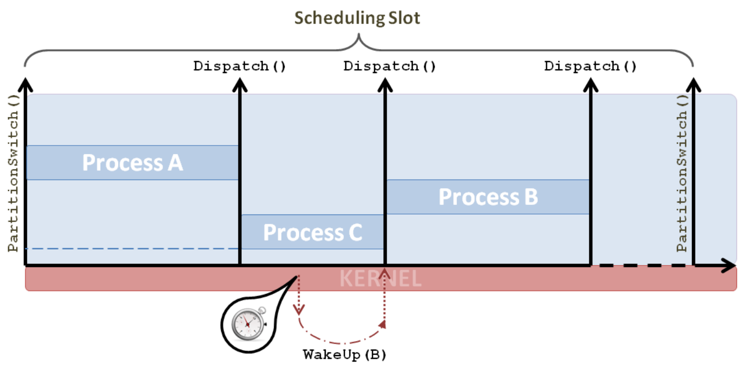


Figure 2 Deferred dispatching mechanism within a time slot.

3.3 Time-composable ARINC APEX

With respect to the subset of ARINC services we have implemented so far, the main timing-composability issues arise from the IO communication between partitions. The basic message-based communication mechanisms provided by the ARINC SAMPLING and QUEUING services is based on channels, as logical links between one source port and one or more destination ports. The timing-composability issues raised by IO services, either through

[3] The DELAYED_START and TIMED_WAIT ARINC services are representative examples of requests for a timed-out process activation.

sampling or queuing ports are mainly due to the variability induced by the amount of data to be read or written. Whereas ports are characterised by a maximum size, forcing the exchange of the maximum amount of data would obtain a constant-time behaviour at the cost of an unacceptable performance loss. Moreover, the potential blocking incurred by queuing port could further complicate the disturbing effects of inter-partition communication. Also the natural countermeasure of isolating the effects of the service execution on the hardware state cannot be seamlessly applied in this context. Inhibiting the caches for example is likely to kill performance since the read and write operations are inherently loop intensive and greatly benefit from both temporal and spatial locality.

To counter this unstable and disturbing behaviour we separate the variable (loop-intensive) part of the read/write services and accommodate such variability so that it incurs less disturbing effects on the execution of the application code. The concrete specification of an ARINC system typically takes a static configuration (e.g., configuration tables) that provides insightful information on the system functional behaviour. We exploit the available information on the inter-partition communication patterns to perform some sort of preventive IO in between partition switch, as depicted in Figure 3.

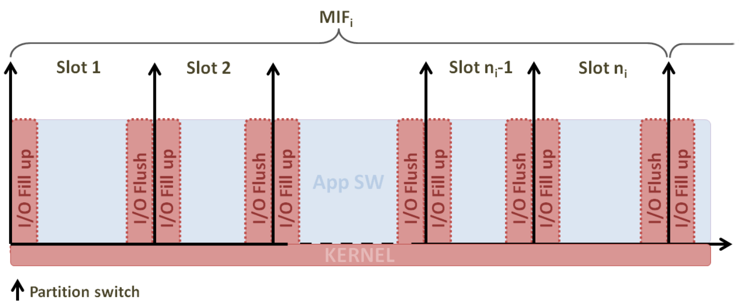


Figure 3 Inter-partition IO management.

We postpone all port writes to the slack time at the end of a partition scheduling slot. Similarly, we preload the required data into the destination partition in a specular slack time, at the beginning of a scheduling slot. The information flow is guaranteed to be preserved as we are dealing with inter-partition communication: (i) the state of all destination (input) ports is already determined at the beginning of a partition slot; (ii) the state of all source (output) ports is not relevant until the partition slot terminates and another partitions gets scheduled for execution. This way, we should not worry about the disturbing effects on the hardware state as no optimistic assumption should ever be made on partition switching; moreover, the input-dependent variability can be analysed within some sort of end-to-end analysis. We are currently implementing a similar approach with respect to intra-partition communication (i.e., via ARINC blackboards, buffers etc.).

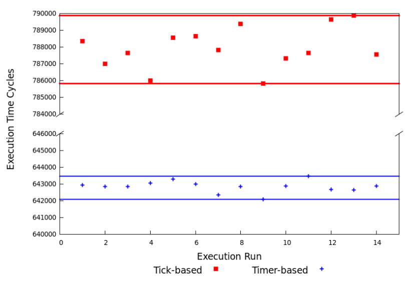
4 Experimental assessment

In our experiments we wanted to assess whether and to what extent our preliminary implementation of kernel primitives and services achieve time composability between OS and user application. We performed our analysis on the basis of the timing information collected by uninterrupted and consecutive end-to-end runs of software units at the granularity level of kernel primitives, ARINC services and application main procedures. Measurements were perfectly suited to meet our objectives as the set of properties we wanted to prove on the OS layer (steady timing behaviour and zero-disturbance) can be arguably assessed by means of a small number of selective examples. In fact, in the absence of history dependence, the timing behaviour of the analysed procedures rapidly fall into predictable behavioural patterns.

The PROARTIS Sim tool, a SocLib based simulator of a PowerPC 750 platform developed within the PROARTIS project, was used to collect timing traces that were later fed to RapiTime [11], a hybrid measurement-based timing analysis tool from Rapita Systems Ltd. The adopted simulator is highly configurable and has been designed to guarantee fixed-latency execution of each processor instruction, except for memory accesses whose latency depends on the current cache state. Since caches are the only residual source of history dependence we were able to exclude, when needed, any source of interference in the execution time by simply enforcing a constant response of the cache, either always miss (i.e., inhibition) or always hit (i.e., perfect cache). The simulator tracing capabilities allowed us to collect execution traces without actual software instrumentation, thus avoiding the so-called *probe effect*. The baseline PROARTIS Sim configuration in our experiments included the perfect cache option, which corresponds to enforcing the latency of a cache hit on every memory access. In the lack of a fine-grained control over the cache behaviour, this parametrisation was meant to exclude the variability stemming from caches without incurring the peformance penalty of thoroughly disabling them. According to our overall approach, in fact, the majority of our experiments address those services that are executed outside of the user application and there is no need to execute them with acceleration features disabled. It is worth noting that the raw numbers obtained under the always hit option are directly proportional to those obtainable under an always miss policy; thus, providing both would not add to our reasoning.

Our experiments were conducted over a relevant set of OS services, which we considered to be the most critical ones from the timing composability standpoint: *time management*, *scheduling primitives*, and *sampling port communication*. All of them were measured under different inputs or different task workloads (i.e., for kernel primitives).

We wanted to first measure whether and to what extent the basic time-management primitives may affect the timing behaviour of a generic user application. We evaluated first the performance of a selective application within the original POK implementation, which uses the decrementer register as a tick counter. Subsequently, we set up a new scenario where no interference arises from the time management service, as the latter was implemented by interval timers set to fire outside the execution boundaries of the examined procedure. Caches have been enabled for this experiment and configured with Least Recently Used (LRU) replacement policy.



	MinET	MaxET	Delta
Tick-based	785825	789888	4063
Timer-based	642076	643472	1396

Figure 4 Execution under tick-based and interval-timer time management.

The experimental results shown in Figure 4 are not surprising. The tick-based time management mechanism (upper band in the plot) should be discarded in favour of the interval timer, since its disturbance on the application code is clearly higher, due to the set of useless time-management activities performed on every tick. Interestingly, as highlighted by the different areas between the straight lines, the cache-induced variability experienced by the application under a tick-counter policy is considerably greater than that suffered under interval-based timer, as a consequence of increased pollution of cache states.

Moving on to scheduling primitives, we observe that inattentive implementation and design choices may affect both the latency and jitter incurred by scheduling primitives such as partition switch, process dispatching or state update. To provide experimental evidence of the steady timing behaviour of our implemented scheduling primitives, as opposed to the standard ones in the original version of POK, we focus on task status update and task election. These activities are performed in a single operation in tick-based approaches, whereas they execute separately in our approach: this is because status update is performed only at partition switch, whereas thread dispatching occurs at the end of every job execution, according to the run-to-completion semantics. We enforced a perfect cache behaviour so that no overhead from the hardware is accounted for in measured execution times. We also concocted our experiments to follow a strictly deterministic periodic pattern, which allowed us to restrain our observations to a limited number of runs. Figure 5 shows observed execution times for the thread selection routine (that is part of the larger scheduling primitive). The workload in the top chart is two partitions, with three and two threads respectively, while the bottom chart reports a larger example comprises three partitions with ten, five and two threads each.

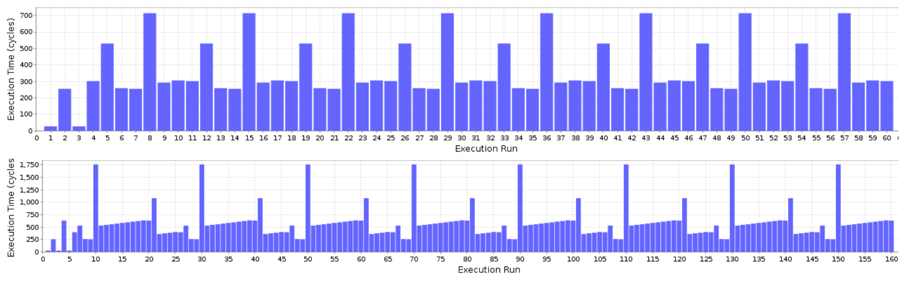


Figure 5 FPPS thread selection under different workloads.

The original POK scheduler always performs the same operations at every clock tick, mainly checking whether the current partition or thread should be switched to the next ones. Under those premises, two potential sources of variability originate from the possibility that a partition/thread needs to be actually switched at a particular scheduling point, and from the number of threads to be managed in the executing partition, respectively. The graphs in Figure 5 illustrate this situation clearly: higher peaks correspond to partition switches, when the state of all threads in the new partition changes to ready and they must be therefore inserted in the appropriate scheduler queues. For our constant-time scheduler, instead, we must distinguish two cases, since its behaviour is different at partition and thread switch. Figure 6 shows the execution time of the routine invoked at partition switch, which only needs to update thread states[4]. Though the settings are exactly the same as Figure 5 above, status updates are performed in constant time thanks to the bitwise operations on thread masks (Section 3.2).

Figure 7 shows that our constant-time scheduler is capable of detecting the highest-priority thread to be dispatched with fixed overhead, by using De Bruijn sequences. Lower

[4] Except for inter-partition communication overhead.

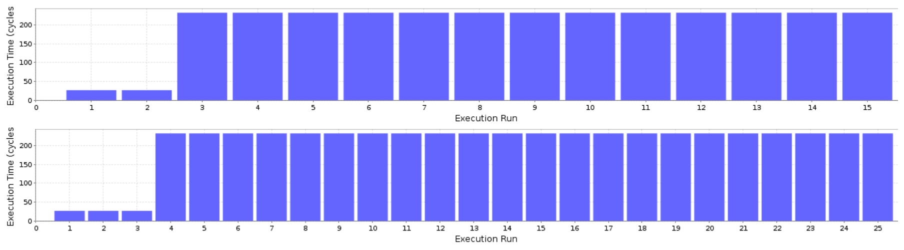


Figure 6 Constant-time thread status update under different workloads.

peaks in Figure 6 correspond to the selection of the system idle thread. From the raw numbers, reported in Table 1, we note that the small delta exhibited by our thread switch implementation is actually due to the difference between the selection of any thread (95) and the idle thread[5] (45). The delta measured on the standard POK implementation, instead, represents real jitter.

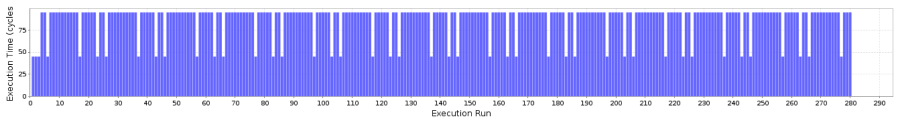


Figure 7 Constant-time thread selection in a test case with three partitions and seventeen threads.

Table 1 Execution times for a user application with tick-based and interval-timer scheduling.

	FPPS (standard POK)			O(1) scheduler (partition switch)			O(1) scheduler (thread switch)		
	Min	Max	Delta	Min	Max	Delta	Min	Max	Delta
2 partitions 5 threads	255	714	459	232	232	0	45	95	50
3 partitions 17 threads	259	1759	1500	232	232	0	45	95	50

When it comes to ARINC APEX, we focused on inter-partition communication to start with. Inter-partition communication via sampling ports have been specifically redesigned for the sake of time composability[6]: our implementation is based on *posted* writes and *prefetched* reads that permits to remove the sources of variability and disturbance from the service itself and serve them out at partition switch. We forced the simulator to resemble a perfect cache when measuring this services as we wanted to exclude the variability stemming from the cache behaviour without incurring the peformance penalty of a disabled cache: a positive effect of relocating the message management in between the execution of two partitions is in fact that of being able to exploit the caches without any side-effect on the user application. Having excluded the cache variability causes the analysed service to exhibit a steady timing (actually constant) behaviour where the execution time only varies as a function over the input (message) size. We triggered the execution of the sampling services with different sizes of the data to be exchanged. The dependence of READ and WRITE services on the input size

[5] The idle task is elected for execution when no other task is runnable.
[6] The implementation of a similar mechanism for queuing ports is work in progress at the time of writing.

is shown in Table 2: the increase in the execution time of each service is related to an increase in the input data size, here ranging from 1 to 256 Bytes. The redesigned implementations of both services (newWRITE and newREAD in Table 2) are instead constant, as the invocation of the services themselves does actually execute neither a read nor a write operation, whose execution is instead deferred at the begin and end of a partition switch respectively.

Table 3 shows the partition switch overhead (observed under different input sizes) that is the penalty that has to be paid for relocating the message passing mechanism on partition switch. From what we observed in our experiments, the incurred time penalty is quite limited and, more importantly, when summed to the time previously spent in the READ or WRITE service, it does not exceed the execution time of the standard implementation with the same input.

Table 2 Execution times for the READ and WRITE services.

	WRITE	NewWRITE	READ	NewREAD
1B	523	436	618	448
4B	580	436	794	448
32B	1112	436	1383	448
64B	1720	436	1758	448
96B	2024	436	2086	448
128B	2936	436	2974	448
256B	5368	436	5406	448

Table 3 Maximum observed partition switch overhead.

	Partition Switch (standard)	Read+Write Overhead
32 B	27674	+ 661
64 B	29498	+ 1269
96 B	32224	+ 1973
128 B	33146	+ 2485
192 B	37686	+ 3807
256 B	41619	+ 5118
384 B	48630	+ 7455

5 Conclusion

Composability in the time dimension is a fundamental enabler for the hierarchical decomposition of large complex systems into smaller, tractable units. Whereas hardware platform are widely acknowledged to have great influence on the timing composability, in this paper we focus the role of the real-time operating system in enabling timing composability in IMA systems and identified the properties that make an operating system timing-composable with user applications. In that light, we redesigned a real-time partitioned kernel and provided experimental evidence that the degree of time composability may greatly benefit from proper design choices in the implementation of the operating system.

References

1 APEX Working Group. Draft 3 of Supplement 1 to ARINC Specification 653: Avionics Application Software Standard Interface. 2003.

2 Julien Delange and Laurent Lec. POK, an ARINC653-compliant operating system released under the BSD license. *13th Real-Time Linux Workshop*, 10 2011.

3 F.J. Cazorla et al. PROARTIS: Probabilistically analysable real-time systems. *ACM Transactions on Embedded Computing Systems*, to appear.

4 Freescale. PowerPC 750 Microprocessor, 2012. `https://www-01.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_750_Microprocessor`.

5 Charles E. Leiserson, Harald Prokop, and Keith H. Randall. Using de Bruijn Sequences to Index a 1 in a Computer Word, 1998.

6 Isaac Liu, Jan Reineke, and Edward A. Lee. A PRET Architecture Supporting Concurrent Programs with Composable Timing Properties. In *44th Asilomar Conference on Signals, Systems, and Computers*, pages 2111–2115, November 2010.

7 Ingo Molnar. Goals, Design and Implementation of the new ultra-scalable O(1) scheduler, Jan. 2002. Available on-line at `http://casper.berkeley.edu/`, visited on April 2012.

8 F. Mueller. Compiler support for software-based cache partitioning. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, 1995.

9 Peter Puschner, Raimund Kirner, and Robert G. Pettit. Towards composable timing for real-time software. In *Proc. 1st International Workshop on Software Technologies for Future Dependable Distributed Systems*, Mar. 2009.

10 Peter Puschner and Martin Schoeberl. On Composable System Timing, Task Timing, and WCET Analysis. In *Proc. of the 8th Int. Workshop on WCET Analysis*, 2008.

11 Rapita Systems Ltd. Rapitime, 2012. `http://www.rapitasystems.com/rapitime`.

12 J. Reineke et al. A definition and classification of timing anomalies. In *WCET*, 2006.

13 Gang Yao, Giorgio C. Buttazzo, and Marko Bertogna. Feasibility analysis under fixed priority scheduling with limited preemptions. *Real-Time Systems*, 47(3):198–223, 2011.

Analysis of WCET in an experimental satellite software development*

Jorge Garrido, Daniel Brosnan, Juan A. de la Puente, Alejandro Alonso, and Juan Zamorano

Real-Time Systems group (STRAST)
Universidad Politécnica de Madrid (UPM), Spain
`str@dit.upm.es`

Abstract

This paper describes a case study in WCET analysis of an on-board spacecraft software system. The attitude control system of UPMSat-2, an experimental micro-satellite which is scheduled to be launched in 2013, is used for an experiment on analysing the worst-case execution time of code automatically generated from a Simulink model. In order to properly test the code, a hardware-in-the-loop configuration with a simulation model of the spacecraft environment has been used as a test bench. The code has been analysed with RapiTime, with some modifications to the original instrumentation routines, in order to take into account the particularities of the test configuration. Results from the experiment are described and commented in the paper.

1998 ACM Subject Classification C.3 Real-time and embedded systems

Keywords and phrases Real-time systems, embedded systems, timing analysis, WCET calculation

Digital Object Identifier 10.4230/OASIcs.WCET.2012.81

1 Introduction

UPMSat-2 is a project aimed at developing an experimental micro-satellite that can be used as a technology demonstrator for several research groups at UPM, the Technical University of Madrid. The Real-Time Systems Group at UPM (STRAST)[1] is responsible for designing and building all the on-board and ground-segment software for the satellite. The software is being coded in Ada with the Ravenscar profile tasking restrictions [4], and runs on a LEON3 [9] computer board. The GNATforLEON compilation chain [14], including the Open Ravenscar Real-time kernel (ORK) [5], is being used for software development.

Software standards for on-board spacecraft software [8, 7] require schedulability analysis to be used in the verification process. This kind of analysis, in turn, requires the worst case execution time (WCET) of each task to be known. Therefore, WCET measuring methods and tools [16] must be used as a first step in performing timing analysis on embedded on-board systems.

This paper describes the approach that the authors have taken to calculate the WCET of the UPMSat-2 Attitude Determination and Control System (ADCS) subsystem. The ADCS functional code has been automatically generated from a Simulink[2] engineering model, and

* This work has been partly funded by the Spanish Ministry of Economy and Competitiveness (MINECO), project TIN2011-28567-C03-01 (HI-PARTES).
[1] `www.dit.upm.es/str`
[2] *Simulink* is a registered trade mark of The MathWorks Inc.

then integrated with concurrent, real-time container code following a model-driven approach. The WCET of the resulting code has been analysed using RapiTime[3], a well-known tool for hard real-time systems analysis.

The rest of the paper is organised as follows: Section 2 describes the ADCS subsystem and its relationship to other components of the UPMSat-2 software. Section 3 presents the methodological approach to WCET analysis and the details of the analysis process. Section 4 summarizes the results obtained so far. Finally, section 5 presents the conclusions of the analysis and some ideas for future work.

2 System description

2.1 Hardware platform

All the computer-related functions on board of the UPMSat-2 satellite will be executed on a single computer platform, called the On-Board Computer (OBC) [6]. The OBC hardware will be based on a LEON computer with SRAM main memory and a solid state disk (SDD), as well as a number of peripherals for interaction with the satellite sensors and actuators.

The LEON family of processors[4] is a 32-bit synthesizable VHDL processor core that implements the SPARC V8 architecture [15]. The flight version of the OBC, which will be based on LEON3, is still under development. For this reason, an engineering model has been used for this experiment. The engineering model is based on a GR-XC3S1500 Spartan3 development board[5] with a LEON2 processor at 40 MHz clock frequency and 64 MB of SDRAM. Cache memory is not used in this implementation. The main difference between the LEON2 processor used in the engineering model and the envisaged production LEON3 are that the latter has a 7-stage pipeline instead of the 5-stage pipeline of LEON2. Other differences, such as the presence of an MMU and SMP support in LEON3, are not significant as these features are not used in this project. Since the purpose of this work is to validate the WCET calculation methodology, it can be assumed that the engineering model is a representative instance of the flight computer.

2.2 Software architecture

The on-board software system is composed of the following subsystems (figure 1):

- *Platform monitoring*: this subsystem is in charge of assessing the state of the satellite by periodically reading housekeeping sensor data (battery and solar panel voltages and currents, temperatures at various points of the spacecraft, etc.)
- *Telemetry and telecommand (TMTC)*: this subsystem interacts with the telecommunications hardware (TMC) in order to send messages (telemetry) to a ground-based station and receive telecommands from it.
- *Attitude determination and control system (ADCS)*: this subsystem computes the orientation (attitude) of the satellite with respect to the Earth, and takes corrective actions in order to keep it within the specified values. The attitude is derived from 3-axis measurements of the Earth magnetic field made with special sensors called *magnetometers* (MM). Control actions are performed by means of *magnetorquers* (MT), which are electromagnets

[3] `http://www.rapitasystems.com/rapitime`
[4] `http://www.gaisler.com/leonmain.html`
[5] `http://www.pender.ch/products_xc3s.shtml`

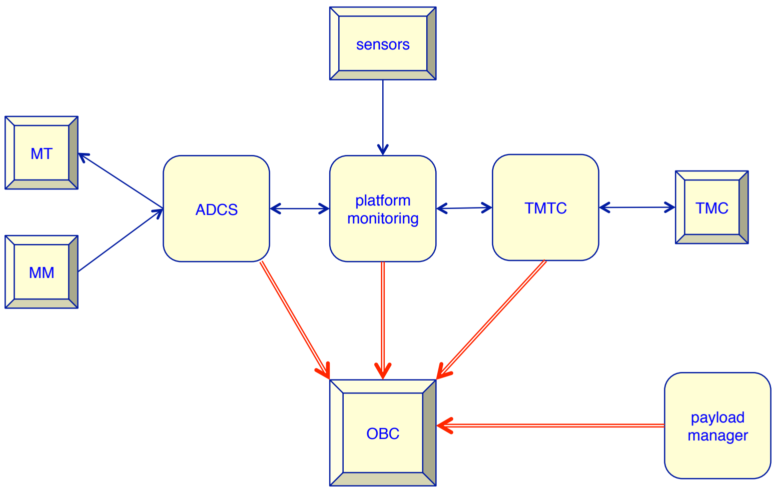


Figure 1 UPMSat-2 software architecture.

that can develop a magnetic field that is used to change the attitude of the satellite as needed.

In addition to the above main subsystems, there is another software component which manages the operation of the satellite payload. In this case, the payload consists of a series of experiments to be performed when ordered by specific telecommands.

The code for the UPMSat-2 software is written in Ada [1], and is being produced using a model-driven approach which was first developed in a previous project [11]. Under this approach, a software system is made of a number of components with well-defined interfaces. The internals of a component have three main parts, which implement separation of concerns between the functional, concurrent, and synchronisation aspects of the component. In particular, functional code generated with high-level modelling tools such as Simulink can be incorporated in the functional part of a component, leaving the complexity of concurrency, timing and synchronization to the other parts (figure 2). Concurrency and synchronization are implemented by Ada tasks as restricted by the Ravenscar profile, i.e. the number of tasks if fixed, the scheduling method is fixed-priority pre-emptive scheduling (FPPS), and communication among tasks is limited to shared data objects accessed with an immediate ceiling priority protocol (ICPP) [4]. The concurrent constituents of components are automatically generated from a reduced set of archetypes [3, 12]. Since the code of the concurrent elements is derived from a few simple patterns, the key issue for analysing the timing behaviour of the system is to get accurate estimates of the WCET of the functional code. This code can be rather complex, depending on the particular functions that are executed by each component. In the following, we will centre on the functional code of the ADCS subsystem as a representative example of the issues involved.

2.3 The Attitude Determination and Control System (ADCS)

The attitude of the satellite is represented by three angles measuring the orientation of the spacecraft with respect to an Earth-based reference frame. The attitude angles are computed from the readings of three magnetometers, which measure the intensity of the Earth magnetic field on the platform reference axes. There are also three magnetorquers that are used as actuators in order to adjust the attitude to the required angle values.

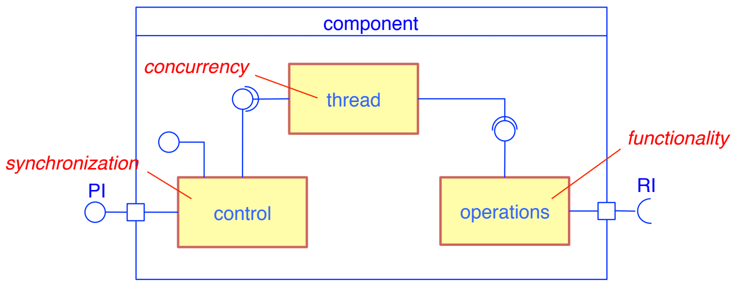


Figure 2 Structure of a software component.

The angular orientation of the satellite may change due to perturbations originated at the environment, mainly by the fluctuation of the Earth magnetic field, but also by the drag of residual atmosphere and the solar radiation pressure. There are three magnetorquers that are used as actuators in order to adjust the attitude to the required angle values in the presence of perturbations. A control algorithm is used to compute the appropriate values for the magnetorquers intensity signals.

The attitude control algorithm is designed by aerospace engineers based on a mathematical model of the spacecraft dynamics and the torque perturbations. Due to the complexity of the model, a simulation model has been created using Simulink in order to design, test and validate the structure of the control algorithm and to tune its parameters to the most appropriate values.

Figure 3 shows the general structure of the simulation model. The blocks represent the spacecraft dynamics and the environment elements that are relevant for the design, including the Earth magnetic field, environmental perturbations, and the changes in the gravitational acceleration. The variables in the model represent different torques acting on the spacecraft, the quaternions representing the attitude of the satellite with respect to a vertical reference frame, the rotation matrix from the vertical reference frame to the satellite reference frame ($\mathbf{C}_{\mathrm{BV}}$), and the angular velocity of the body with respect to the vertical reference frame ($\mathbf{pqr}$).

The block labelled "controller" represents the control algorithm, which has as inputs the measurements of the magnetic field provided by the magnetometers ($\mathbf{B}_{\mathrm{b\_T}}$) and its derivatives ($\mathbf{B}_{\mathrm{dot}}$). The outputs of the control block are the intensity supplied to the magnetorquers ($\mathbf{I}_{\mathrm{A}}$) and the corresponding actuator torques ($\mathbf{T}_{\mathrm{Nm}}$). It must be noted that the sensor and actuator models are included in the controller model.

The functional code of the attitude controller is automatically generated using the Simulink code generation tools. In order to do this, and to properly model the interface elements as well, a discrete-time model of the attitude controller is required. While a discrete-time model could be obtained from a continuous-time controller model, a direct approach using discrete-time blocks in the whole model has been followed. A sampling frequency of 1Hz has been used for this purpose, based on the control engineers' knowledge of the spacecraft dynamics.

The current version of the controller functional code is comparatively simple, with no loops or conditional control structures. It has about 50 lines of C code, which are integrated with the concurrent and control constituents of the ADCS subsystem by means of the Ada interface facilities. The code implements the well-known PID (proportional-integral-derivative) control algorithm [2]. Further developments in the control algorithm are expected to produce longer and more complex code.

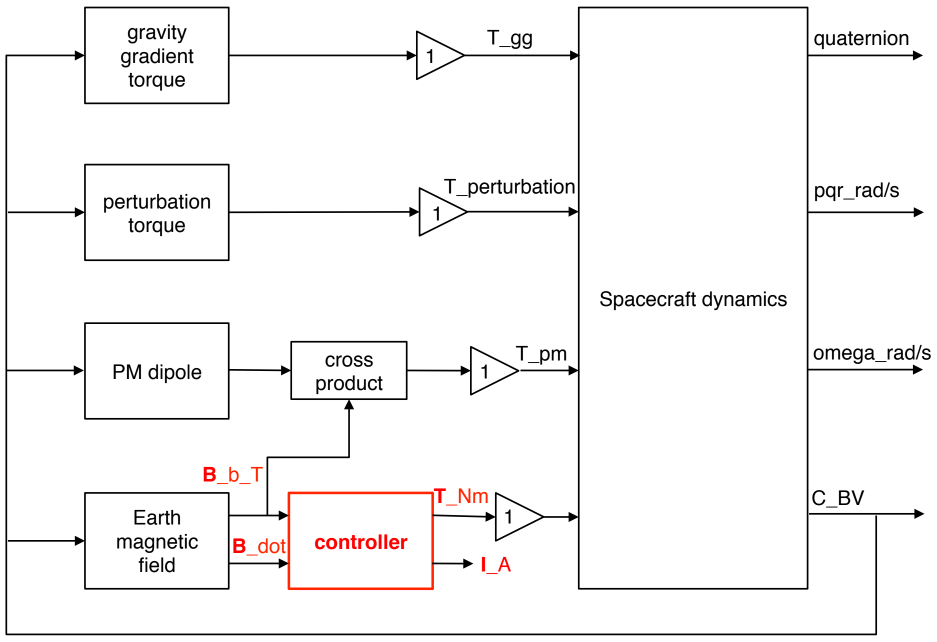


Figure 3 ADCS simulation model.

3 Analysis of the ADCS execution time

3.1 Object of the analysis

The code that has been analysed is the ADCS functional code automatically generated by the above described simulation model. While using a purely static code analysis could be feasible, a measurement-based approach has been used in order to get accurate estimates on the WCET on real hardware [16]. The hardware platform is the engineering version of the OBC that was introduced in section 2.1. As it has already been discussed, this platform is representative of the final flight computer hardware which is still to be built.

An important aspect of the analysis is the software context in which the code runs. In this work we have chosen to execute the ADCS functional code on its own, with only the underlying ORK kernel in place. The only active interrupt source is the system clock interrupt, which has a fairly low frequency, about 0.59Hz. The execution of the interrupt handler would only add a few tens of microseconds to the measured value of execution time in the rare case that the interrupt occurs while the ADCS task is executing.

It could be argued that measuring the execution time in isolation does not account for cache interference and other effects that may result from preemption in a concurrent execution environment. Since the current configuration of the OBC does not include a cache, we can still consider that executing the code in isolation provides realistic measurements of the execution time, at least for the engineering model. Further development of the flight version of the OBC are open to using the LEON3 cache, though, which should be taken into account in future measurements. Another question is the possible effects of control flow breaks on the processor pipeline. The SPARC V8 architecture does not perform branch prediction, using delayed branching instead. Therefore, in this case such effects can be safely ignored.

3.2 Test environment

As it frequently happens in real-time embedded system projects, it is not possible to test the satellite software in its real environment. Hardware-in-the-loop (HIL) techniques provide a useful testing framework in such cases. The basic idea is to test the embedded system against a simulation model instead of the real environment. The test environment is built from the simulation model depicted in figure 3 above, replacing the controller block by the real computer and controller software. Some additional components have been included as well, in order to model the sensors and actuators that carry out the interaction between the computer and the modelled environment.

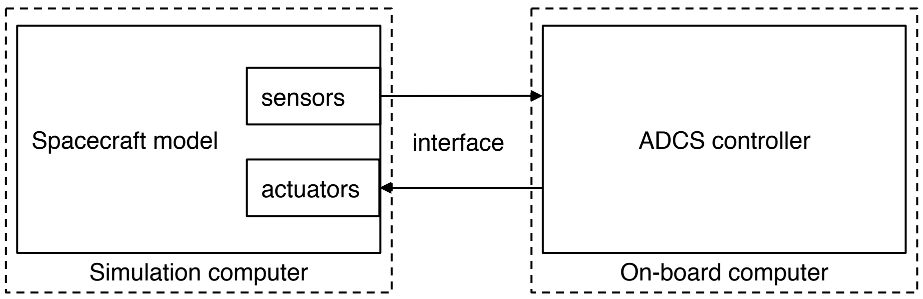


Figure 4 ADCS test configuration.

Figure 4 shows the architecture of the ADCS test configuration. The simulation model is implemented in Simulink on a PC-based platform, whereas the system under test is the ADCS code running on an engineering version of the satellite computer board. The interface between them is built on a serial line during the first stages of the development cycle, to be replaced with actual analog lines as the OBC implementation is advanced enough. This interface is modelled by replacing the original controller model (controller block in figure 3) with two new blocks: one than sends magnetic field data from the simulation model to the computer by means of the serial line, and another one that receives actuator data from the computer and inputs them to the attitude simulation block.

3.3 WCET measurement

We have chosen RapiTime[6] for WCET measurements as we are interested in on-target execution time measurements, using the above described test configuration. The version used is RVS 3.0.

RapiTime collects execution traces to generate time measurement statistics, among which worst case execution time measurements as required for schedulability analysis. In order to do this, RapiTime analyses the structure of the source code and adds instrumentation points at relevant places. The instrumentation points are calls to a procedure that records the execution time at which the point is reached. The instrumented code is repeatedly executed long enough to acquire relevant statistical information about the executions. Once this is done, the generated trace is checked for the correct format and compliance with previously extracted structural information, and a statistics report is generated.

In our case, the ADCS functional code has been instrumented with the RapiTime tools. In a first step, only the auto-generated controller code has been analysed, leaving out the

[6] `http://www.rapitasystems.com`

Listing 1 Instrumentation procedure

```
procedure Ipoint ( Id : Natural ) is
begin
   trace_buffer(trace_idx).ident := Id ;
   trace_buffer(trace_idx).timestamp := Ada.Real_Time.Clock ;
   trace_idx := trace_idx + 1;
end Ipoint;
```

interface code that is required for reading the magnetometers, which is simulated in the test platform.

The default setup for the RapiTime instrumentation, in which the trace is output to a serial line during the execution, is not suitable for our test platform, due to the bias that the serial communication would add to the measurement. Therefore, we changed the instrumentation code so that it writes the traces to an array instead. The trace elements are pairs of system time values and instrumentation point identifiers. The `Ada.Real_Time.Clock` function provided by ORK was used for system time measurements. The clock granularity is 100ns, and the time type representation is 64 bits long. RapiTime uses natural number values as instrumentation point identifiers, which were mapped to 32 bit Ada integers. Consequently, trace values use 12 bytes for each instrumentation point. The trace array is stored in the target computer memory, and later extracted to the developer workstation with the GRMON[7] debugging tool. Listing 1 shows the code of the instrumentation procedure.

In order to have enough data for the results to be relevant, the simulation should cover at least one complete orbit. The reason for it is the need to cover different values of the Earth magnetic field at different orbital positions. Since the devised orbital period for the satellite is about 1.64 hours, 5930 iterations of the control method have to be executed to cover a full orbit at the specified 1Hz rate. The size of the trace array, considering that there are two instrumentation points for each iteration, is thus 140 KB, which can be easily stored in the 64 MB RAM of the target computer.

The final step is to process the data on the development workstation. The results are described in the next section.

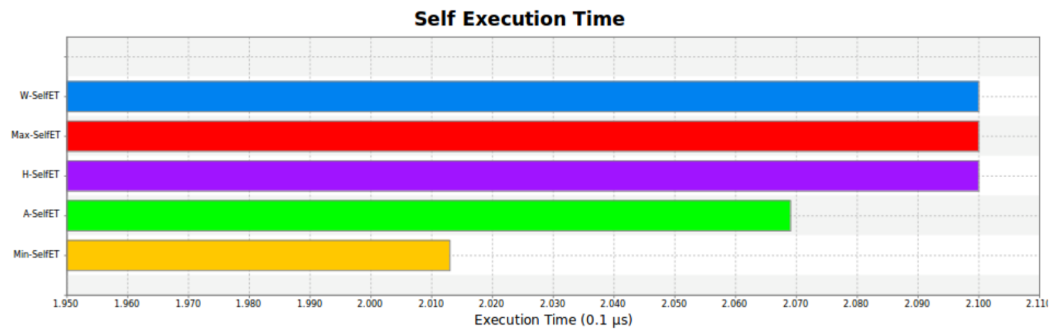
4 Results

In addition to WCET measurements, RapiTime generates a large amount of data about the ADCS code. The most relevant data refer to code coverage. RapiTime reports the number of times that each instrumentation point has been reached, which can be easily used to derive the execution count for every possible execution path.

For our purposes, the main results refer to statistics of time measurements. Figure 5 shows the minimum (Min-ET), average (A-ET), high water mark (H-ET), and maximum (Max-ET) execution time registered for the ADCS controller function, together with an estimate of the worst case execution time (W-ET) [13]. The term “Self” in the figure means that subprogram calls are not included, which is all right in this case as the code user test does not make any such calls. WCET estimation, which does not necessarily match the worst execution time registered in the experiments, is one of the features that make this tool

[7] `http://www.gaisler.com/`

really powerful for systems validation. “Over” refers to measurements including function calls. “Freq” refers to the number of times the code has been executed during the test.



Function »	Min-ET »		Avg-ET »		H-ET »		Max-ET »		W-ET »		
Name	Min-Freq	Min-OverET	A-Freq	A-OverET	H-Freq	H-OverET	Max-Freq	Max-OverET	W-Freq	W-OverET	#Tests
⊞ tasking.adcs_control	5.930	2.013	5.930	2.069	5.930	2.100	5.930	2.100	5.930	2.100	5.930

Figure 5 Execution time results for the ADCS control function.

In our case, the WCET estimate has a value of 2100 (210μs). Since the controller task period is 1s, the processor utilization is 0.21%, which is a fairly low value leaving time for more complex control algorithms if needed, and for the rest of the subsystems as well.

As can be seen from figures 5 and 6, the difference between the minimum and the worst case execution time is just 8.7μs, and the mode is 207μs with a probability of 0.147. With such a small difference, the distribution spans a very narrow range. However, the most interesting point is that, although the actual WCET might be greater than the estimated value of 210μs, the probability of such a situation is very low, $1.69 \cdot 10^{-4}$. We can take this result as an indication that we can safely use the estimated WCET for response time analysis.

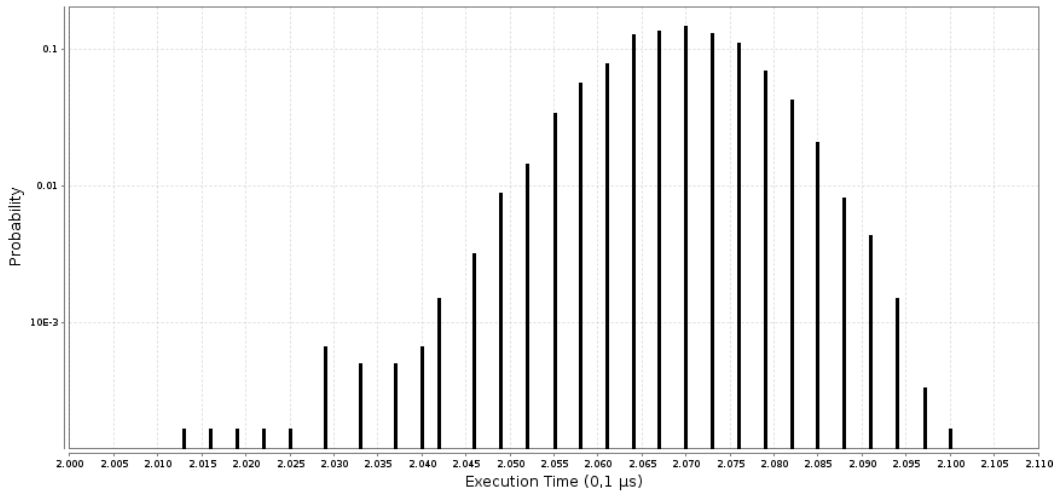


Figure 6 Probability distribution of execution time.

5 Conclusions and future work

The UPMSat-2 project provides an excellent testbed for many of the results obtained by UPM researchers over the recent years. In spite of being an experimental satellite, it is still a relevant project in terms of size and complexity. In particular, using the ORK technology developed by the real-time systems group, together with timing analysis methods such as implemented by RapiTime and other tools, provides an opportunity to validate the technology on a real satellite mission and consolidate our approach to embedded software development.

Hardware in the loop, supported by RapiTime and Simulink tools, has proved to be a successful approach to WCET analysis in terms of time, cost and results. Accordingly, it will be one of the techniques used for the final validation of the attitude determination and control system of the UPMSat-2, as well other subsystems of the spacecraft on-board software. Although the current implementation of the controller function is very simple and consequently does not require some of the advanced WCET techniques implemented by the tool, the experiment has shown that the testing infrastructure is valid and can be used to analyse the execution time of other, more complex functions as the development advances.

Future work includes adding real sensors and actuators to the test environment. This will allow us to get real and complete timing measurements on the whole system and not just the auto-generated code. The next step is to perform measurements with all the OBC subsystems in place, and use response time analysis tools to estimate the time behaviour of the full system, including interference among different tasks and blocking due to access to common resources. This may require optimizing the instrumentation routine in order to get a more efficient implementation of measurements.

The flight version of the OBC will include a LEON3 processor with cache memory. The issue of the effects of multitasking on cache affinity will have to be further investigated. A possible way to improve timing predictability may be freezing the cache during the execution of interrupt handlers, so that the interrupted task preserves its cache affinity, at the possible cost of increasing the execution time of interrupt handlers. It is also possible to disable the cache during the execution of critical application tasks, making the cache control registers part of the task context. This technique has been implemented in an extension to ORK that has been used in a related project [10].

On-going work also includes higher-level modelling of the on-board software using some of the tools developed in the CHESS project.[8]

Acknowledgements

The authors wish to acknowledge the collaboration of the rest of the UPMSat-2 team, especially Assal Farrahi and Javier Cubas from the “Ignacio Da Riva” institute (IDR/UPM).

References

1 *ISO/IEC 8652:1995(E)/TC1(2000)/AMD1(2007): Information Technology — Programming Languages — Ada.*

2 Karl Johan Åström and Tore Hägglund. *Advanced PID Control.* ISA - The Instrumentation, Systems, and Automation Society, Research Triangle Park, NC 27709, 2005.

[8] CHESS (Composition with Guarantees for High-integrity Embedded Software Components Assembly) is an 7FP project funded under the Artemis JTU.

3 Matteo Bordin and Tullio Vardanega. Automated model-based generation of Ravenscar-compliant source code. In *Proc. 17th Euromicro Conference on Real-Time System*, ECRTS '05, pages 59–67, Washington, DC, USA, 2005. IEEE Computer Society.

4 Alan Burns, Brian Dobbing, and Tullio Vardanega. Guide for the use of the Ada Ravenscar profile in high integrity systems. *Ada Letters*, XXIV:1–74, June 2004.

5 Juan A. de la Puente, José F. Ruiz, and Juan Zamorano. An open Ravenscar real-time kernel for GNAT. In Hubert B. Keller and Erhard Plödereder, editors, *Reliable Software Technologies — Ada-Europe 2000*, number 1845 in LNCS, pages 5–15. Springer-Verlag, 2000.

6 Juan A. de la Puente, Juan Zamorano, Alejandro Alonso, and Daniel Brosnan. A real-time computer control platform for an experimental satellite. In *Jornadas de Tiempo Real — JTR-2012*, 2012. Available at `http://www.ctr.unican.es/jtr12/programme.html`.

7 European Cooperation for Space Standardization. *ECSS-E-ST-40C Space engineering — Software*, March 2009. Available from ESA.

8 European Cooperation for Space Standardization. *ECSS-Q-ST-80C Space Product Assurance — Software Product Assurance*, March 2009. Available from ESA.

9 Gaisler Research. *LEON3 Product Sheet*, 2008.

10 Enrico Mezzetti, Adam Betts, José Ruiz, and Tullio Vardanega. Cache-aware development of high-integrity systems. In Jorge Real and Tullio Vardanega, editors, *Reliable Software Technologiey –– Ada-Europe 2010*, volume 6106 of *Lecture Notes in Computer Science*, pages 139–152. Springer Berlin / Heidelberg, 2010.

11 Marco Panunzio and Tullio Vardanega. A component model for on-board software applications. In *36th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2010*, pages 57–64, 2010.

12 José Pulido, Juan A. de la Puente, Matteo Bordin, Tullio Vardanega, and Jérôme Hugues. Ada 2005 code patterns for metamodel-based code generation. *Ada Letters*, XXVII(2):53–58, August 2007. Proceedings of the 13th International Ada Real-Time Workshop (IRTAW13).

13 Rapita Systems Ltd. *RVS Reference Guide*, 2011. Version 3.0.

14 José F. Ruiz. GNAT Pro for on-board mission-critical space applications. In Tullio Vardanega and Andy Wellings, editors, *Reliable Software Technologies — Ada-Europe 2005*, volume 3555 of *LNCS*. Springer-Verlag, 2005.

15 SPARC International, Upper Saddle River, NJ, USA. *The SPARC architecture manual: Version 8*, 1992.

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

A Formal Framework for Precise Parametric WCET Formulas*

Benedikt Huber, Daniel Prokesch, and Peter Puschner

Institute of Computer Engineering
Vienna University of Technology, Austria
`{benedikt,daniel,peter}@vmars.tuwien.ac.at`

Abstract

Parametric worst-case execution time (WCET) formulas are a valuable tool to estimate the impact of input data properties on the WCET at design time, or to guide scheduling decisions at runtime. Previous approaches to parametric WCET analysis either provide only informal ad-hoc solutions or tend to be rather pessimistic, as they do not take flow constraints other than simple loop bounds into account. We develop a formal framework around path- and frequency expressions, which allow us to reason about execution frequencies of program parts. Starting from a reducible control flow graph and a set of (parametric) constraints, we show how to obtain frequency expressions and refine them by means of sound approximations, which account for more sophisticated flow constraints. Finally, we obtain closed-form parametric WCET formulas by means of partial evaluation. We developed a prototype, implementing our solution to parametric WCET analysis, and compared existing approaches within our setting. As our framework supports fine-grained transformations to improve the precision of parametric formulas, it allows to focus on important flow relations in order to avoid intractably large formulas.

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

Keywords and phrases Worst-case execution time analysis, parametric WCET analysis, path expressions, frequency expressions, algebraic framework

Digital Object Identifier 10.4230/OASIcs.WCET.2012.91

1 Introduction

In hard real-time systems, it is crucial to guarantee that timing constraints are met. Consequently, determining the maximum time it might take to execute a task, its so called Worst-Case Execution Time (WCET), is both a necessary task in certification, and an important metric in the design of hard real-time systems. Since the early days of WCET analysis, there is a vital interest in parametric WCET analysis, which attempts to calculate a closed-form formula for the WCET, parametrized over an abstraction of the input space.

Formulas describing the WCET are particularly interesting during development. For example, formulas are well-suited to specify the timing behavior of components, or to classify the impact of input on the timing behavior [9]. Furthermore, WCET formulas can be used to determine the WCET depending on the actual input at runtime [18, 7, 15], which can guide dynamic scheduling, or facilitate the early detection of timing problems. We are particularly interested in the application of these techniques to optimization and the classification of the impact of architectural parameters.

* This work was partially funded under the European Union's 7th Framework Programme under grant agreement no. 288008: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST).

In recent years, there has been active development on techniques for determining symbolic flow facts, which characterize possible execution paths depending on the input data and reduce the search space for the worst-case execution path. Interesting examples include the automated computation of symbolic loop bounds [19] and solving of recurrence relations derived from abstract programs [1]. However, as already noted in the article of Chapman [6] in 1994, parametric WCET formulas are often not precise enough to allow for the derivation of tight bounds, if they fail to take additional flow facts (besides loop bounds) into account.

Flow facts of particular interest are bounds of inner loops in nested loops, where the bound of an inner loop varies with the iterations of the outer loop. Infeasible pairs model that two statements are mutually exclusive with respect to an execution context, and are common in embedded system code, especially in auto-generated code from synchronous languages or MATLAB Simulink [5].

The Implicit Path Enumeration Technique (IPET, [13, 16]) currently is the most widely-used technique for calculating the WCET. It generates an integer linear program (ILP), for which sophisticated solvers are available. In ILP, it is relatively easy to model flow facts besides loop bounds, improving the precision of the WCET bound. There is a parametric variant of the ILP problem called Parametric Integer Programming [14]; experiments with solvers for this powerful problem description language have been disappointing though [4, 3]. A function generating ILPs for each concrete input data configuration can also be viewed as a form of parametric ILP. While this approach has some interesting applications [5], its applicability is limited as the number of concrete ILP problems to solve increases exponentially in the number of input bits.

1.1 Motivating Example

The motivating example in Figure 1 illustrates the problem of taking additional flow facts into account, in this case a so-called triangle loop.

```
/* Nested loops */
void f(int N) {
  /* Loop Bound: N */
  for (i = 0; i < N; i++)
    if rand()
      B4();
    else
      B5();
      /* Loop Bound: i (N-1) */
      for (j = 0; j < i; j++)
        /* Global Bound: N (N-1) / 2 */
        B7();
      end
    end
  end
}

/* Symbolic Frequency Bounds depending on N */
```

$$f_{B4} + f_{B5} \leq N \, f_{B1} \quad (1)$$
$$f_{B7} \leq (N-1) \, f_{B5} \quad (2)$$
$$f_{B7} \leq ½ \, N \, (N-1) \, f_{B1} \quad (3)$$

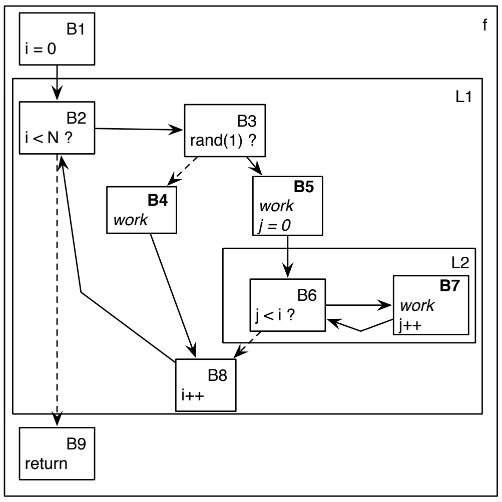


Figure 1 Example: Triangle Loop

In order to simplify the presentation, we will assume that the cost of all nodes but B_4, B_5 and B_7 is zero. Considering each iteration of the outer loop in turn, it is not difficult to see that a precise bound for the WCET of f is

$$\mathrm{WCET}(f, N) = \sum_{i=0}^{N} \max(B_4, B_5 + iB_7)\,. \tag{1}$$

While automatically calculating a closed formula for the exact parametric WCET of f is hard, it is interesting to investigate different approximations.[1] If no flow facts but symbolic loop bounds are taken into account [6, 2], the formula closely corresponds to a regular expression describing the set of paths, replacing concatenation by addition, and set union by the **max** operator. For loops, the loop body is multiplied by the symbolic loop bound, in this case N for the outer loop, and $N - 1$ for the inner loop. After simplification, we thus obtain

$$\mathrm{WCET}(f, N) = \max(NB_4, NB_5 + N(N-1)B_7)\,. \tag{2}$$

For larger values of N or B_7, the quadratic term dominates, leading to a WCET overestimation of up to 100%. Another recent approach is that of [4], which propagates constraints for each node, and then computes the sum of all nodes, multiplied by their symbolic execution bound. In this case, this would lead to the approximation

$$\mathrm{WCET}(f, N) = NB_4 + NB_5 + \frac{N(N-1)}{2}B_7\,. \tag{3}$$

This approach works well for single-path programs, but is rather pessimistic otherwise, as choice is mapped to addition instead of the maximum. Both of the previous approaches are easy to model in our framework. The techniques presented in this article allow to derive the following approximation, and prove it correct:

$$\mathrm{WCET}(f, N) = \max(NB_4, NB_5) + \frac{N(N-1)}{2}B_7\,. \tag{4}$$

This leads to a slight over-approximation if B_4 is expensive, but is a better approximation than the first one in the common case. In order to further improve the preciseness of the formula, one may introduce a case distinction, and either use Equation 2, if B_4 is executed at least $\frac{N}{2}$ times, or Equation 4 if it is executed less often. This approach increases the size of the formula, however, and thus in general may lead to intractably large formulas.

We believe that there is no single optimal strategy for constructing parametric formulas, and thus lobby for a formal framework, presented next, which allows to selectively refine formulas. In contrast to the related framework of Colin and Bernat [8], we chose an algebraic approach which is decoupled from the semantics of the analyzed program, and permits relatively simple correctness proofs. In Section 4, we describe the construction of formulas, the refinement of formulas using given, symbolic constraints, as well as the simplification and evaluation of formulas. Our experiments are described in Section 5, followed by our conclusion in Section 6.

[1] For this particular problem, the exact WCET can be computed manually, by determining the smallest integer k, such that if $i = k$, executing the nested loop is more expensive than executing B_4. With $k = \mathbf{min}(N, \mathbf{max}(0, \lceil\frac{B_4 - B_5}{B_7}\rceil))$, the WCET is then given by $\mathrm{WCET}(f, N) = kB_4 + (N-k)B_5 + \frac{N(N-1)-k(k-1)}{2}B_7\,.$

2 Background

In this work, we are mainly concerned with the construction of parametric WCET formulas from a given program representation, and the refinement of formulas using given program flow constraints. Therefore, we briefly review control flow representation and flow constraints, but do not address other relevant issues here, such as program-flow or processor-behavior analysis.

2.1 Control Flow Representation

The representation of programs in our analysis closely follows the machine code representation in the LLVM compiler framework[2], which is a central component of our evaluation framework.

A *control flow graph* (CFG) $\mathcal{G} = \langle V, E \rangle$ models the possible execution sequences of a function. Each node $v \in V$ corresponds to the execution of a sequence of instructions, and an edge (v, v') from $E \subseteq V \times V$ to a possible change of control from the last instruction of v to the first instruction of v'. The successors of a node v are given by $\text{succ}(v) = \{v' \mid (v, v') \in E\}$, the predecessors by $\text{pred}(v) = \{v' \mid (v', v) \in E\}$. A node v_d dominates a node v (denoted as v_d dom v) if every path from the start node to v must go through v_d. v_d strictly dominates v if v_d dom v and $v_d \neq v$.

We require several properties of the CFG representation, which are established in a preprocessing step. Each CFG $\mathcal{G}$ has a unique entry node $s_{\mathcal{G}}$, the only node where the set of predecessors is empty, and a unique exit node $t_{\mathcal{G}}$, with $\text{succ}(t_{\mathcal{G}}) = \emptyset$.

We only consider reducible CFGs [11] and thus irreducible loops need to be eliminated before WCET analysis. In a reducible CFG, loops $L \subseteq V$ are identified by a unique *header node* h_L, which dominates all loop members $v \in V_L$. A *back edge* is an edge from a member of a loop L to its loop header h_L. The acyclic *forward CFG* is obtained from a CFG by removing all back edges. Loops may be nested: The loop L_2 is an *inner loop* of L_1, if the header h_{L_2} is a member of loop L_1.

2.2 Flow constraints

In order to be WCET-analyzable, the maximum number of iterations of any loop in a program must be statically determinable. Applying to the CFG representation of a program, we define the *loop bound* of a loop L to be the maximum number of times the loop header node h_L of L is entered via any of its backedges in every execution.

Considering only the control-flow structure of the program together with simple numeric loop bounds will in most cases lead to an imprecise WCET bound. A reason for this are *infeasible paths*, i.e., structurally possible program paths that are not taken in any execution due to functional dependencies of program variables. Most prominent examples are mutually exclusive statements, and *triangle loops*, i.e., nested loops in which the loop bound of the inner loop depends on the iteration counter of the outer loop.

Linear flow constraints are the basis for IPET-based WCET calculation methods. In the corresponding ILP problem, the control-flow structure is expressed by means of linear relations between execution frequencies of CFG edges. Additional restrictions, for example to express relations between edge execution frequencies in different loop scopes, can easily be added into the system of linear constraints. Linear flow constraints have the form $\sum_i a_i \cdot f_{e_i} \leq C$, where f_{e_i} is the execution frequency of edge e_i, and $a_i, C \in \mathbb{Z}$ are constants.

[2] `http://www.llvm.org`

In case of symbolic a_i or C, the ILP problem is parametric and cannot be solved by standard IPET methods. Flow facts are either provided through manual annotations by the programmer or automatically derived by static program analysis.

3 Formal Framework

3.1 Path Expressions

Based on the work of [17], we can regard any path π in a directed graph $\mathcal{G} = \langle V, E \rangle$ as a string over E. A *path expression* P of *type* (v, w) with $v, w \in V$ (denoted as $P(v, w)$) is a regular expression over E such that every string π in the language $\sigma(P)$ is a path from v to w. Let $P(v, w)$ be a path expression of type (v, w). Then, all subexpressions P_1 and P_2 of path expression P are also path expressions, whose type can be defined recursively as follows [17]:

1. If $P = P_1 \cup P_2$, then P_1 and P_2 are of type (v, w) (alternative paths).
2. If $P = P_1 \cdot P_2$, there must be a unique vertex u such that P_1 is of type (v, u) and P_2 is of type (u, w) (path concatenation).
3. If $P = P_1^*$, then $v = w$ and P_1 is of type (v, v) (loop).

We call a path expression *complete* iff $\sigma(P(v, w))$ is the set of *all* paths from v to w in $\mathcal{G}$. For any given CFG $\mathcal{G}$ with entry node s and exit node t, the set of all structurally feasible (possibly infinite) paths through the CFG is defined by a complete path expression $P(s, t)$.

The underlying algebraic structure of path expressions is a Kleene algebra, i.e., an idempotent semiring (dioid) over E with the two binary operations of alternative paths $\cup$ as addition with neutral element $\emptyset$ and path concatenation $\cdot$ as multiplication with neutral element ε (empty string), and the additional unary operation of repetition * (cf. Table 1).

a^* is equivalent to the infinite expansion to $(\varepsilon \cup a \cup aa \cup aaa \cup \ldots)$. We exploit the algebraic structure to keep the a path expression P compact, as equivalence transformations on P do not change the language $\sigma(P)$.

Due to the associativity of both (binary) operations $\cup$ and $\cdot$ and by assigning operator precedences in the order *, $\cdot$, $\cup$ (starting with the highest), we can omit most brackets in path expressions. We also omit $\cdot$ in the notation, as usual for multiplication.

3.2 Frequency Expressions

Instead of calculating a cost formula directly from path expressions, we first introduce an abstraction from the set of paths to node frequencies. This abstraction lies at the heart of the successful IPET analysis, and is the basis for the concept of linear flow constraints. In our context, it permits us take flow constraints into account, to reason about node frequencies and prove the correctness of formula transformations.

The frequency $f_\pi(e)$ of an edge e on a path π is defined as the number of occurrences of e in π; that is, f_π is a function mapping edges to their occurrence count in π. A frequency constraint is a predicate on the frequencies of edges on a path, and acts a filter selecting valid paths from the set of all structurally possible ones. Formally, given a path expression $P(v, w)$, and a set of frequency constraints $\mathcal{C}$, the set of *valid paths* for $P(v, w, \mathcal{C})$ is given by $\sigma(P, \mathcal{C}) = \{\pi \in \sigma(P) \mid \forall C \in \mathcal{C} : C(f_\pi)\}$.

An expression P' is a *sound approximation* of $P(v, w, \mathcal{C})$, if $\sigma(P') \supseteq \sigma(P, \mathcal{C})$, that is, each valid path is included in the language described by P'. An approximation of an expression P' is called *exact* with respect to C, if $\sigma(P') = \sigma(P, \mathcal{C})$.

Frequency expressions are syntactically similar to path expressions; we just introduce a bounds notation $P^{[L,U]}$ which is equivalent to the expansion $\bigcup_{L \leq i \leq U} P^i$, with $P^i = P^{[i,i]} = PP \cdots P$ (i times) and $P^0 = \varepsilon$. Consequently we replace a path expression P^* by $P^{[0,\infty]}$ to simplify notation during constraint refinement (see Section 4.2). However, concatenation is commutative for frequency expressions. For example, while $P_1 = e_1 e_2$ and $P_2 = e_2 e_1$ are different path expressions, they are equivalent if interpreted as frequency expressions.

Frequency expressions are useful for two reasons. First, they allow us to take non-local constraints into account, as discussed in Section 4.2. Second, frequency expressions allow us to limit the growth of a formula when splitting subexpressions. For example, suppose that e_i and e_j are mutually exclusive, that is $\{\neg(f_{e_i} > 0 \wedge f_{e_j} > 0)\} \in \mathcal{C}$. Then $P = e_i \cdot Q \cdot e_j$ can be refined to $(e_i \cup e_j) \cdot Q$, while for path expressions we would be stuck with $(e_i \cdot Q) \cup (Q \cdot e_j)$.

3.3 Cost Expressions

Let $c : E \to \mathbb{N} \cup \{-\infty\}$ be a cost function, which assigns to each edge $e \in E$ of a CFG $\mathcal{G} = \langle V, E \rangle$ its maximum execution cost c_e. [3] Then we can derive a symbolic expression for the (possibly approximated) maximum cost $c(P(s,t))$ over all paths from the entry node s to the exit node t from the frequency expression $P(s,t)$ (or a sound approximation thereof), by replacing the underlying algebraic structure with $\langle \mathbb{N} \cup \{-\infty\}, \mathbf{max}, +, -\infty, 0 \rangle$, which we denote as $\mathbb{N}_{\max}$ in the following. $\mathbb{N}_{\max}$ is also a commutative dioid like the algebra of frequency expressions, with a total order defined on its elements by the order of the natural numbers, or equivalently, using the **max** operation: $a \leq b$ if $\mathbf{max}(a,b) = b$.

While in general for the frequency expression $(a \cup b)^N = \bigcup_{k=0}^{N} a^k\, b^{N-k}$, the total order of the elements in $\mathbb{N}_{\max}$ allows for following simplification due to monotonicity: $N * \mathbf{max}(a,b) = \mathbf{max}(N * a, N * b)$. Furthermore, as $\mathbf{max}(N * a, (N+1) * a) = (N+1) * a$, only the upper bound U in a frequency expression $P^{[L,U]}$ needs to be considered when calculating the maximum cost. As a consequence, in $\mathbb{N}_{\max}$ we cannot conveniently reason about edge (or node) frequencies, but only about (possibly approximated) path costs.

Table 1 provides a comparison of path-, frequency- and cost expressions.

4 Construction and Evaluation of WCET Formulas

In this section, we first give a concise description on how to construct path expressions in our framework, then present the refinement of frequency expressions to account for flow constraints, and finally describe the normalization of frequency expressions and our partial evaluation framework.

4.1 Building Path Expressions

We first consider an acyclic CFG $\mathcal{G} = \langle V, E \rangle$ with entry node s and exit node t. We want to obtain a complete path expression $P(s,t)$, approximating the set of valid paths from s to t. We recursively define $P(v,w)$ as

$$P(v,w) = \begin{cases} \varepsilon & \text{if } v = w \\ \bigcup_{(w',w) \in E} \; (P(v,w') \cdot (w',w)) & \text{if } v \neq w \end{cases}$$

[3] Given execution costs c_v of basic blocks $v \in V$ of the CFG, edge costs can be derived by attaching c_v either to all of its incoming edges or all of its outgoing edges.

Table 1 Comparison of path expressions, frequency expressions and cost expressions.

Path expressions	
Interpretation	Language of structurally possible program paths
Algebraic structure	Kleene algebra $\langle E, \cup, \cdot, ^*, \emptyset, \varepsilon\rangle$ (idempot. semiring with Kleene closure)
	$\cup$ is associative, commutative and idempotent; $\cdot$ is associative
	$\cdot$ is distributive w.r.t. $\cup$: $(a \cup b) \cdot c = (a \cdot c) \cup (b \cdot c)$, $a \cup (b \cdot c) = (a \cdot b) \cup (a \cdot c)$
	Zero element: $a \cup \emptyset = \emptyset \cup a = a$; Identity element: $a \cdot \varepsilon = \varepsilon \cdot a = a$
	$\emptyset$ annihilates E w.r.t. $\cdot$ $(a \cdot \emptyset = \emptyset \cdot a = \emptyset)$
	$\emptyset^* = \varepsilon^* = \varepsilon$
Example	$(e_{3,4}e_{4,8}) \cup (e_{3,5}e_{5,6}(e_{6,7}e_{7,6})^* e_{6,8})$
Frequency expressions	
Interpretation	Execution frequencies of edges in the CFG
Algebraic structure	Similar to path expr., but comm. dioid: $\cdot$ is commutative $(a \cdot b = b \cdot a)$
	Bounds notation for $P^{[L,U]}$ repetition bounds, $P^* \mapsto P^{[0,\infty]}$
	$\emptyset^{[0,U]} = \emptyset^{[0,0]} = \varepsilon$, while $\emptyset^{[1,U]} = \emptyset$
Example	$(e_{3,4}e_{4,8}) \cup (e_{3,5}e_{5,6}e_{6,8}(e_{6,7}e_{7,6})^{[0,N]})$
Cost expressions	
Interpretation	Formula for (maximum) execution cost
Algebraic structure	Commutative dioid $\langle \mathbb{N}_{\cup\{-\infty\}}, \mathbf{max}, +, -\infty, 0\rangle$
	Due to total order and monotonicity: $N * \mathbf{max}(a, b) = \mathbf{max}(N * a, N * b)$
Example	$\max(NB_4, NB_5) + \frac{N(N-1)}{2} B_7$

As we assumed the CFG to be acyclic, there is a partial order $\prec$ on the nodes of the graph with $w \in \text{pred}(v) \Rightarrow w \prec v$, for all $v, w \in V$. By determining this topological order, and representing each expression $P(s, v)$ only once in memory, we obtain a closed form for $P(s, t)$ in time and space linear in the size of the CFG.

Now consider a CFGs $\mathcal{G} = \langle V, E\rangle$ with reducible loops. We observe that the set of cycle-free paths (i.e., those which do not include back edges) is generated by calculating $P(s, t)$ for the forward CFG $\mathcal{G}^F = \langle V, E^F\rangle$. Furthermore, all cycles are composed of paths starting from a loop header and ending at the corresponding back edge. Therefore, in the general case we recursively define $P(v, w)$ as

$$P(v, v) = \begin{cases} \varepsilon & \text{if } v \text{ is not a loop header} \\ \bigcup_{(v',v) \in E \setminus E^F} P(v, v') \cdot (v', v) & \text{if } v \text{ is header of loop } L \end{cases}$$

$$P(v, w) = \bigcup_{(w',w) \in E^F} (P(v, w') \cdot (w', w)) \cdot P(w, w)^* \qquad \text{if } v \neq w$$

The construction of path expressions simply takes the union of all paths leading to predecessors. However, those paths will often have a common prefix, namely the path expression from the entry to the dominator of predecessors. Therefore, we factor out common prefixes during construction, using the equivalence $\bigcup_j (P(s, d) \cdot P(d, j)) = P(s, d) \cdot \left(\bigcup_j P(d, j)\right)$.

▶ **Example 1.** Consider the example presented in Section 1.1. Let $e_{i,j}$ denote the edge from B_i to B_j. Then applying the construction algorithm, and factoring out the common prefix

$P(B_2, B_3)$ leads to the following path expressions:

$$\begin{aligned}
P(B_1, B_9) &= e_{1,2}P(B_2, B_2)^* e_{2,9} \\
P(B_2, B_2) &= e_{2,3}P(B_3, B_8)e_{8,2} \\
P(B_3, B_8) &= (e_{3,4}e_{4,8}) \cup (e_{3,5}e_{5,6}P(B_6, B_6)^* e_{6,8}) \\
P(B_6, B_6) &= e_{6,7}e_{7,6}
\end{aligned}$$

4.2 Constraint Refinement of Frequency Expressions

The formulas derived so far express possible paths constrained only by the program structure as obtained from the CFG. In order to calculate a (precise) WCET bound, we need to take flow constraints into account. As argued before, frequency expressions are a well-suited formalism for this task. Therefore, we interpret the initial path expression as frequency expression before constraining the formula.

4.2.1 Constraint Refinement

The following observation illustrates how to take local frequency bounds into account.

▶ Observation 1 (Constraint Refinement). Given a a frequency expression $P(s, t) = R \cdot Q(u, v)^{[A,B]}$, and a frequency constraint $C = \sum_{(u',u) \in E} f_{(u',u)} \leq N$, $C \in \mathcal{C}$. Then the frequency expression $P'(s, t) = R \cdot Q(u, v)^{[A,N]}$ is a sound approximation of $P(s, t, \mathcal{C})$.

Due to the way frequency expressions are constructed, it is thus always possible to account for constraints which limit the frequency of a node relative to its immediate dominator. For non-empty frequency expressions $Q(v, v)$, corresponding to the body of a loop, the constraint only needs to refer to total frequency of the loop's back edges. Local constraints, in particular simple loop bounds, should always be applied to frequency expressions, in order to facilitate the calculation of a numeric WCET bound.

▶ **Example 2.** Consider the path expressions presented in Example 1, interpreted as frequency expressions. In the motivating example, we have $f(e_{8,2}) \leq N \in \mathcal{C}$ for all paths from B_1 to B_9. Applying Observation 1, we thus get

$$P'(B_1, B_9, \mathcal{C}) = e_{1,2}P(B_2, B_2)^{[0,N]}e_{2,9}$$

4.2.2 Global Bounds

Frequency expressions also allow us to take non-local frequency bounds into account. We start with the following two observations:

▶ Observation 2. Given a frequency expression $P(s, t) = (Q \cdot R)^{[A,B]}$, the expression $P'(s, t) = Q^{[A,B]} \cdot R^{[A,B]}$ is a sound approximation of $P(s, t)$.

▶ Observation 3. Given a frequency expression $P(s, t) = ((Q \cdot R) \cup S)^{[A,B]}$. Then $P'(s, t) = Q^{[0,B]} \cdot (R \cup S)^{[A,B]}$ is a sound approximation of $P(s, t)$.

These observations allow to move subexpressions to an outer scope, which in turn enables the inclusion of non-local loop bounds. The following example illustrates this technique.

▶ **Example 3.** Consider the frequency expression from Example 2. Applying both transformations presented above to lift $P(B_6, B_6)$ gives

$$\begin{aligned}
P''(B_1, B_9) &= e_{1,2}P''(B_2, B_2)^{[0,N]}P(B_6, B_6)^{[0,N(N-1)]}e_{2,9} \\
P''(B_2, B_2) &= e_{2,3}\left((e_{3,4}e_{4,8}) \cup (e_{3,5}e_{5,6}e_{6,8})\right)e_{8,2}
\end{aligned}$$

Then applying constraint refinement to $P(B_6, B_6)$ results in

$$P'''(B_1, B_9) = e_{1,2}P''(B_2, B_2)^{[0,N]}P(B_6, B_6)^{[0,\frac{1}{2}N(N-1)]}e_{2,9}$$

Note, that while using the first observation might improve the WCET bound, applying the second or third one may make it worse. Thus it is necessary to devise a heuristic which decides whether a non-local constraint should be applied. Another alternative to overcome possible degradation is to mimic the IPET approach and split the formula, distinguishing the case when the non-local constraint is useful, and the one when the local one is better.

4.2.3 Infeasible Pairs

An infeasible node $x \in V$ is a node that does not lie on any feasible path from s to t. Given a frequency expression $P(s, t)$ and an infeasible node x, the set of valid paths $\sigma(P, \mathcal{C})$ satisfies the constraint $C = \{f_e = 0 \mid e \in E_x\}$, $C \subseteq \mathcal{C}$, where E_x is the set of edges incident to x, i.e., the frequency of all incoming edges and all outgoing edges of x is zero.

The frequency expression $P(s, t)[e \mapsto \emptyset]$, $e \in E_x$ is a sound approximation for $P(s, t, \mathcal{C})$.[4] Recall that $\emptyset^{[0,U]} = \emptyset^{[0,0]} = \varepsilon$, while $\emptyset^{[1,U]} = \emptyset$. With this transformation, every subexpression P' of the form $P' = \prod_k P_k$ where some $P_k = \emptyset$ will thus be annihilated in P, i.e., any path that formerly contained x at least once, is pruned from the set $\sigma(P)$, while all other paths are preserved.

An infeasible pair (x, y) is a pair of nodes $x, y \in V$ such that any path from s to t that contains both x and y is infeasible. We can restrict the set of valid paths by taking the union of the path set in which x is infeasible and the path set in which y is infeasible. Formally, we obtain a sound approximation for $P(s, t, \mathcal{C})$ where $C = \{\neg\left(f_{e_x} > 0 \wedge f_{e_y} > 0\right), e_x \in E_x, e_y \in E_y\}$, $C \subseteq \mathcal{C}$ by the frequency expression $P(s, t)[e_x \mapsto \emptyset] \cup P(s, t)[e_y \mapsto \emptyset]$, $e_x \in E_x$, $e_y \in E_y$. The resulting expression describes the set of paths pruned only of paths that contain both nodes x and y, and hence is an exact approximation.

Obviously, the size of the formula increases when taking infeasible pairs into account. Indeed, as WCET calculation is NP-complete in the presence of infeasible pairs[5], it is not possible to obtain a compact formula in the general case. However, by exploiting commutativity in frequency expressions to reduce the size of the duplicated part of the formula, and by taking only those infeasible pairs into account which improve the WCET significantly, we hope keep the size of the formula in reasonable limits.

4.3 Simplification and Partial Evaluation

We use the properties of the frequency expression algebra to normalize frequency and cost expressions, which provides the basis for the partial evaluation described at the end of this section.

4.3.1 Normalized Frequency Expressions

In the normalized form, every frequency expression is either a single node, a union $\bigcup_i P_i$ of normalized frequency expressions, or a product $\prod_i P_i^{[L_i,U_i]}$. The normalized form of frequency expressions has the following properties:

[4] $P[e \mapsto e']$ denotes the frequency expression P, with all occurrences of the subexpression e replaced by e'.

[5] As can be shown by a polynomial-time reduction of W2SAT.

- *Subsumption (Unions)*: Given two products $P = \prod_i P_i^{[L_i,U_i]}$ and $Q = \prod_j Q_j^{[L_j,U_j]}$, if for every $Q_j^{[L_j,U_j]}$ there is a $P_i^{[L_i,U_i]}$ with $P_i = Q_j$, $L_i \leq L_j$ and $U_j \leq U_i$, then P subsumes Q ($\sigma(Q) \subseteq \sigma(P)$) and thus $P \cup Q = P$. For every normalized frequency expression $\bigcup_i P_i$, each P_i is a product, and if P_i subsumes P_j, P_j is not present in the union.
- *Frequency Product (Products)*: The product $P^{[L_1,U_1]} \cdot P^{[L_2,U_2]}$ simplifies to $P^{[L_1+L_2,U_1+U_2]}$, and $(\prod_i P_i^{[L_1,U_1]})^{[L_2,U_2]}$ simplifies to $\prod_i P_i^{[L_1 L_2,U_1 U_2]}$. Furthermore, we apply $P \cdot \bigcup_\emptyset = \bigcup_\emptyset$ whenever possible. Therefore, in every frequency expression $\prod_i P_i^{[L_i,U_i]}$, all P_i are distinct, and each P_i is either a non-empty union of frequency expressions or a single node.

4.3.2 Partial Evaluation

Given a frequency expression $P(s,t)$ for a control flow graph, we perform partial evaluation to obtain a numeric or parametric formula for the WCET. We call this step *partial* evaluation, as the evaluation functions c and f are allowed to be partial.

For evaluation purposes, we introduce a special node $\underline{c}$, which represents one unit of cost. The evaluation algorithm takes a frequency expression $P(s,t)$, a partial cost function c (assigning costs to edges), and a frequency evaluation function f, transforming symbolic frequencies (either into numeric ones or different symbolic frequencies, e.g. to reflect some parameter of interest, like size of an input array).

First, all edges e where $c(e)$ is defined are replaced by $\underline{c}^{c(e)}$, and all expressions $P^{[L,U]}$ are simplified to $P^{[f(L),f(U)]}$. The actual evaluation then corresponds to the normalization of the frequency expression, as described before. This is sufficient for full evaluation, while for partial evaluation the size of the formula can be further reduced by additional simplifications, which exploit the total order of numeric cost values.

Given the result of the partial evaluation, we would like to obtain information on node frequencies in the evaluated formula. In principle, this can be achieved by keeping references to original expressions during simplifications, and keeping track of selected branches in unions, though this has not been elaborated yet.

5 Experiments

In order to validate the applicability of our parametric execution time formula framework, we implemented a prototype on top of the Open Timing Analysis Platform [12]. This evaluation framework is based on the LLVM compiler framework, and extracts description of machine code CFGs from the internal compiler representation. Flow facts are provided by the SWEET [10] analysis tool, developed at the Mälardalen Real-Time Research Center (MRTC) which is integrated in our evaluation framework.

For our preliminary experiments, we generate machine code for the ARM instruction set and use a simple cost model (one cycle per instruction). In Table 2, we present results for three benchmarks which feature triangle loops. One is adapted from the motivating example in Section 1.1, and two are taken from the MRTC benchmark set.

In these experiments, formulas are parametric with respect to loop bounds (first column). Local bounds L_i constrain the loop iteration count relative to the loop entry frequency, global bounds G_i are relative to the function entry. The fourth column displays the result of evaluating the formulas with the specified numeric loop bounds.

We compare the standard approach which only uses simple loop bounds (*local bounds only*), the result which only uses bounds relative to the function entry (*global bounds only*),

Table 2 Results of the comparison of different approaches to parametric analysis.

	approach	formula	cycles
`intro_example`	local bounds only	$6 + 12L_1 + L_1 * \max(28, 19L_6)$	1836
$L_1 = 10, L_6 = 9,$	global bounds only	$6 + 45L_1 + 19G_6$	1311
$G_6 = 45$	lifting inner loops	$6 + 40L_1 + 19G_6$	1261
	IPET	—	1121
`insert_sort`	local bounds only	$29 + 14L_1 + 11(L_1\,L_3)$	1046
$L_1 = 9, L_3 = 9,$	lifting, global	$29 + 14L_1 + 11G_3$	650
$G_3 = 45$	IPET	—	650
`janne_complex`	local bounds only	$32 + 14L_4 + 22L_2 + 14(L_2\,L_4)$	1216
$L_2 = 8, L_4 = 8,$	global bounds only	$18 + 8L_2 + 15G_4$	262
$G_4 = 12$	lifting inner loops	$18 + 8L_2 + 14G_4$	250
	IPET	—	250

and the transformation moving nested loops to the outer scopes (*lifting inner loops*), described in Section 4.2.2.

6 Conclusion

We presented a framework for the calculation of symbolic execution time formulas, which provides a solid foundation for parametric WCET analysis. We construct path expressions from control flow graphs, and then add commutativity to obtain frequency expressions. The key idea is that while frequency expressions are easier to work with than path expressions, they still represent a sound approximation to the set of possible execution paths. By the virtue of this property, it is possible to selectively include additional flow constraints, for example global loop bounds or infeasible pairs, and prove this refinements correct.

The preliminary experiments have been encouraging, and the resulting formulas indeed reflect our intuition, and suggest that this approach is not only theoretically sound, but also works well in practice. We are eager to extend it to a fully-featured implementation, supporting context-sensitive supergraphs, feedback on worst-case frequencies, and additional formula refinements. There are many interesting applications to parametric WCET analysis, especially at design time, and we believe that this framework provides a good basis for further improving the state-of-the art in this area.

References

1 Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Closed-form upper bounds in static cost analysis. *J. Autom. Reason.*, 46(2):161–203, February 2011.

2 Ernst Althaus, Sebastian Altmeyer, and Rouven Naujoks. Precise and efficient parametric path analysis. In *LCTES '11: Proceedings of the ACM SIGPLAN/SIGBED 2011 conference on Languages, compilers, and tools for embedded systems*, pages 141–150, New York, NY, USA, April 2011. ACM.

3 Sebastian Altmeyer. Parametric WCET analysis, parameter framework and parametric path analysis. Master's thesis, Universität des Saarlandes, 2006.

4 Stefan Bygde, Andreas Ermedahl, and Björn Lisper. An efficient algorithm for parametric wcet calculation. *Journal of Systems Architecture - Embedded Systems Design*, 57(6):614–624, 2011.

5 Susanna Byhlin, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Applying static WCET analysis to automotive communication software. In *ECRTS*, pages 249–258. IEEE Computer Society, 2005.

6 Roderick Chapman. Worst-case timing analysis via finding longest paths in spark ada basic-path graphs. Technical Report Technical Report YCS-94-246, Department of Computer Science, University of York, October 1994.

7 Joel Coffman, Christopher Healy, Frank Mueller, and David Whalley. Generalizing parametric timing analysis. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '07, pages 152–154, New York, NY, USA, 2007. ACM.

8 Antoine Colin and Guillem Bernat. Scope-tree: A program representation for symbolic worst-case execution time analysis. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, ECRTS '02, pages 50–, Washington, DC, USA, 2002. IEEE Computer Society.

9 S.V. Gheorghita, S. Stuijk, T. Basten, and H. Corporaal. Automatic scenario detection for improved WCET estimation. In *Design Automation Conference, 2005. Proceedings. 42nd*, pages 101 – 104, june 2005.

10 Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Bjorn Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, RTSS '06, pages 57–66, Washington, DC, USA, 2006. IEEE Computer Society.

11 Paul Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, 19(4):557–567, July 1997.

12 Benedikt Huber, Wolfgang Puffitsch, and Peter Puschner. Towards an open timing analysis platform. In *11th International Workshop on Worst-Case Execution Time Analysis*, July 2011.

13 Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, DAC '95, pages 456–461, New York, NY, USA, 1995. ACM.

14 Björn Lisper. Fully automatic, parametric worst-case execution time analysis. In Jan Gustafsson, editor, *WCET*, volume MDH-MRTC-116/2003-1-SE, pages 99–102. Department of Computer Science and Engineering, Mälardalen University, Box 883, 721 23 Västerås, Sweden, 2003.

15 Sibin Mohan, Frank Mueller, Michael Root, William Hawkins, Christopher Healy, David Whalley, and Emilio Vivancos. Parametric timing analysis and its application to dynamic voltage scaling. *ACM Trans. Embed. Comput. Syst.*, 10(2):25:1–25:34, January 2011.

16 Peter P. Puschner and Anton V. Schedl. Computing maximum task execution times - a graph-based approach. *Real-Time Systems*, 13(1):67–91, 1997.

17 Robert Endre Tarjan. A unified approach to path problems. *J. ACM*, 28(3):577–593, July 1981.

18 Emilio Vivancos, Christopher A. Healy, Frank Mueller, and David B. Whalley. Parametric timing analysis. In *LCTES/OM*, pages 88–93. ACM, 2001.

19 Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction. In *Proceedings of the 18th international conference on Static analysis*, SAS'11, pages 280–297, Berlin, Heidelberg, 2011. Springer-Verlag.

Evolutionary Techniques for Parametric WCET Analysis*

Amine Marref

Department of Computer Science
Umm Al-Qura University
Makkah, Saudi Arabia
ajmarref@uqu.edu.sa

Abstract

Estimating the worst-case execution time (WCET) of real-time programs is pivotal in their verification. WCET estimation either yields a numeric value that represents the maximum execution time of the program when executed on a specific hardware platform; or yields a parametric expression in the form of some function of the input which when instantiated with a particular input value, gives a WCET estimation of the program when triggered by this input specifically (on a specific hardware platform). Parametric WCET analysis provides extra accuracy as the WCET estimation can be tuned to particular input values at runtime; and is usually of interest to dynamic-scheduling schemes.

In this paper we use genetic programming as an alternative method to approach the problem of parametric WCET analysis. Parametric expressions are captured automatically by the genetic program based on end-to-end executions of the program under analysis. The technique is complementary to static parametric WCET analysis and more amenable to industrial practice. Experimental evaluation shows that the presented technique computes accurate parametric expression in an almost negligible time.

1998 ACM Subject Classification D.2.5 Testing and Debugging

Keywords and phrases real-time systems, parametric worst-case execution-time analysis, end-to-end testing, genetic programming

Digital Object Identifier 10.4230/OASIcs.WCET.2012.103

1 Introduction

WCET analysis can result in two types of WCET estimations: numeric values or parametric expressions. WCET estimations in the form of numeric values are the predominant in the literature; in this case the WCET analysis returns a single constant value that indicates the maximum number of clock cycles (or other time-measurement units) that the program can potentially spend during its lifetime execution on a specific hardware platform. On the other hand, parametric WCET estimations come in the form of mathematical functions of the inputs of the program.

Parametric WCET estimations are more useful than numeric-form estimations when the scheduling algorithm — that uses the results of the WCET analysis — can utilize the extra context-sensitivity provided by the parametric expression. In this case, the scheduling algorithm can compute a constant WCET value $w_p(v_i)$ for some program p with parametric

* This research is funded by the program of strategic technologies of the national plan for science, technology, and innovation in Saudi Arabia No. 11–INF1705–10.

expression $w_p(v)$ each time p consumes a new input vector $v = v_i$, by instantiating the input v_i inside the parametric expression $w_p(v)$.

Parametric WCET analysis has been the subject of several research works whose most common factor with respect to our work is that they are based on static analysis of program code to determine the parametric expressions. In this paper, we present a novel approach for the problem of parametric WCET estimation based on genetic programming, and is more suitable for end-to-end WCET analysis used in the industry.

This paper is structured as follows. Section 2 introduces genetic programming. Section 3 explains the problem of parametric WCET analysis from the paper's point of view. Section 4 explains the use of genetic programming to approach the problem of parametric WCET analysis. Section 5 describes the experimental evaluation of our approach. Section 6 describes related work in parametric WCET analysis. Section 7 contains concluding remarks and directions for future work.

2 Genetic Programming

Genetic programming (GP) [8] is a bio-inspired computer algorithm that mimics natural evolution of living organisms. It is similar to genetic algorithms with the exception that individuals are computer programs as opposed to vectors of values.

The objective of GP is to evolve a computer program that solves a given problem. In order to do so, a population of computer programs called individuals — that are randomly generated initially — is evolved across a number of generations. The evolution of the population involves the exchange of genetic material between the individuals through cross-over operations, and the alteration of the genetic material of single individuals through mutation operations. A selection strategy is applied to the individuals of a population in a given generation to decide which ones are allowed to proceed to the next generation. Such selection is based on the fitness of the individuals which is a problem-dependent value that specifies the goodness of an individual in solving the problem at hand. The evolution continues until a good-enough individual that solves the problem adequately is found, or until a maximum number of generations is reached.

Each individual in the population is a program represented by its abstract-syntax tree (AST). All non-leaf nodes of the AST represent operators, and leaf nodes represent problem variables or constant values. Crossing-over two programs means taking one or more subtrees from the first program and inserting them into the second program, and taking one or more subtrees from the second program and inserting them into the first program (cross-overs can be single-point or multiple-point). Mutating means changing the content of one or more nodes in the AST.

GP can be used to solve a variety of optimization problems amongst which is symbolic regression that we shall describe here because it is the essence of our approach. To solve a symbolic-regression problem (also known as function-discovery problem), the genetic program (GP[1]) takes as input a set of m observations of values of some variable x, a set of observations of values of some variable y, and tries to identify the function f_0 such that $y = f_0(x)$ is true for all pairs (x, y) in the m observations — and also true outside the m observations. The function f_0 to be determined is a computer program that will be evolved by the GP. The initial population of the GP contains a number of n randomly-generated functions $f_1, f_2, \cdots, f_n$ — each represented as an AST e.g. Figure 1 shows the AST representation of

[1] We use GP to refer to both "genetic programming" and "genetic program".

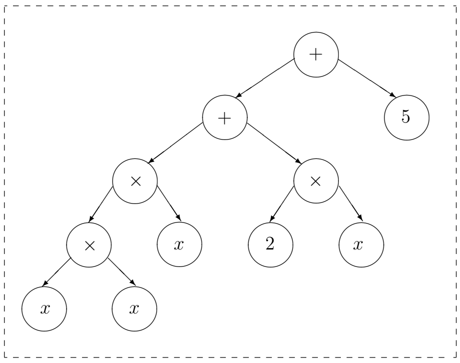


Figure 1 An example abstract-syntax tree corresponding to some function $f_i(x) = x^3 + 2x + 5$.

some function $f_i(x) = x^3 + 2x + 5$ in the GP. The functions will be crossed over and mutated over generations to produce new fitter functions. The fitness of a function f_i is calculated as the sum of differences ($|x - y|$) for all pairs (x, y) in the m observations. At the end, the GP either discovers f_0 during evolution or another function of equal or inferior fitness. Notice that the GP can derive some function f'_0 that satisfies $y = f'_0(x)$ for all observed data pairs (x, y) but does not satisfy $y = f'_0(x)$ in the general case i.e. for some unobserved pairs (x, y) the relation $(y = f_0(x) \wedge y \neq f'_0(x))$ holds. This is a classic case of over-fitting and is usually (partially) tackled by dividing the m observations into a training part used during evolution, and a testing part used after evolution to give an indication of how well the derived function f'_0 generalizes to new unseen data. Should the derived function not generalise well enough, evolution is restarted with the derived function f'_0 injected in the initial population of the new GP run.

3 Parametric WCET Analysis

The objective of parametric WCET analysis is to derive a function that expresses the WCET of some program in terms of its input. These parametric expressions contain constant factors and terms, and variable factors and terms. The constant factors and/or terms in the parametric WCET expression refer to known entities in the analysis such as program-segment execution times and program-segment execution counts. These are derived by (i) the WCET analysis through its flow analysis (execution counts) or processor-behaviour analysis/runtime measurements (execution times), (ii) input by the user (usually as execution counts in the form of loop and recursion bounds, and sometimes as execution times of black-box components or libraries not available for the analysis), or (iii) a mixture of analysis-derived and user-input execution times and execution counts. We use the term program segment to refer to program entities of interest to parametric WCET analysis e.g. a loop body whose number of iterations is controlled by the input and which appears as a term $a_j x_j$ in the parametric expression — where a_j is the execution time of the loop body and x_j is its execution count.

The variable factors and terms in the parametric expression can refer to the whole input, parts of the input, and/or some of the input's properties of interest. For example, in a program that returns the factorial of a nonnegative integer, the whole input i.e. the integer whose factorial is of interest becomes a variable in the parametric expression; and in a program that sorts integers, the size of the input (which is a property of the input) becomes a variable in the parametric expression. Here we use the word input to mean the type of the input as opposed to a single instantiation of it e.g. an input is an array of n integers, is a real number, etc.

Input can influence the execution time of the program in two different ways: by altering the execution times of the program's segments, and/or altering their execution counts. For example, if part of the input is passed as one of the operands of some variable-latency instruction such as multiplication, the WCET estimation will change according to the value passed to the variable-latency instruction. In this case, input affects the execution time of the program segment that contains the variable-latency instruction, and consequently affects the overall program's execution time. If part of the input contributes directly or indirectly to the value of some variable that controls the number of iterations of some loop, the execution time of the program will also change according to the value of this input. In this case, the input affects the number of times a set of program segments executes, and consequently affects the overall program's execution time. Figure 2 shows example scenarios of how input affects the execution time of the program. In line 32, input variable c affects the flow of the program and consequently affects the execution time. Including c in the parametric WCET expression of `foo()` will be in the form of a conditional statement. Formula (1) shows (a very simplified) parametric WCET expression of function `foo()` of Figure 2. The function $f_{mul}(a, b)$ gives the execution time of the multiplication operation in terms of its operands a and b. Formula (1) is the ultimate form of WCET parameterization as it accounts for the diverse ways by which the input affects the execution time.

$$w_{foo} = \begin{cases} n + m + f_{mul}(x, x) + 1 & \text{if } c = \mathit{TRUE} \\ n + f_{mul}(x, x) + 2 & \text{if } c = \mathit{FALSE} \end{cases} \tag{1}$$

In the parametric-WCET literature, the derived parametric expressions are in the form of polynomial functions where the variables refer to input parts or properties that influence loop iteration numbers. The derived parametric expressions are usually expressed conditionally over subdomains of the input space. In the literature, if the derived parametric WCET of some program p has the form $w_p = a_k x^k + a_{k-1} x^{k-1} + \cdots + a_0$, it means that the program have parts that have execution times $a_i, i \in [0..k]$ and are repeated $x^i, i \in [0..k]$ times where x is a part or a property of the input to program p. In these types of expressions, x is a variable that will control the iteration number of one or more loops in the program under analysis. In this paper, we will focus on deriving parametric WCET expressions where the final expression is polynomial. Deriving more complex parametric expressions that account for variable-latency instructions or those that are expressed conditionally over input space is left for future work.

4 Parametric WCET Analysis using Genetic Programming

Parametric WCET analysis requires knowledge about the input parts or properties that affect the execution time of the program — and which will appear in the final parametric expression. We assume in this paper that knowledge about the exact parts or properties of the input that influence the WCET is available to us through some third-party analysis or user-input; and we exclusively consider those that affect loop bounds and/or recursion depths as we have discussed. We will refer to the input parts and properties that appear in the final parametric expression as parameters.

The problem of deriving parametric WCET expressions reduces to a problem of symbolic regression. Let p be the program under analysis. Program p is executed for a number of m

```
// The input to this program is composed of the following fields:
//   1. two integer arrays,
//   2. an integer variable,
//   3. and a boolean variable.
private static int foo (Integer[] A, Integer[] B, int x, boolean c)
{
        // The size of the array (n/m) is a property of the input.
        // Assume that execution time of the following segment is 1.
        int n = A.length;
        int m = B.length;

        // This is a part of the program whose execution time depends on
        // some property of the input. This is a case where the input
        // affects execution counts.
        for(int i=0; i<n; i++)
        {
                // Do something.
                // Assume that execution time of this body is 1.
        }

        // This is a part of the program whose execution time depends on
        // some part of the input. This is a case where the input affects
        // execution times directly assuming multiplication is variable-
        // latency instruction in the target hardware.
        // Assume that execution time of multiplication of a and b is
        // f_{mul}(a,b).
        int y = x*x;

        // This is a part of the program whose execution time depends on
        // some part of the input. This is a case where the input affects
        // execution counts in a conditional manner.
        if(c)
        {
                for(int i=0; i<m; i++)
                {
                        // Do something.
                        // Assume that execution time of this body is 1.
                }
        }
        else
        {
                // Do something.
                // Assume that execution time of this part is 1.
        }
        return 0;
}
```

Figure 2 Some function `foo()` that illustrates different scenarios of how input influences the execution time of a program.

inputs: in each run $i \in [1..m]$ the values $x_{1,i}, x_{2,i}, \cdots, x_{n,i}$ of all n parameters $x_1, x_2, \cdots, x_n$ are recorded together with p's end-to-end execution time t_i. After all runs have completed, we obtain a m-by-$(n+1)$ matrix of real numbers. The objective then is to find the function w_p that satisfies $t_i = w_p(X_i), i \in [1..m]$ where X is a vector defined as $X =< x_1, x_2, \cdots, x_n >$ and X_i is the value of X in run i i.e. $X_i =< x_{1,i}, x_{2,i}, \cdots, x_{n,i} >$. Let X^k be the vector that contains the parameters $x_1, x_2, \cdots, x_n$ raised to the power k i.e. $X^k = x_1^k, x_2^k, \cdots, x_n^k$.

The parametric expression w_p is just a function that expresses the execution time of the program p in terms of its parameters. It becomes a parametric *WCET* expression once the m runs exercise different program segments in their worst-case execution scenarios. This means that if the expression w_p has the form $w_p = A_k X^k + A_{k-1} X^{k-1} + \cdots + A_0$ where $A_i, i \in [0..k]$ is a vector of constants, then w_p is WCET expression if the values in the vectors $A_k, A_{k-1}, \cdots, A_0$ — which correspond to program-segment execution times — are maximum. This depends on the quality of testing which is a common issue in all measurement-based and end-to-end WCET analyses. Here we will focus on using the GP's symbolic regression to learn a parametric expression as opposed to trying to prove that the factors $A_k, A_{k-1}, \cdots, A_0$ have their maximum-possible values — which is a separate problem to solve outside this paper.

In order to discover the parametric expression, the GP is informed with the types of operators and terminals (variables and constants) that can potentially appear in the derived parametric expression. As we have discussed before, the operators will appear in the non-leaf nodes and the terminals will appear in the leaf nodes of the AST representation of the parametric expression. In our case, we have limited our attention to polynomials; which by their mathematical definition allow addition and subtraction of terms, each term can be composed by multiplying variables by variables or variables by constants, raising a variable to the power of a constant, or dividing a variable by a constant. Therefore the set of operators to consider is {addition, subtraction, multiplication, division}.

In addition to this, depending on implementation, the GP can benefit from knowledge about the shape, depth, and size of the ASTs that represent the target parametric expressions. A binary AST is usually the choice for symbolic regression problems where the target expression is a polynomial. The depth of the binary AST will influence the order of the polynomial. For example, the parametric expression of Figure 1 requires an AST of depth 5 (assuming root depth is 0) for a polynomial of order 3.

Notice that two runs of the GP (based on the same data set) that are performed to derive the parametric expression w_p of some program p are not guaranteed to derive the same parametric expression i.e. they will potentially result in two expressions w_p and w'_p such that $w_p \neq w'_p$. However, if the two runs of the GP are given the same resources (time, processing power, memory, etc.) it is unlikely that the structure of the parametric expressions will be different; but their constant factors are likely to differ slightly. The reason for this is that GP is a search-based method that is based on some element of randomness in the genetic operations such as cross over. The GP converges towards a solution of some fitness — after a number of generations — which is often almost the same in different runs of the GP on the same problem instance — as long as the GP has access to the same resources in these different runs. Here, "almost the same" is seen in the form of parametric expressions that have the same AST but have slightly-different constant factors.

5 Evaluation

To evaluate our approach, we use the Mälardalen WCET benchmarks [10] which have historically been used to evaluate parametric WCET approaches in the literature. The benchmarks have been modified to make them amenable to end-to-end testing — basically by allowing the function `main()` to take input arguments, processing them using the function `atoi()`, and passing them to the function of interest (e.g. `factorial`, `insertion_sort`, etc.). The benchmarks we used are described in Table 1 and they include all benchmarks used in [3, 16, 9, 11, 15, 6, 2, 5, 1] except those not available in [10]. The benchmark `janne_complex` has been augmented with a new parameter c that substitutes the constant value 30 used in the outermost loop — to make it more interesting to analyse.

Each benchmark program in Table 1 is compiled for the ARM architecture using a *gcc* cross compiler and executed on Simplescalar [4] using the configuration shown in Figure 3. The reason behind using Simplescalar instead of actual hardware is that we don't have access to actual hardware and accompanying execution-time measurement equipment. The reason behind using the configuration of Figure 3 is to top the most complex hardware platform used in parametric WCET analysis by [2] where the authors use MPC565 for evaluating their work.

Table 1 The benchmark programs used in the evaluation.

Benchmark	**Description**	**Parameters and Ranges**
bsort100	Bubble Sort (Triangular Loop)	Size n of array to be sorted $n \in [1..100]$
cnt	Matrix Count	Size n of side of square matrix $n \in [1..30]$
crc	Cyclic Redundancy Check	Size n of input string $n \in [1..200]$
fac	Factorial	Integer n $n \in [1..100]$
fir	Finite Impulse Responder	Variables in_len, $coef\_len$, and $scale$ $in\_len, coef\_len, scale \in [1..700]$
insertsort	Insertion Sort	Size n of array to be sorted $n \in [1..100]$
janne_complex	Nested Loop Program	Variables a, b, and c $a, b, c \in [1..100]$
matmult	Matrix Multiplication	Size n of side of square matrix $n \in [1..20]$
sqrt	Square Root Computation by Taylor Series	Integer n $n \in [1..100]$
st	Statistics Program	Size n of arrays to be processed $n \in [1..300]$

```
# Pipeline                    -res:imult 1                    -cache:il2 none
-fetch:ifqsize 4              -res:memport 2                  -cache:dl2 none
-decode:width 1               -res:fpalu 1
-issue:width 1                -res:fpmult 1                   # Branch Predictor
-issue:inorder false                                          -bpred taken
-issue:wrongpath true         # Cache
-lsq:size 2                   -cache:il1 il1:128:16:2:1       # Default settings are used
-res:ialu 1                   -cache:dl1 none                 # for everything else.
```

Figure 3 Configuration of the Simplescalar architecture used in the experiments.

The experimentation setup is straightforward. Each benchmark program p is executed $m = 10,000$ times using randomly generated inputs. The number m was chosen to be at the same time large enough to allow more input diversity, and small enough not to affect the GP's runtime too severely — since the fitness function computes a sum of differences of complexity $\Theta(m)$. In each run, the value X_i of the parameters of interest X and the end-to-end execution time $t_i, i \in [1..m]$ of p are captured by reading the value sim_cycle generated by Simplescalar at the end of each execution of the program under analysis. It is worth noting that the end-to-end execution time corresponds to the entire program, not just the function of interest and consequently the parametric timing expression corresponds to the function `main()` — including the use of the function `atoi()` and all initializations such as array initialization; and the function of interest. Measuring the end-to-end execution time of the actual function of interest inside the benchmark program reduces to parsing the Simplescalar trace (generated via `-ptrace`). This adds an unnecessary overhead to

Table 2 The results of the experimental evaluation.

Benchmark p	**Expression w_p by *Eureqa***	**Error e_p**	**Time to find w_p**
bsort100	$35n^2 + 917n + 5.55e4$	± 0.20	2 minutes
cnt	$988n^2 + 100n + 5.82e4$	± 0.22	2 minutes
crc	$180n + 2.04e5$	± 0.20	1.5 minute
fac	$52n + 5.60e4$	± 0.0012	10 seconds
fir (*)	$1706in\_len + 4.45e4$	± 0.57	10 minutes
insertsort	$13n^2 + 970n + 5.54e4$	± 0.28	2.5 minutes
janne_complex	$20.78c - 17.46a + 5.80e4$	± 0.05	10 seconds
matmult	$189n^3 + 1753n^2 + 74n + 6.24e4$	± 0.02	4 minutes
sqrt	$31n + 5.76e4$	± 0.13	5 seconds
st	$2302n + 5.93e4$	± 0.26	2.5 minutes

the experiment because the parametric expression can be derived regardless of whether the end-to-end execution times are measured at the function level, or the whole-program level.

After the m executions, we pass the resulting m-by-$(n+1)$ data matrix M_p of data to the GP to perform symbolic regression. We use the rows $[1..0.9m]$ for training i.e. evolution, and the remaining $[0.9m + 1..m]$ for testing i.e. measuring the error of the derived parametric expression when applied to unseen data. For the GP, we used both *Eureqa Formulize v0.96* (a standalone application based on [12] and freely downloadable from [7]) and *gptips v1.0* (a library for MATLAB based on [14] and freely downloadable from [13]). The results we show in this paper are those obtained by *Eureqa* because it runs faster than the MATLAB-based *gptips*, and it also comes with the paid option of using a cloud cluster to accelerate computation. However we did not use the cloud cluster in our experiments, but it would be interesting to use it for future work. It is worth noting that *gptips* is open-source and allows more customization of the GP such as introducing specialized functions to use in symbolic regression, and also user-crafted genetic operators. Such customization ought to accelerate evolution, but we have not tested this. The experiments took place in a desktop computer of the specifications *Intel(R) Core(TM)2 Duo CPU @2.80 GHz* running *32-bit Windows 7 Professional* with *4Gb of RAM.*

Table 2 summarizes the findings. The GP is run for each benchmark program's data matrix for 10 minutes. Notice that the GP can be left running indefinitely over some data matrix M_p until an optimal symbolic expression w_p is obtained. We argue that 10 minutes is a reasonable time to leave the GP running for the relatively-small problems dealt with in this work — which (i) have small numbers n of independent variables that affect the size of the ASTs generated by the GP, (ii) have small numbers of observations m that affect the cost of the fitness operation, and (iii) have a known polynomial structure and consequently the GP's search space is pruned because only relevant operators are used during evolution. The best solution after 10 minutes is the one shown in Table 2. The error e_p in the parametric expression w_p is also computed and it is the average of the sum of $|w_p(X_i) - t_i|$ in the $0.1m$ unseen runs. So if the error in the parametric expression is $e_p = 0.01$, it means that the average value for the difference $|w_p(X_i) - t_i|$ per unseen run is 0.01. The recorded execution time of the GP in Table 2 is the first instant in time in which the best solution is found. For example, for the benchmark program `bsort100`, the derived expression that is shown in Table 2 has been found after 2 minutes, and did not improve over the remaining 8 minutes during the 10-minute execution of the GP.

The sources of the errors e_p in the table come from the execution-time variations imposed by the hardware architecture. For example, let program p contain a flat loop — and no other loop — that iterates x times where x is some input variable to p; then its parametric expression has the shape $w_p = ax + b$ where a is the execution time of the body of the loop, and b is the execution time of everything else outside the loop. In this case, it is perfectly possible to witness the following scenarios during execution: (i) when $x = 2$, $a = 100$; (ii) when $x = 3$, $a = 95$; and (iii) when $x = 4$, $a = 110$. This could happen if x controls some conditional statement before the execution of the loop, and hence alters the execution history prior to the execution the loop which leads to different values of the execution time a of the loop's body. In cases like this, it is very hard if not impossible for the GP to derive an expression w_p with error $e_p = 0$ because of the complex program-hardware interaction that creates what can be referred to as "noise" in the data matrix M_p.

The GP was unable to find an accurate parametric expression for the benchmark program `fir` with the three parameters specified in Table 1 (error had average magnitude $e_{fir} = 52$) because of the complex interactions between these variables and their effect on program flow — which cannot be captured by a polynomial. The set of parameters was reduced to one element namely in_len while the other two parameters were fixed to their original values 35 and 285 in [10]. The one-parameter expression is the one shown in Table 2 which still has the worst accuracy.

Other than that, the GP was able to approximate the parametric expressions of the benchmark programs with great accuracy despite the hardware exhibiting variations in execution times due to the presence of out-of-order execution, cache, and branch prediction. The expressions were derived in a record time which never exceeded few seconds/minutes per benchmark program. The shape of the parametric expressions corresponds to loops and loop nests in the benchmark programs, and their ASTs are isomorphic with the ASTs derived for the same programs using static parametric WCET in the literature. The obvious differences are in the constant terms and factors which correspond to execution times — which are expected to be different because (i) the hardware platform is different, and (ii) end-to-end testing is used instead of static analysis.

6 Related Work

A relatively recent review on WCET analysis is provided by [17] where different analysis methods, calculation techniques, and available tools are described and compared. In this section we shall exclusively review parametric WCET analysis which has been the topic of research in [3, 16, 9, 11, 15, 6, 2, 5, 1].

Bernat and Burns [3] present a tree-based approach for parametric timing analysis where they derive algebraic expressions for parts of the code, link them together according to the tree structure to build larger expression, and then use software tools such as Maple to simplify the parametric expressions. The technique is manual as no tool was implemented for it, it takes information about the parameters affecting the WCET through a system of code annotation, and uses a timing model supplied by the user or a third-party analysis.

Vivancos et al. [16] use a path-based approach where parametric expressions are derived using fixed-point caching behaviour. The technique is not described in great detail because of the focus on applications of parametric timing analysis. However, the authors do mention limitations in their technique namely the ability to only handle well-structured non-recursive code, and inability to handle cases where there are nested parametric expressions inside loop nests.

Van Engelen et al. [15] introduce a method for parametric WCET analysis using Newton-

Gregory formulae that handles rectangular and non-rectangular loop nests of seemingly arbitrary structures. Unfortunately, the work is purely theoretical and lacks evaluation on actual benchmark programs with the exception of small code snippet. The work here is included in the review for the sake of completeness only.

Coffman et al. [6] use a summation solver called *Emtadel* that automatically derives the number of times the body of a traingular loop nest executes; in the form of a polynomial expression involving the use of min/max operators. The use of Emtadel comes as the answer to the problem of nested parametric loops the authors encountered in [16]. The main problem with this approach is that there is no reference to where to download Emtadel or similar software to reproduce the results obtained by the approach.

Altmeyer et al. [2] present an automatic technique for performing parametric WCET analysis of executable code based on dependency analysis between program variables and parameters. The technique identifies the parameters automatically by performing a simple read/write analysis of memory cells and CPU registers, and using the variables that are read from before written to as the analysis parameters. The dependency analysis is used to form relationships between the variables that control loop iterations and the parameters of the program.

Althaus et al. [1] present a parametric-analysis technique that exploits the usually-regular structure of code written for critical real-time applications — in particular where loops have single entries. They present an algorithm of polynomial-time complexity in practice and exponential-complexity in theory which works on executable code and derives the parametric expressions of loops. For code that does not satisfy the single-entry loop property, they present transformation techniques to force the property which unfortunately adds to the complexity of the approach.

Lisper et al. [9, 5] use polyhedral flow analysis together with symbolic integer-linear programming to derive parametric WCET expressions. The analysis is very accurate but at the cost of prohibitive complexity: the derived parametric-WCET expressions are too complicated to be evaluated instantly during runtime — unless manual simplifications are applied to them. In addition to this, the analysis requires extensive resources as it failed for some problem instances.

The technique we present is different from all previous techniques in the sense that it is based on end-to-end runtime measurements as opposed to static analysis, and uses off-the-shelf GP tools which gives it the advantage of amenability to industrial use. Table 3 shows a comparison of our technique with the techniques in the literature using the following metrics (columns in Table 3).

- **Input.** This metric refers to the way by which the parametric analysis identifies the variables that appear in the parametric expression i.e. they are derived automatically, semi-automatically, or manually.
- **Automation.** This metric refers to whether or not the parametric analysis (excluding input information) is fully-automatic, semi-automatic, or manual.
- **Operators.** This metric refers to the operators supported in the final parametric expression namely arithmetic operators ($+$, $-$, $\times$, $\div$), the conditional operator, and the max/min operators.
- **Limitations.** This metric refers to the main drawbacks of the method that might hinder its use in practice.

Table 3 Comparison with parametric WCET analysis techniques.

Technique	**Input**	**Automation**	**Operators**	**Limitations**
Ours	Manually-derived (or user-provided).	Fully-automatic	Arithmetic	Does not come with safety guarantees.
Bernat and Burns [3]	Manually-derived: user-provided.	Manual	Arithmetic, conditional.	No implementation is available.
Vivancos et al. [16]	Semi-automatic: the analysis looks for all parametric loops only and reads their induction variables.	Fully-automatic	Arithmetic	Well-structured non recursive code only. No support for nested-parametric loops. Not clear how variable bounds on loop-induction variables are related to program input.
Coffman et al. [6]	Semi-automatic: the analysis looks for all parametric loops only and reads their induction variables.	Fully-automatic	Arithmetic, max/min.	Approach based on a software package that is unfindable on the web.
Altmeyer et al. [2]	Semi-automatic: only constant-offset dependencies can be derived automatically.	Fully-automatic	Arithmetic, conditional.	User input needed for to resolve non constant-offset dependencies, such input is very unlikely considering it is based on disassembled binary code.
Lisper et al. [9, 5]	Manual	Semi-automatic	Arithmetic, max/min.	Technique does not scale well, and fails for some problem instances. Parametric expressions are overly-complex which hinders their use in dynamic scheduling — unless manually simplified.
Althaus et al. [1]	Not clear.	Fully-automatic	Arithmetic, conditional.	Complexity polynomial in practice but can grow to exponential in theory.

7 Conclusions and Future Work

In this paper we have shown how to perform parametric timing analysis using genetic programming. End-to-end execution times of the program are recorded together with values of parameters that trigger them, and are then input to the genetic program which performs symbolic regression to discover the parametric expressions. The technique has been successful in deriving accurate parametric expressions.

Genetic programming can be used in conjunction with static-analysis methods for parametric timing analysis to validate their results, and should be of great interest to industry where end-to-end measurements are the way forward to performing WCET estimation. The technique can be explored further in the the following directions (to list but a few): (i) investigate the use of the method on more substantial case studies, (ii) augment the proposed technique by automatic identification of parameters that affect the WCET, (iii) apply the method on programs that run on actual hardware, and (iv) incorporate variable-latency instructions in the parametric expressions.

References

1 E. Althaus S. Altmeyer and R. Naujoks. Precise and efficient parametric path analysis. *SIGPLAN Not.*, 46(5):141–150.

2 S. Altmeyer, C. Humbert, B. Lisper, and R. Wilhelm. Parametric timing analysis for complex architectures. In *Proceedings of the 2008* 14th *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'08)*, pages 367–376, August 2008.

3 G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *Proceedings of the 25th Workshop on Real-Time Programming, Palma, Spain*, June 2000.

4 D.C. Burger and T.M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, 1997.

5 S. Bygde, A. Ermedahl, and B. Lisper. An Efficient Algorithm for Parametric WCET Calculation. In Patrick Kellenberger, editor, *The* 15th *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2009*, pages 13–21. IEEE Computer Society, August 2009.

6 J. Coffman, C. Healy, F. Müeller, and D. Whalley. Generalizing parametric timing analysis. *ACM SIGPLAN Notices*, 42(7):152–154, 2007.

7 Cornell Creative Machines Lab. Eureqa. *http://creativemachines.cornell.edu/eureqa*, April 2012.

8 J.R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems).* A Bradford Book, 1 edition, December 1992.

9 B. Lisper. Fully Automatic, Parametric Worst-Case Execution Time Analysis. In *Proceedings of the 3rd International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 77–80, Porto, July 2003.

10 Mälardalen WCET Research Group. WCET project/benchmarks. *http://www.mrtc.mdh.se/projects/wcet/benchmarks.html*, April 2012.

11 S. Mohan, F. Müeller, W. Hawkins, M. Root, C.A. Healy, and D.B. Whalley. Parascale: Exploiting parametric timing analysis for real-time schedulers and dynamic voltage scaling. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 233–242, 2005.

12 M. Schmidt and H. Lipson. Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85, 3 April 2009.

13 D.P. Searson. GPTIPS: Genetic Programming & Symbolic Regression for MATLAB. *http://sites.google.com/site/gptips4matlab/*, April 2012.

14 D.P. Searson, D.E. Leahy, and M.J. Willis. GPTIPS : An Open Source Genetic Programming Toolbox For Multigene Symbolic Regression. In *Proceedings of the International Multiconference of Engineers and Computer Scientists 2010 (IMECS 2010)*, volume 1, pages 77–80, Hong Kong, 17-19 March 2010.

15 R.A. van Engelen, K.A. Gallivan, and B. Walsh. Parametric timing estimation with Newton-Gregory formulae: Research Articles. *Concurrency and Computation: Practice and Experience*, 18(11):1435–1463, September 2006.

16 E. Vivancos, C. Healy, F. Müeller, and D. Whalley. Parametric timing analysis. *SIGPLAN Not.*, 36(8):88–93, 2001.

17 R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Müeller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-Case Execution-Time Problem–Overview of Methods and Survey of Tools. *ACM Transations on Embedded Computing Systems*, 7(3):1–53, 2008.